LEABHARLANN CHOLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH | TRINITY COLLEGE LIBRARY DUBLIN
Ollscoil Átha Cliath | The University of Dublin

Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

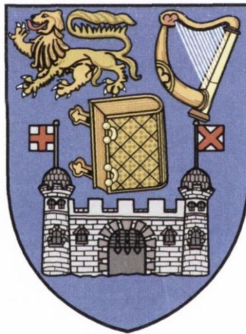
Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

Space & Time Efficient Sparse Matrix Transpose

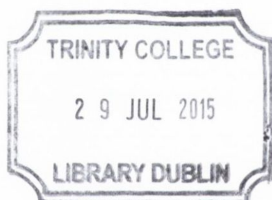
A thesis submitted to the
University of Dublin, Trinity College,
for the degree of
Doctor of Philosophy



Robert Crosbie

2015

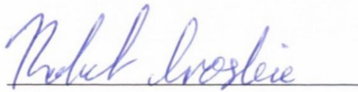
The University of Dublin, Trinity College



Thesis 10609

Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.



Robert Crosbie — Friday 14th November, 2014

Permission

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

Robert Crosbie — Friday 14th November, 2014

Abstract

Matrix operations are fundamental to linear algebra and have many important applications in areas such as simulation of physical systems, economic modeling, linear optimization and numerical analysis. One of the fundamental operations on matrices is the matrix transpose. In many linear algebra applications the matrices are extremely large and require considerable memory to store. Therefore it is desirable to transpose in-place to avoid creating a new matrix which would double the memory usage. Transposing dense matrices in-place has been studied over several decades, and many good algorithms have been found. An area that has been relatively neglected is that of in-place transpose of sparse matrices — that is, matrices where the value of most matrix elements is zero and are stored in a sparse format. The best previous algorithm requires $\Theta(nnz + n)$ time and $\Theta(nnz + n)$ additional space to transpose an $n \times n$ sparse matrix with nnz non-zero entries.

This thesis describes our new family of space-efficient in-place transpose algorithms for sparse matrices stored in the common *Compressed Sparse Row* format. These algorithms require only $\Theta(n)$ space, which is asymptotically better than the best previous algorithm, and greatly reduce the additional space in practice. This is especially important for very large sparse matrices, which are often used to model linear algebra problems at ever finer levels of detail.

Our three best algorithms perform the transpose in $\Theta(nnz+n)$ time and $\Theta(n)$ space. Our Corresponding Row algorithm transposes the 259 sample matrices in 90% of the execution time of the existing Saad algorithm, requiring on average 21% of the memory overhead of Saad. Our HyperPartition with RadixSort algorithm has negligible memory overhead, less than 1% of Saad. This algorithm is efficient for most matrices with and has an average of 90% of the execution time of Saad however, it does not perform well for some matrices. Our Hybrid HyperPartition with RadixSort transpose

takes the best of our two previous algorithms. Our Hybrid algorithm has a memory overhead of just 9.3% of Saad and transposes the matrices on average in 68% of the execution time of Saad in serial and 32% in parallel.

Contents

Abstract v

Contents vii

List of Algorithms xii

List of Examples xv

List of Figures xvii

List of Listings xxi

List of Tables xxiii

1 Introduction 1

My Thesis 1

Contribution 3

Thesis Outline 5

2 Background 9

2.1 Linear Algebra 9

2.2 Dense and Sparse Matrices 11

2.2.1 Dense Linear Algebra 12

2.2.2 Sparse Linear Algebra 13

2.2.3 Direct Methods and Iterative Methods 15

2.2.4 Mathematical Optimisation of Sparse Matrices . . . 16

2.2.5 Software Packages for Sparse Linear Algebra 17

2.3 Matrix Storage Formats 18

2.3.1 Two Dimensional Dense Format 20

2.3.2 One Dimensional Dense Format 21
2.3.3 Morton Ordered Dense Format 21
2.3.4 The *Compressed Coordinate* (COO) Format 23
2.3.5 The *Compressed Sparse Row* (CSR) Format 25
2.3.5 The *Compressed Sparse Column* (CSC) Format . . 25
2.3.6 Block Compressed Row Storage (BCRS) Format . . 26
2.3.7 Compressed Diagonal Storage (CDS) Format 27
2.3.8 The *Recursive Sparse Blocks* (RSB) Format 28
2.4 Memory Hierarchy and Cache Performance 29
2.5 Complexity Analysis of Algorithms 33
2.5.1 $\mathcal{O}(x)$ — Big-O: Upper Bound 34
2.5.2 $\Omega(x)$ — Big Omega: Lower Bound 34
2.5.3 $\Theta(x)$ — Big Theta: Double Bound 35
2.5.4 $\sim(x)$ — Tilde: Tighter Double Bound 35

3 Matrix Transpose **37**
3.1 The Matrix Transpose Operation 37
3.2 Dense Matrix Transpose 39
3.2.1 *In-Place* Dense Matrix Transpose 40
3.2.2 *In Place* Dense Rectangular Transpose 42
3.3 Sparse Matrix Transpose 53
3.4 *Out-of-Place* (OOP) Sparse Transpose 54
3.4.1 Parallel Sparse Matrix Transpose 55
3.4.2 Sparse Matrix Transpose Unit 57
3.4.3 Description of Out-of-Place Transpose Algorithm . 57
3.4.4 Analysis of Out-of-Place Algorithm 61
3.5 The *In-Place* (IP) Sparse Transpose 61
3.5.1 The Saad In-Place Transpose Algorithm 62
3.5.2 Analysis of Saad In-Place Algorithm 66
3.6 Performance Evaluation of Algorithms 67
3.6.1 Matrix Collections – Sample Input Matrices 67
3.6.2 Experimental Setup 69
3.6.3 Presentation of Data 71

3.7 Evaluation of Sparse Transpose Algorithms 72
3.8 Summary 74

4 Space Efficient *In-Place* Sparse Matrix Transpose 75
4.1 In-Place Transpose with Reduced Memory 76
4.1.1 Finding the *old_row_index* in $\Theta(n)$ Space 77
4.1.2 Determine if an Element has Already Been Processed 78
4.2 Generic *In-Place* Sparse Transpose 83
4.3 *In-Place* Transpose with Binary Range Search 85
4.3.1 Binary Range Search 85
4.3.2 Cycle-Chasing Transpose with Binary Range Search 87
4.3.3 Memory Overhead of Transpose with Binary Range Search 89
4.3.4 Execution Time of Transpose with Binary Range Search 90
4.4 In-Place Sparse Transpose with Radix Lookup Table 91
4.4.1 Building the Radix Table 92
4.4.2 Radix Table Lookup 94
4.4.3 Cycle Chasing Transpose with Radix Table Lookup 96
4.4.4 Memory Usage of Radix Lookup Table Transpose . 98
4.4.5 Execution Time of Radix Lookup Table Transpose . 99
4.5 Ensuring In-Row Ordering 105
4.5.1 Sorting Rows with Two Array QuickSort 105
4.5.2 Sorting Sub-Rows with Two Array Insertion Sort . 107
4.5.3 Execution Time of Sorting 107
4.5.4 Runtime Complexity of Sorting Phase 109
4.6 Conclusion 110

5 Corresponding Row Cycle-Chasing Transpose 111
5.1 Constant-Time Row Index Lookup 112
5.2 Using the Corresponding Row 113
5.3 Search and Update Corresponding Row Table 115
5.4 Building the Corresponding Row Table 118
5.5 Corresponding Row Cycle Chasing Algorithm 119

5.6 Cache-Friendly Corresponding Row Algorithm 121
5.7 Corresponding Row Memory Usage 122
5.8 Corresponding Row Algorithm Execution Time 124
5.9 Corresponding Row Performance Evaluation 127
5.9.1 Hardware Counters 127
5.9.2 Branch Misses of CF Corresponding Row 128
5.9.3 Normal Corresponding Row Performance Evaluation 129
5.9.4 Performance Evaluation of Cache-Friendly Algorithm 134
5.9.5 Summary of Normal and Cache-Friendly Evaluation 135
5.10 Factors Influencing Cache Performance 135
5.11 Cycle Length and Cache Performance 139
5.12 Summary 142

6 HyperPartition Sparse Matrix Transpose 145
6.1 The HyperPartition Sparse Matrix Format 146
6.1.1 Grouping Rows 147
6.1.2 Unused Data in CSR Sparse Matrix Format 148
6.1.3 The HyperPartition Structure 149
6.1.4 Using the HyperPartition Format 151
6.2 Converting to HyperPartition Format 152
6.3 HyperPartition Cycle-Chasing Transpose 154
6.4 Sorting HyperPartition after Cycle-Chasing 157
6.5 Converting from HypCSR back to CSR 158
6.6 Heuristic: Choosing Number of Bits to Steal 159
6.6.1 Remaining Bits Heuristic 161
6.7 HyperPartition Memory Usage 162
6.8 HyperPartition Transpose Execution Time 163
6.8.1 HyperPartition Execution Time: Excluding Symmetric 166
6.8.2 Serial Performance of the *Remaining Bits* Heuristic 168
6.9 Parallel HyperPartition Transpose 170
6.9.1 Parallel Sorting Algorithm 170
6.10 Parallel HyperPartition Memory Usage 171
6.11 Parallel HyperPartition Execution Time 173

6.11.1 Parallel HyperPartition vs. Serial Saad 174
6.11.2 Parallel Performance of *Remaining Bits* Heuristic . 175
6.12 Reviewing the *Remaining Bits* Heuristic 177
6.13 Summary 178

7 Further Optimisations — RadixSort and Hybrid Transpose 185
7.1 Most Significant Digit (MSD) RadixSort 186
7.1.1 MSD RadixSort Algorithm 189
7.1.2 Choosing Number of Buckets for Radix Sort 191
7.2 HyperPartition RadixSort Results 193
7.2.1 Parallel HyperPartition with RadixSort Performance 196
7.3 MSD RadixSort Summary 199
7.4 Structural Analysis 200
7.4.1 Detecting Structural Symmetry 207
7.5 Hybrid HyperPartition Transpose Algorithm 208
7.6 Hybrid HyperPartition Transpose Performance 209
7.6.1 Parallel Hybrid HyperPartition Performance 211
7.7 Hybrid HyperPartition Summary 213

8 Conclusion and Future work 215
8.1 Contributions 216
8.2 Future Work 218
8.3 Summary 219

A Matrix Tables 223

B Detailed HyperPartition Performance Graphs 229

C MatrixMarket File Format 271

Bibliography 273

Glossary 295

List of Algorithms

3.1 Dense Square Transpose 40
3.1 Input Matrix M in CSR format 58
3.2 Out-Of-Place sparse matrix transpose 59
3.3 The Saad *In-Place* sparse transpose - PART I: Initialize . . 63
3.3 The Saad *In-Place* sparse transpose - PART II: Main Loop 64
4.1 Generic $\Theta(n)$ *In-Place* Sparse Transpose w/ Row Index Lookup 84
4.2 Index Lookup using Binary Range Search Algorithm 86
4.3 Sparse Transpose with Binary Row Index Range Search . . 88
4.4 Build Radix Lookup Table 94
4.5 Index Lookup using a Radix Lookup Table 95
4.6 Sparse Transpose with Radix Table Row Index Lookup . . 97
4.7 Two Array QuickSort (Median of Three) 106
4.8 Two Array InsertionSort 108
5.1 Searching and Updating the *corresponding_row_table* . . . 116
5.2 Building the *corresponding_row_table* 118
5.3 Corresponding Row Cycle-Chasing Sparse Transpose 120
6.1 Convert from CSR format to HyperPartition format 155
6.2 Cycle Chasing HyperPartition Transpose 180
6.3 Convert HyperPartition back to CSR format 182
6.4 Convert HyperPartition back to CSR format in Parallel . . 183
7.1 Radix BucketSort Algorithm - 256 Buckets 190
7.2 Detect Structural Symmetry Heuristic 209

List of Examples

1.1 Sample Matrices M and its Transpose M^T 1

2.1 Sample Sparse Matrix M 19
2.2 Matrix M in CSR representation 26

3.1 Sample Matrices M and its Transpose M^T 38
3.2 Matrix M in CSR representation 58
3.3 Transposed Matrix M^T in CSR representation 61
3.4 Algorithm 3.3 - Saad-IP Circuit Chasing Step: 1 65
3.5 Algorithm 3.3 - Saad-IP Circuit Chasing Step: 2 65

4.1 Matrix M in CSR representation 77
4.2 Building the Radix Table (in reverse) 95
4.3 Radix Table for Matrix M 95

5.1 Sample Matrix M 111
5.2 Matrix M in CSR representation 112
5.3 Corresponding Row - Step 1 113
5.4 Corresponding Row - Step 2 114
5.5 Corresponding Row - Step 3 116

6.1 Matrix M in CSR representation 150
6.2 Matrix M in HyperPartition CSR representation 150
6.3 Matrix M^T in HypCSR after Hyper Circuit Chasing 181
6.4 Matrix M^T in HypCSR after Hyper-Sorting 181
6.5 Transposed Matrix M^T in CSR representation again 181

7.1 Radix bit Passes of BucketSort 188

List of Figures

2.1 Example Sparse Matrix: ASIC_680k 14
2.2 Morton Order Z-Curve 22
2.3 Morton Order Array 23
2.4 The Block Compressed Row Storage (BCRS) format . . . 27
2.5 Compressed Diagonal Storage (CDS) Format 28
2.6 Morton Order Z-Curve 29
2.7 Memory Hierarchy 30

3.1 Block Transpose of Dense Matrix 41
3.2 Morton Order Array 53
3.3 OOP Algorithm Memory Overhead 73
3.4 OOP Algorithm Runtime 74

4.1 Binary Range Search Algorithm Memory Overhead 90
4.2 Binary Range Search Algorithm Execution Time 91
4.3 Radix Lookup Table Algorithm Memory Overhead 99
4.4 Memory Usage of Radix Table Sizes: $\frac{n}{16}$, $\frac{n}{8}$, $\frac{n}{4}$, n 100
4.5 Memory Usage of Radix Table Sizes: $2n$, $4n$, $8n$, $16n$ 101
4.6 Radix Lookup Table Algorithm Runtime 102
4.7 Algorithm Runtime of Radix Table Sizes: $\frac{n}{16}$, $\frac{n}{8}$, $\frac{n}{4}$ 103
4.8 Algorithm Runtime of Radix Table Sizes: $2n$, $4n$, $8n$, $16n$ 104
4.9 Sort Time stacked on top of Algorithm Time 109

5.1 Corresponding Row Memory Overhead 123
5.2 Corresponding Row Algorithm Execution Time 125
5.3 Corresponding Row Algorithm Execution Time 126
5.4 Branch Misses of CF Corresponding Row Algorithm 129
5.5 Normal Corresponding Row Cache Performance 130

5.6 CF: Corresponding Row: Cache Performance 133
5.7 CF: Corresponding Row: Cache Performance per Element 137
5.8 CF: Corresponding Row: Cache Misses vs. Avg. Cycle Length 141
5.9 CF: Corresponding Row: Avg Cycle Length Execution Time 142

6.1 Stealing bits from the 32 *bit* integer 27,993,600 148
6.2 Converting an element to HyperPartition using 2 decimal places 151
6.3 Bit Masks for converting to HyperPartition. $sb = steal_bits$ 152
6.4 Where information is stored in the HyperPartition Format 154
6.5 Integer Bits Available in the number 7,694 162
6.6 HyperPartition Memory Overhead 164
6.7 HyperPartition Memory Overhead [Close-Up] 164
6.8 HyperPartition Serial Execution Time w/ QuickSort 165
6.9 HyperPartition UnSymmetric Serial Execution Time w/ QuickSort 167
6.10 Serial Hyperpartition with QuickSort with; $k = 1, 3, 6, 10$. 169
6.11 Parallel HyperPartition Memory Overhead 172
6.12 Parallel HyperPartition Memory Overhead 172
6.13 HyperPartition Parallel Execution Time w/ QuickSort . . 174
6.14 Parallel HyperPartition and Serial Saad 175
6.15 Parallel Hyperpartition with QuickSort with; $k = 1, 3, 6, 10$ 176

7.1 HyperPartition Sort Time stacked on top of Algorithm Time 187
7.2 RadixSort Number of Buckets 192
7.3 HyperPartition Serial Memory with RadixSort 194
7.4 HyperPartition Serial Execution Time with RadixSort . . . 195
7.5 HyperPartition Parallel Memory with RadixSort 197
7.6 HyperPartition Parallel Memory with RadixSort - Zoom . 197
7.7 HyperPartition Parallel Execution Time with RadixSort . 198
7.8 Some Structurally Symmetric Matrices 201
7.9 HyperPartition QuickSort Execution Time Just Symmetric 202
7.10 Some Unsymmetric Matrices 203

7.11 Some Triangular Matrices - Lower Triangle of Symmetric 204
7.12 Relative Cache Performance of Struct Sym/Unsym Matrices 206
7.13 Hybrid (Hyp/Corr) Memory Overhead 210
7.14 Serial Execution Time of Hybrid HyperPartition with Radix-Sort 211
7.15 Hybrid HyperPartition Parallel Execution Time with RadixSort 212
7.16 Hybrid RadixSort Parallel Execution Time Relative vs. Serial 213

B.1 Serial Hyperpartition With QuickSort: $k = 1 \rightarrow 10$ 230
B.2 Parallel HyperPartition with QuickSort: $k = 1 \rightarrow 10$ . . . 235
B.3 Serial UnSymmetric HyperPartition QuickSort: $k = 1 \rightarrow 10$ 240
B.4 Serial Symmetric HyperPartition with QuickSort: $k = 1 \rightarrow 10$ 245
B.5 Serial HyperPartition with RadixSort: $k = 1 \rightarrow 10$ 250
B.6 Parallel HyperPartition with RadixSort: $k = 1 \rightarrow 10$ . . . 255
B.7 Serial Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$ 260
B.8 Parallel Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$ . . . 265

List of Listings

2.1 Two Dimensional Dense Example 20
2.2 Sparse Coordinate Example 24
2.3 Sparse Compressed Row Example 25
5.1 Cache Friendly Implementation 122
C.1 MatrixMarket File Format 271

List of Tables

2.1 *Stoker:* Intel Xeon E7-4820 Cache Details 31
2.2 *Stoker:* Intel Xeon E7-4820 Cache Miss Latency from *Calibrator* tool 32

5.1 Monitored PAPI Events 128

8.1 Algorithm Complexities 221

A.1 Matrix Information 224
A.2 Matrix Source 225
A.3 Algorithm Memory Usage (MegaBytes) 226
A.4 Algorithm Execution Time (seconds) 227

Chapter 1

Introduction

`My Thesis:`

> *Sparse matrices in Compressed Sparse Row (CSR) storage format can be transposed in $\Theta(nnz + n)$ time using just $\Theta(n)$ additional space. Additional techniques can be used to further reduce time and space in practice.*

Linear Algebra [Golub 96, Anton 02] and matrix operations are essential in many areas of science, engineering, finance, and numerous other fields. It is therefore important to have fast and efficient linear algebra software. One of the fundamental linear algebra operations on matrices is the Matrix Transpose [Cayley 59, Golub 96]. From a computational perspective, the transpose operation is mainly used to change between row-major and column-major layouts to improve cache reuse and efficiency.

Taking a matrix M, its transpose M^T may be obtained by swapping all the rows with all the columns and vice-versa. As shown in Example 1.1 the first column becomes the first row and the fourth row becomes the fourth column. The elements along the diagonal remain in place.

$$M = \begin{pmatrix} \mathrm{a} & 0 & 0 & 0 & b & 0 \\ c & \mathrm{d} & 0 & 0 & 0 & e \\ 0 & f & \mathrm{g} & 0 & 0 & 0 \\ h & 0 & 0 & \mathrm{i} & j & 0 \\ 0 & 0 & 0 & 0 & \mathrm{k} & l \\ 0 & m & 0 & 0 & n & \mathrm{o} \end{pmatrix}$$

(a)

$$M^T = \begin{pmatrix} \mathrm{a} & c & 0 & h & 0 & 0 \\ 0 & \mathrm{d} & f & 0 & 0 & m \\ 0 & 0 & \mathrm{g} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathrm{i} & 0 & 0 \\ b & 0 & 0 & j & \mathrm{k} & n \\ 0 & e & 0 & 0 & l & \mathrm{o} \end{pmatrix}$$

(b)

Example 1.1: Sample Matrices M and its Transpose M^T

A problem that has been studied since at least the 1950's [Windley 59, Knuth 98] is how a matrix can be transposed in-place. By in-place we mean

that the transpose of the matrix is stored in the same location as the original matrix, and a minimum of additional temporary storage is needed to perform the transpose operation. For dense matrices many good algorithms exist. In particular in-place transposition of a square dense matrix is straightforward and very cache efficient [Lawson 79, Whaley 97, Knuth 98, Goto 02]. Transposing rectangular dense matrices in-place is more complicated, but several efficient algorithms have been developed that move "cycles" of matrix elements with a constant amount of additional space [Laflin 70a, Cate 77a]. There has also been interest in in-place transposition algorithms that achieve greater data locality at the cost of more data movement [Alltop 75].

There has been comparatively very little research into the problem of the in-place transposition of a sparse matrix which is arguably a much more difficult procedure. There have been a number of articles which deal with the Out-of-Place sparse matrix transpose [IBM 76, Gustavson 78b, Pissanetzky 84, Gonzalez-Mesa 13] and numerous implementations (see Section 3.4), however there is very little research to be found on the In-Place Sparse Transpose and only one publicly available implementation [Saad 94]. In this Thesis we aim to bridge that gap with our "Space and Time Efficient In-Place Sparse Matrix Transpose".

A *Sparse Matrix* is a matrix where the majority of the entries in the matrix are zero, and the matrix stored in a condensed format in memory omitting most or all of the zero entries. Storing *all* elements (including zeros) of a large sparse matrix in a dense format in memory is inefficient, and in many cases would require much more memory than is available in the given machine. Sparse matrices generally use more complex structures in memory, which makes in-place transposition more difficult. Popular compact formats such as *Compressed Sparse Row* (CSR) and *Compressed Sparse Column* (CSC) [Duff 86, George 81] (Section 2.3.5) are used, where only the non-zero values in the matrix are stored explicitly. The location (i.e. row and column) index of each non-zero value is stored in auxiliary data structures. For formats such as CSR, accessing the matrix elements via (and maintaining these auxiliary data structures during) the transpose is what makes in-place transposition complicated for sparse matrices.

For a square $n \times n$ sparse matrix, the two important values that affect the time and space required for transposition are the number of rows, or order, of the matrix (n) and the number of non-zero values in the matrix (nnz). The Out-of-Place sparse transpose (Section 3.4) requires $\Theta(nnz + n)$ time and $\Theta(nnz + n)$ auxiliary space. The best published / publicly available in-place transposition algorithm for sparse matrix formats like CSR is Saad-IP which requires $\Theta(nnz + n)$ time and $\Theta(nnz)$ auxiliary space [Saad 94].

The cycle-chasing in-place sparse transpose permutes elements to their correct transposed row. However, elements are not necessarily ordered by column index within the rows. This may be adequate for some applications however, if we wish to ensure elements are in order then we can add a second step to the transpose operation to sort the elements within the rows. This can be done using a comparison sort similar to QuickSort as shown in Section 4.5 which has a time complexity of $\mathcal{O}(nnz \,.\, log(n))$. Alternatively we can use a non-comparative sorting algorithm such as the Most Significant Digit Radix Bucket Sort (described in Section 7.1) which has a time complexity of $\mathcal{O}(nnz \,.\, k)$ [Knuth 98, Biggar 08a, Shutler 08]. If n is the number of rows/columns then $k = log(n)$, which remains constant for any particular integer index used (i.e. for 32 bit integer indexes $k \leq 32$). Thus the complexity of the radix bucket sort essentially becomes $\mathcal{O}(nnz)$.

Contribution

We propose a collection of new *in-place* sparse matrix transpose algorithms which use asymptotically less memory ($\Theta(n)$ compared to $\Theta(nnz)$) while maintaining the same asymptotic time complexity ($\Theta(nnz + n)$) of the existing [Saad 94] in-place algorithm. In most sparse matrices nnz is much larger than n, so the space saving can be significant. The saving is particularly important in cases where the sparse matrix occupies much, or even most, of the available memory and transposition with the current algorithms may be infeasible or even impossible. Indeed, there will always be a need to solve larger problems or solve problems in finer detail which will

result in even larger matrices. Therefore it is essential to support very large matrices as efficiently as possible. In practice our new algorithms use just a fraction of the memory overhead of the existing algorithm while actually improving (often considerably) on the execution time. Our algorithms also provide more efficient matrix format conversion between row-major and column-major orderings, which is the exact same procedure as the transpose operation. We perform an extensive experimental evaluation and comparison of a number of sparse transpose algorithms using a test suite of 259 large matrices taken from real world applications and investigate their performance based on appropriate metrics.

Take for example *nlpkkt240*, the largest matrix in our test suite. Transposing this matrix using an Out-of-Place algorithm requires 4,698 MiB of additional memory. If this almost 5 GiB of additional memory is not available, the existing Saad in-place algorithm [Saad 94] can transpose the matrix in 223 seconds using 1,530 MiB of additional memory. We propose a new *Corresponding Row Transpose* algorithm which reduces the memory overhead from $\Theta(nnz)$ to $\Theta(n)$ while keeping the standard matrix structures. The Corresponding Row algorithm can transpose this large matrix using just 320 MiB of additional memory and in this case also takes 223 seconds to transpose the matrix. Of the 259 matrices in our test suite, our Corresponding Row transpose requires on average just 21% of the memory of Saad and performs the transposition in 90% of the execution time on average.

We propose a further *HyperPartition Transpose* algorithm which maintains the reduced $\mathcal{O}(n)$ memory overhead and internally converts to our new HyperPartition format in order to reduce the memory overhead further and improve on cache reuse to improve performance. The HyperPartition algorithm can transpose this largest matrix with just 3.3 MiB of additional memory in just 80.7 seconds. This represents less than 1% of the memory usage and takes just 36% of the execution time of the existing Saad in-place algorithm. The HyperPartition algorithm has extremely low memory overhead and performs well for most sample matrices however, it does not perform well for some matrices.

We examine matrices for which our algorithm does not perform well, investigate why and propose techniques to improve performance. This results in a *Hybrid Transpose* algorithm which maintains our reduced $\Theta(n)$ space complexity and also maintains the $\Theta(nnz + n)$ time complexity of the existing algorithms. The Hybrid algorithm performs the transpose using on average 9% of the memory overhead of the existing in-place algorithm (less than 3% of the out-of-place algorithm) with an average execution time compared to the existing Saad algorithm of 68% in serial and 38.8% in parallel.

Thesis Outline

This document is structured as follows.

- Chapter 2 covers background information underlying the work in this Thesis. We give a basic overview of Linear Algebra, Dense and Sparse matrices, existing algorithms and software for sparse matrices and matrix storage formats. We also provide an overview of memory hierarchy and caches and finish with a description of complexity analysis and the notation we use to theoretically analyse and discuss algorithms.

- Chapter 3 discusses in detail the Matrix Transpose operation, discusses related research, introduces the existing algorithms for the sparse matrix transpose, describes our experimental setup and gives an analysis of the performance of the existing algorithms.

- Chapter 4 introduces two new algorithms for the cycle-chasing in-place sparse matrix transpose which reduce the memory overhead to an asymptotic space complexity of $\Theta(n)$ compared to the $\Theta(nnz)$ and $\Theta(nnz+n)$ of the existing algorithms. Although the savings in memory can be significant, for these first two algorithms this comes at the cost of an increase in time complexity. This chapter experimentally analyses the performance of the algorithms in terms of execution

time and memory usage compared to the existing algorithms. The increase in complexity is evident in the execution time of the *Binary Range Search Transpose* however the execution time of the *Radix Lookup Table Transpose* is broadly similar to the existing Saad in-place algorithm and is actually slightly faster on average. Chapter 4 also outlines our basic procedure for ensuring elements are arranged in column order within rows after the cycle chasing algorithm.

- Chapter 5 describes our new *Corresponding Row* transpose which uses a lookup table to perform the in-place cycle-chasing algorithm in reduced $\Theta(n)$ memory overhead while maintaining the $\Theta(nnz + n)$ time complexity of the existing algorithms. The novel approach here is that the look up and update of the table can be done in amortized constant $\mathcal{O}(1)$ time. We perform extensive analysis of the performance of two implementations of the Corresponding Row algorithm. We also use hardware counters to look in depth at how the algorithm uses caches compared to existing algorithms. Using this analysis we identify properties of the matrices and the factors such as the cycle length which influence cache usage and performance.

- In Chapter 6 we use the results of the analysis in Chapter 5 to develop a technique to improve cache performance of the in-place cycle-chasing transpose. We introduce our new *HyperPartition* sparse matrix storage format which we can easily and quickly convert to during the transpose. We then introduce our *HyperPartition Transpose* algorithm for the in-place transpose of sparse matrices in the HyperPartition format. We also introduce a heuristic with which to select the best parameter to use for determining the size of partitions in the HyperPartition structure. We analyse the performance of the heuristic used for different values and recommended favourable values. We also introduce a parallel version of the HyperPartition transpose which exploits the data segregation provided by our HyperPartitions to reorder HyperPartition elements in parallel. Extensive performance analysis is also provided which shows that the HyperPartition trans-

pose has negligible memory overhead and has improved execution time performance for many of the input matrices due to improved cache usage.

- Chapter 7 introduces some further optimizations for the HyperPartition in-place sparse transpose. A *Radix Bucket Sort* which exploits the type and layout of of the data in our HyperPartitions to improve the efficiency of the sorting phase of the transpose. We analyse the performance of different bucket sizes and recommend a heuristic for choosing an appropriate bucket size depending on matrix and HyperPartition dimensions.
  The HyperPartition transpose from the previous chapter does not transpose matrices which are structurally symmetric as efficiently as the previous algorithms. In Chapter 7 we investigate the structural layout of the matrices in our test suite and introduce an efficient heuristic test to determine if a particular Square matrix is Structurally Symmetric. We then introduce our *Hybrid HyperPartition Transpose* which uses this test for Symmetry to choose between the HyperPartition and Corresponding Row algorithms. The Hybrid algorithm provides a suitable trade-off and has moderate memory overhead with good overall performance.

- We draw conclusions in Chapter 8, discuss contributions and outline areas for future work. Table 8.1 gives a summary of the algorithm complexities.

- Appendix A contains tables of information on a selection of the largest sample matrices used in experiments. Table A.1 gives details of the dimensions of the matrices. Table A.2 lists the applications and problem domains which produced the matrices. Details of algorithm memory usage and execution time for these matrices are given in Tables A.3 and A.4.

- Appendix B contains detailed graphs of the HyperPartition and Hybrid algorithms in Serial and Parallel with the QuickSort and

RadixSort algorithms for different k values of the Remaining Bits Heuristic.

- Appendix C outlines the MatrixMarket file format.
- Bibliography of references and related work is on page 273.
- Glossary on page 295 defines some common terms used in the document.

Chapter 2

Background

This Chapter contains background information underpinning this research.

Matrix Transpose is one of the basic operations of Linear Algebra [Golub 96, Anton 02]. Section 2.1 gives a brief introduction to Linear Algebra. The Matrix Transpose operation itself is discussed in detail along with the related work and research in that area in Chapter 3. In Section 2.2 we discusses the two main types of matrices which result from linear algebra problems, dense and sparse matrices and the two types of matrix software which are specialised for working with each type. Section 2.3 discusses the main types of matrix storage formats, the data-structures which are used to store matrices in memory on computer systems.

A central thrust of this work is modifying sparse matrix transpose algorithms and data-structures to be more efficient by making better use of caches. Section 2.4 gives a brief overview of the memory hierarchy in modern computer systems, how caches work and how they can be exploited. Section 2.5 outlines complexity analysis and the notation used in this document when discussing and comparing algorithms from a theoretical perspective.

2.1 Linear Algebra

Linear Algebra is a branch of mathematics which involves many fields such as systems of linear equations, vectors, vector spaces, matrices and linear transformations. Systems of linear equations are produced during many varied activities such as analysing the forces on components with Finite Element Analysis [Szab'o 91]. Simulating liquids and gasses with Fluid Dynamics [Harlow 57]. Optimising problems in transportation, telecommunications, and manufacturing with Linear Programming [Schrijver 86],

Simulating current in electric circuits [Nagel 73, Kundert 86], Image Processing [Portnoff 99, Na'mneh 06], and Numerical Analysis [Higham 02, Stoer 02] in many scientific and engineering fields.

As such, linear algebra provides part of the essential foundations in a wide range of areas of engineering, economics, statistics and the various different science disciplines.

Linear Algebra Example

The linear equations that are derived from the problem domains above may look similar to the equations shown in the small example in Equation 2.1.

$$\begin{array}{ccccccc} x & + & 2y & + & 4z & = & 3 \\ 5x & + & 4y & - & z & = & 1 \\ 2x & - & 3y & + & 2z & = & 6 \end{array} \tag{2.1}$$

There are three equations and three unknown variables (x, y, z). The standard technique taught in schools is to use *simultaneous equations* to find the unknown variables. However, this technique does not scale. A better technique is to express the problem in matrix form and use standard Linear Algebra techniques (and software) to solve the problem. These equations can be written in a Matrix form where the square 3×3 matrix A holds the coefficients of the unknown variables, the 1×3 vector array v holds the unknown variables (x, y, z) and the product of A and v is the 1×3 array b, which holds the *right hand sides* of the equations.

$$A = \begin{vmatrix} 1 & 2 & 4 \\ 5 & 4 & -1 \\ 2 & -3 & 2 \end{vmatrix}, \qquad v = \begin{vmatrix} x \\ y \\ z \end{vmatrix}, \qquad b = \begin{vmatrix} 3 \\ 1 \\ 6 \end{vmatrix}. \tag{2.2}$$

The Full system of equations $Av = b$ then becomes:

$$Av = b \quad \Rightarrow \quad \begin{vmatrix} 1 & 2 & 4 \\ 5 & 4 & -1 \\ 2 & -3 & 2 \end{vmatrix} * \begin{vmatrix} x \\ y \\ z \end{vmatrix} = \begin{vmatrix} 3 \\ 1 \\ 6 \end{vmatrix} \tag{2.3}$$

The linear system may be solved by calculating the inverse A^{-1} of

the matrix A and pre-multiplying both sides by this inverse. A and its inverse when multiplied together result in the identity matrix I which when multiplied by a vector or matrix results in the same matrix. The calculation proceeds as follows:

$$\begin{aligned} Av &= b \\ A^{-1}Av &= A^{-1}b \\ Iv &= A^{-1}b \\ v &= A^{-1}b \end{aligned} \tag{2.4}$$

Thus we can find the unknowns by calculating A^{-1}, the inverse of A and pre-multiplying b by A^{-1}. For efficiency, linear algebra algorithms generally do not calculate the full inverse, rather they decompose A into its upper U and lower L components (such that $LU = A$) and perform two triangular solves to calculate the value of the unknowns. This reduces the total number of arithmetic operations required.

$$Lv = b \quad \Rightarrow \quad \begin{vmatrix} 2 & 0 & 0 \\ -3 & 2 & 0 \\ 2 & -3 & 2 \end{vmatrix} * \begin{vmatrix} x \\ y \\ z \end{vmatrix} = \begin{vmatrix} 4 \\ 2 \\ 6 \end{vmatrix} \tag{2.5}$$

We use a *triangular solve* when we have a lower triangular matrix L multiplied by a vector v as in Equation 2.5. A triangular solve makes it easier to calculate the values of the unknowns in the vector v. In this case we can simply read the value of the variable x given that the equation is $2x + 0y + 0z = 4$, thus $x = 2$. This value of x can then be used to find the value of y and then z. The same method can be used with an upper triangular matrix U.

This method of finding the unknown variables in systems of linear equations is just one of the common uses of Linear Algebra. There are many many other uses of Linear Algebra.

2.2 Dense and Sparse Matrices

There are two main types of matrices; Dense Matrices and Sparse Matrices.

The main reason for distinguishing between dense and sparse matrices

is how they are stored in memory on computer systems and how the linear algebra algorithms operate on them. In a dense matrix, all the elements in the matrix are stored contiguously in memory. We know the exact layout of the matrix in memory such that every element can be indexed directly.

A sparse matrix is a matrix where the majority of the entries have a value of zero. Sparse matrices tend to be quite large so storing all those zeros in memory (as in dense) is inefficient. Performing arithmetic (adding, multiplying) with all those zeros is also inefficient. Therefore sparse matrices are stored in memory using a compact format such as *Compressed Sparse Row* (CSR) [Duff 86, George 81] (see Section 2.3.5) where just the non-zero values from the matrix are stored. Additional data structures store information on the layout and structure of the matrix. Furthermore, sparse matrix software is designed with this sparsity in mind and use the compact storage formats to reduce the number of arithmetic operations required to perform the algorithms by only accessing the non-zero values.

The main storage formats and data-structures used for both dense and sparse matrices are outlined in Section 2.3.

2.2.1 Dense Linear Algebra

Dense Linear Algebra refers to the class of linear algebra algorithms and software which operate on matrices stored in dense format in memory. The Basic Linear Algebra Subprograms (BLAS) define a standard set of interfaces for performing common linear algebra tasks. The BLAS is divided into three categories. The level 1 BLAS [Lawson 79] consists of scalar and vector routines (dot product, vector-vector multiply). Level 2 BLAS [Dongarra 88] consists of routines which deal with one matrix and one or more vectors (matrix-vector multiply). Level 3 BLAS [Dongarra 90] consists of more complicated single matrix routines and routines including two or more matrices (matrix-matrix multiply). LAPACK (Linear Algebra PACKage) [Anderson 90, Anderson 99] builds on the BLAS and provides routines for solving systems of linear equations, least squares, eigenvalues and routines for factorizing matrices. The BLAS and LAPACK have been

very successful in standardising the interfaces for linear algebra routines.

The basic BLAS [Lawson 79, Dongarra 88, Dongarra 90] just provides a reference implementation of the algorithms and interfaces, they are not tuned for efficiency. With dense matrix algorithms, there are a lot of techniques such a blocking and paneling which can be used to exploit hardware resources, caches and TLBs [Nishtala 04, Gustavson 12]. There are a number of efficient serial and parallel dense linear algebra libraries available which implement the BLAS interfaces and often include routines from LAPACK and other useful routines. Some of these optimized implementations are IBM ESSL [IBM 70], Intel MKL [Intel 93], AMD ACML [AMD 03], ATLAS [Whaley 98, Whaley 01] and GotoBLAS [Goto 02, Goto 08a, Goto 08b].

The BLAS libraries are highly efficient for dense systems or systems with a specific dense structure (such as banded, skyline, etc.) as they can exploit the logical, sequential structure of the matrix. As mentioned above, dense routines however, are generally not appropriate for handling sparse matrices; The high proportion of zeros means that they require excessive amounts of space to store them in memory, and any dense routine will spend a high proportion of its time executing unnecessary operations involving zero.

2.2.2 Sparse Linear Algebra

A large proportion of linear systems that occur in real world applications tend to be sparse, in which the vast majority of the entries are zero. For example, Figure 2.1 shows the $ASIC_680k$ matrix from our test suite. A square matrix with 682,862 rows and 3,871,773 non-zero values, with an average of 5.7 elements per row/column. With an average of 5.67 elements per row, this matrix is 99.999% sparse and would require 3,474 GiB to store in memory in a full dense format however only requires 47 MiB to store in the Compressed Sparse Row format. The definition of a sparse matrix also states that the sparsity can be exploited, either to reduce the amount of storage required to represent the matrix in memory or to reduce the amount of computation required when operating on the matrix or both.

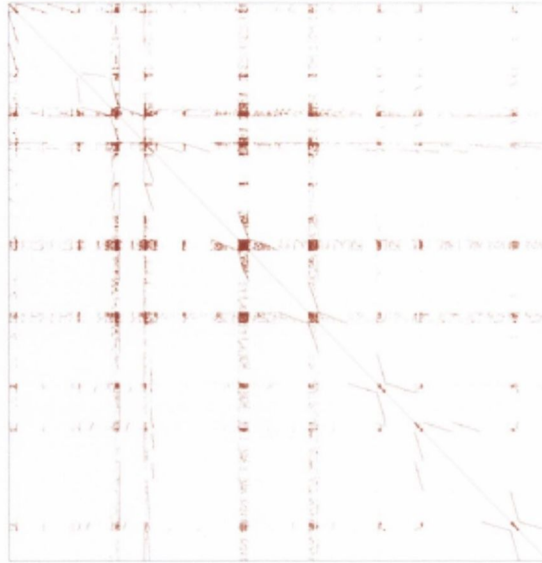


Figure 2.1: Example Sparse Matrix: ASIC_680k is a 682,862 × 682,862 matrix with 3,871,773 non-zero values meaning just 0.0001% of the elements are non-zero (5.7 elements per row on average). Pixels in the image represent locations in the matrix which contain non-zero elements. As the size of the matrix is much larger than the dimensions of the image, a single pixel indicates that there is at least one non-zero element within a block of elements. This matrix would require 3,474 GiB to store in a dense format however requires just 47 MiB to store in the CSR format.

Section 2.3 outlines a number of the common sparse matrix storage formats. We will be working mainly with matrices stored in the CSR format (Section 2.3.5).

The memory layout of sparse matrices is much less amenable to random access than dense matrices. It is not possible to directly index every element in the matrix without going through additional meta structures, it may also be necessary to scan through parts of the array. It is not possible to directly calculate the location of an element in memory based on its row and column index, nor indeed know in advance if that particular element is zero or non-zero. This makes it difficult for algorithms to take advantage of processor features such as caches. As a result of this, sparse algorithms are not as efficient as their dense counterparts as they are highly dependent on

memory bandwidth and latency. Thus it is very important to investigate methods to make sparse matrix algorithms more efficient.

2.2.3 Direct Methods and Iterative Methods

There are two main approaches to solving sparse systems, direct methods and iterative methods. A direct method [George 81, Bunch 76, Duff 86, Davis 06, Gould 05, Stewart 01] performs calculations based on Gaussian elimination to carry out the algorithm (decomposition, etc.). The direct sparse method skips calculations on zeros as only non-zeros are stored in the sparse structure.

An iterative method [Saad 03, Houscholder 52, Conrad 77] takes a different approach: it takes an initial *guess* at the unknown variables in the x vector, then uses a number of mathematical techniques (such as conjugate gradient [Hestenes 52, Saad 03, Straubhaar 08]) to see how close the guess was and generate a better estimate of the unknown variables in x. The process is repeated *iteratively* until a set of values is found which are within some pre-defined limits of precision. Iterative methods depend primarily on sparse matrix-vector multiplication and mathematical techniques for refining the estimates. Sparse matrix-vector multiplication has already received a great deal of attention in the research community [Demmel 01, Vuduc 05, Lee 08].

Iterative methods can often solve the problem in less time and often using considerably less memory than direct methods. However, iterative techniques can sometimes be unstable (values may explode to infinity or degrade to zero). It is also possible that the algorithm may not converge using an iterative method, depending on the matrix.

There are a number of advantages to using Direct Methods with sparse matrices. Direct methods are guaranteed to complete with a solution in an amount of time relative to the number of rows/elements in the matrix (provided a solution exists). Thus direct methods are more predictable in their running time, and can find solutions in some cases where iterative methods fail. Furthermore, it is often required to solve the same set of

linear equations with multiple different right hand side arrays b (boundary conditions). With a direct method, we can decompose the matrix into L and U once, and then just perform the triangular solve for each different boundary condition. It is generally possible to perform multiple triangular solves at the same time in order to improve the cache reuse of the triangular matrix.

2.2.4 Mathematical Optimisation of Sparse Matrices

The majority of research into optimising sparse linear algebra algorithms [George 81, Bunch 76, Duff 86, Davis 06] has approached the problem from a mathematical perspective. We can see this in the work on preconditioners and convergence algorithms such as conjugate gradient [Hestenes 52] from the iterative methods.

Much of the research on optimising direct methods for sparse linear algebra focuses on reordering techniques [Gould 05, Duff 86]. These reordering techniques use Linear Algebra, Graph Theory and Combinatorics to produce an ordering to swap the rows and columns of the matrix to move elements into positions which will make operations on the matrix more efficient. There are a number of packages such as Scotch [Chevalier 08], Metis [Karypis 98, Gupta 97] and CHOLMOD [Chen 08] for working with graphs which provide methods specifically for reordering sparse matrices.

There are two main goals with the current techniques. The first is to reduce the bandwidth of the matrix (Cuthill McKee [Cuthill 69, Cuthill 72]), that is, swap rows and columns so that the entries in the matrix are closer to the diagonal line. The second technique is to reorder the rows and columns to reduce fill-in (AMD [Amestoy 96, Larimore 98], Nested Dissection [Karypis 98, Gupta 97, Bornstein 99]). Fill-in is where operations on the matrix cause entries which were originally zero, to become non-zero. Fill-in is a major difficulty for sparse matrix algorithms because they modify the structure of the matrix, space for new elements needs to be created in the middle of the compact matrix arrays. Inserting a new entry into a sparse data structure can be very expensive. Fill-in is not a problem in

dense matrices because there is already a place in the matrix for the new non-zero value. An additional benefit of the Nested Dissection technique is that it also reduces some of the dependency between rows and columns in the matrix, which means that different parts of the matrix can be (largely) decomposed in isolation.

2.2.5 Software Packages for Sparse Linear Algebra

Sparse linear algebra is an important problem with many practical real-world applications. As a result, a great deal of research and engineering effort has been devoted to constructing efficient libraries that implement both direct and iterative methods. Many of these libraries have been built using Fortran [Backus 56] or low-level C [Kernighan 88]. In this section we outline some of the many packages and libraries which are available for working with sparse matrices.

The Harwell Subroutine Library (HSL) [Gould 04] is a large collection of FORTRAN routines which implement many different variants of the sparse linear algebra algorithms. The NIST Sparse Blas [Remington 96] provides a basic implementation of the BLAS for sparse matrices. The BeBOP [Demmel 01] group in Berkley have developed a number of optimised packages for Sparse Matrix-Vector Multiplication: OSKI [Vuduc 05] and SPARSITY [Im 04]. Tim Davis has developed a number of packages which work with sparse matrices CSparse [Davis 06], LDL [Davis 05a], UMFPACK [Davis 97, Davis 04] and CHOLMOD [Chen 08] which are all part of SuiteSparse [Davis 05b].

The Sparskit [Saad 94] package which we have mentioned before is a basic tool-kit for sparse matrix computations. Oblio [Dobrian 04] is a sparse toolkit for solving linear systems. PETSc [Balay 97] is a tool for solving applications modeled by partial differential equations which uses sparse matrix linear algebra algorithms. Spooles [Ashcraft 99] is the SParse Object Oriented Linear Equations Solver. SuperLU [Li 03, Li 05] is another package for sparse matrix decomposition and solving systems of sparse linear equations. It includes versions for shared memory and dis-

tributed memory systems. MUMPS [Amestoy 98] is a Multifrontal Parallel sparse direct Solver which runs in parallel using MPI [Snir 95, Snir 98]. TAUCS [Toledo 03] implements a number of sparse matrix algorithms using Cilk [Blumofe 95] for multi-threading. The Watson Sparse Matrix Package (WSMP) [Gupta 01] is a package from IBM for solving large sparse linear systems. PARDISO (PARallel DIrect SOlver) [Schenk 01] provides a number of routines for sparse matrix factorisation.

In addition to the many packages listed above, most of the big mathematical software systems such as Matlab [MATLAB 10], Mathematica [Wolfram 03], Octave [Eaton 09], Sage [Ercal 10], Magma [Bosma 97] and Maple [Monagan 05] also include functionality for working with sparse matrices. In many cases they simply include (or can be configured to work with) a number of the sparse matrix packages listed above.

2.3 Matrix Storage Formats

This section describes some of the common data structures and storage formats for representing dense and sparse matrices in memory on computer systems.

The matrices used in sparse linear algebra are often extremely large; matrices with tens of thousands or millions of rows and columns are common. Although the order of the matrix may be large, typically most of the values are zero. To reduce memory requirements and processing time, matrices are stored in so-called *compressed* formats. *Compact* may be a better term than *compressed* as the format does not use any compression algorithm such as Huffman [Huffman 52] or LZW [Ziv 77, Welch 84]. The data is simply stored without the zero elements. That is, only the non-zero values are stored explicitly in memory. Arrays or other data structures with additional meta-information define the structure and layout of the matrix.

Compact formats such as CSR are very important for sparse matrices. As we mentioned above in Section 2.2.2 the $ASIC_680k$ matrix in Figure 2.1 would require 3,474 GiB to store in a dense format however requires just 47 MiB to store in the CSR format. Aside from saving space, compact

formats such as CSR also help avoid a considerable amount of unnecessary arithmetic (adding/multiplying by zero) during matrix algorithms.

There are many different formats in which a sparse matrix can be stored in memory and each option has its advantages and disadvantages. These formats dictate the amount of memory required to store the matrices and may have considerable influence on the run-time speed and efficiency of the linear algebra algorithms operating on them. In general the algorithms need to be altered to varying degrees in order to make more efficient use of resources when operating on particular storage formats.

However, some storage formats will often be more beneficial for certain algorithms and some for others. For example, the compressed sparse row (CSR) format is more appropriate for algorithms which operate on matrices by rows, as the format facilitates easy access by rows (but not by columns). Alternatively, a format which stores the matrix in column order (such as CSC) is more appropriate for routines which access the matrix by columns.

$$M = \begin{pmatrix} a & 0 & 0 & 0 & b & 0 \\ c & d & 0 & 0 & 0 & e \\ 0 & f & g & 0 & 0 & 0 \\ h & 0 & 0 & i & j & 0 \\ 0 & 0 & 0 & 0 & k & l \\ 0 & m & 0 & 0 & n & o \end{pmatrix}$$

Example 2.1: Sample Sparse Matrix M

Example 2.1 gives a pedagogical example of a sparse 6×6 matrix M. This matrix is used to demonstrate how a number of the formats below store the matrices in memory and will be used throughout the document to demonstrate how the transpose algorithms operate. The elements of the matrix are represented by the sequential letters of the alphabet from 'a' to 'o' ordered by row. We use letters rather than numbers for matrix values to ease legibility of the examples and make a clear distinction between the non-zero values of the matrix elements, the row/column indexes and pointer values. These letters can be thought of as representing the floating

point values in the matrix. The matrix has ($n = 6$) rows, ($m = 6$) columns and is sparse with ($nnz = 15$) non-zero elements. There are also 21 zeros. In all our examples we use zero-based indexing as in the C programming language. Thus the row indexes of an $n \times n$ matrix run from $0 \rightarrow (n-1)$ inclusive.

2.3.1 Two Dimensional Dense Format

The two-dimensional array is a common format for storing dense matrices. All the values in the matrix are stored together in memory in a two dimensional array. Listing 2.1 shows an example of our dense 6×6 matrix M from Example 2.1 being statically allocated as a two dimensional array $A[][]$.

```
1  A[6][6] = { { a, 0, 0, 0, b, 0 },
2              { c, d, 0, 0, 0, e },
3              { 0, f, g, 0, 0, 0 },
4              { h, 0, 0, i, j, 0 },
5              { 0, 0, 0, 0, k, l },
6              { 0, m, 0, 0, n, o } };
```

Listing 2.1: Two Dimensional Dense Example

The two dimensional format is very simple; elements are easy to access directly as matrix element a_{ij} can be found at array position $A[i][j]$. In this representation, the first index i, specifies the row and the second index j, specifies the column.

One difficulty with the two dimensional dense array regards the ordering. In the C programming language, two dimensional arrays are stored in a *row-major* ordering as we have shown in Listing 2.1. However in the Fortran programing language, two dimensional arrays are stored in a *column-major* ordering. This can cause confusion between libraries and algorithms written in the two languages if this difference is not known. However, working between the two languages is straightforward once this difference is addressed; either restructure the ordering of the matrix or swap the column and row indices in the algorithms.

An issue with naïvely using the default row-major ordering of two dimensional arrays in `C` is that many linear algebra algorithms are column orientated, so accessing the row-major `C` arrays by columns is bad for memory locality. This problem can easily be worked around in a number of ways; restructure the algorithm to operate by rows (which often can be simply done by swapping the inner and outer *for* loops) or storing the matrix in a transposed form in the array such that it is stored in a column-major order. M^T in row-major is identical in memory to M in column-major. Alternatively a one dimensional dense format can be used with the elements ordered appropriately.

2.3.2 One Dimensional Dense Format

The one-dimensional array is another popular format for storing dense matrices. This uses a single one-dimensional array of size $nrows \times ncols$ to store the whole matrix contiguously in memory. This format avoids the row-major and column-major issues of the two-dimensional array. We can choose ourselves whether to store the matrix by rows or by columns.

Rather than accessing element a_{ij} as $A[i][j]$ as in the two dimensional array, we calculate the position of the element in the array using arithmetic. Thus the element a_{ij} can be found at position $A[(i * nrows) + j]$ if the matrix is stored in a row-major format or $A[i + (j * ncols)]$ if the matrix is stored in a column-major format.

Column-major algorithm implementations seem to be slightly more common, perhaps due to the prevalence of codes written in Fortran. However, there are many algorithm implementations which are column-major and there are many which are row-major. There are also certain algorithms which by their nature are specifically row or column orientated, thus both orientations are important for good data locality.

2.3.3 Morton Ordered Dense Format

A format which is related to the one dimensional array dense format is the Morton Ordered Dense Format [Morton 66, Wise 01]. This format avoids

the problem of having to choose a row-major or column-major ordering by storing the elements in the matrix in a sequence of 2×2 hierarchical blocks. This ordering means that elements that are close to each other horizontally and vertically, in adjacent sub-blocks of rows and columns are also stored closer to each other within the one-dimensional array. This gives better cache reuse for algorithms which operate on the matrix in blocks.

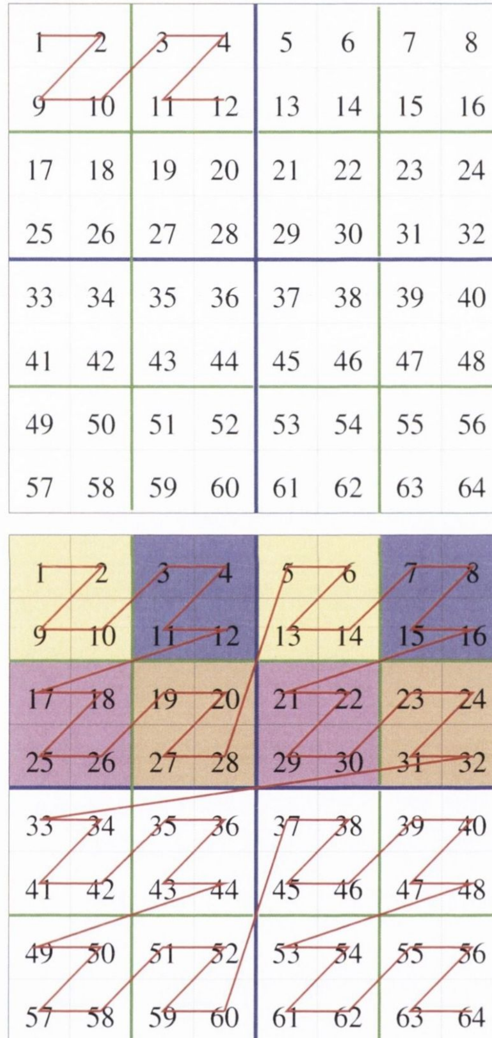


Figure 2.2: Morton Order Z-Curve Matrix

Figure 2.2 shows an 8×8 matrix. At each level of the Morton ordering the matrix is bisected horizontally and vertically as shown in Figure 2.2

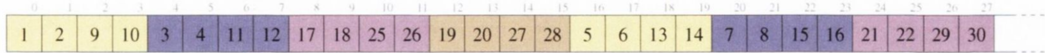


Figure 2.3: Matrix stored in Morton Order

by the blue lines first dividing the matrix into four, 4×4 sub-matrices, which are further divided by the green lines into 2×2 blocks. Each block is recursively stored sequentially in memory in order from top-left to top-right to bottom-left to bottom-right. Thus giving the *Z-Curve* path that the elements are ordered in, as shown by the red line.

The matrix is repeated on the right of Figure 2.2 showing the full path of the Z-Curve through the matrix. 2×2 blocks which will be stored together are highlighted in different colours. Figure 2.3 shows the first 28 elements of the array storing the morton ordered matrix. The 2×2 blocks are again highlighted by colour.

The advantage of Morton ordering is that the position of each element can be calculated by simply interleaving the bits of the binary representation of the row and column indices. Take for example element 20 stored at row 3 (index 2 = 010), column 4 (index 3 = 011). Interleaving these index bits gives: 001101 = index 13. Thus element 20 is stored at index 13 in the array.

Variants of the Morton Ordering Format are used in a number of techniques for fast matrix multiplication [Strassen 69, Coppersmith 90, Valsalam 02].

2.3.4 The *Compressed Coordinate* (COO) Format

The Sparse Coordinate Format (COO) is the basic sparse matrix storage format. A triplet of information is stored about every non-zero element in the matrix: The column index of the element, the row index of the element and the non-zero value of the element.

Listing 2.2 shows the static allocation of the matrix M in COO format. Additional whitespace is included to group elements by row for readability. The sparse coordinate format consists of three arrays of size (nnz), the

row_indexes[], the *col_indexes*[] and the *non_zeros*[]. These arrays store respectively the row index, column index and non-zero value of the elements of the matrix. No zero values are stored in the matrix, the corresponding row and column index for each non-zero value indicates the location of that value in the original matrix M in Example 2.1.

```
1  row_indexes[15] = { 0, 0,   1, 1, 1,   2, 2,   3, 3, 3,   4, 4,
   5, 5, 5 };
2  col_indexes[15] = { 0, 4,   0, 1, 5,   1, 2,   0, 3, 4,   4, 5,
   1, 4, 5 };
3    non_zeros[15] = { a, b,   c, d, e,   f, g,   h, i, j,   k, l,
   m, n, o };
```

Listing 2.2: Sparse Coordinate Example

The format does not require elements be stored in order, however elements would usually be stored in order in a row or column major ordering. The coordinate format is not an efficient layout for most sparse algorithms as it is not possible to determine in advance where the elements of a particular row or column are located in the matrix, the whole matrix structure must be searched to check for the existence of and find that element. If elements are ordered a binary search would take $\mathcal{O}(log(nnz))$ time. Finding an element in an unordered COO matrix would take $\mathcal{O}(nnz)$ time.

The sparse coordinate format is sometimes used when initially constructing a sparse matrix in memory. Many of the sparse matrix file formats such as the MatrixMarket format (Appendix C) store elements in a coordinate format. The sparse coordinate format is rarely used for computation. A COO matrix is generally converted to another format such as CSC or CSR before computation. The Saad In-Place transpose algorithm (Section 3.5) is one algorithm that does use this Sparse Coordinate format. Saad internally converts to COO in order to perform the in-place cycle-chasing transpose.

2.3.5 The *Compressed Sparse Row* (CSR) Format

The *Compressed Sparse Row* (CSR) Format and its sister format, *Compressed Sparse Column* (CSC) [Duff 86, George 81] are two of the most common sparse matrix representations which are supported by many of the Sparse Linear Algebra packages listed in Section 2.2. For consistency we arbitrarily standardize on the Compressed Sparse Row format for the transpose algorithms in this document.

Compressed Sparse Row stores the *non-zero* values of the matrix in one contiguous array. Additional arrays are used to store meta-data relating to the structure of the matrix; the locations of the start of each row and the column index corresponding to each element. The elements are stored in a row-major order, as is evident by name of the structure. The Compressed Sparse Column format stores elements in column-major order.

Given that the CSR format stores elements in row order and CSC stores elements in column order, they essentially represent the transpose of the other. The procedure to transpose a sparse matrix is identical to the procedure for converting between CSR and CSC formats. Hence the transpose algorithms presented here can also be used to convert between row-major and column-major orderings. In fact, the Out-of-Place transpose procedure in the Sparskit2 package is called `csrcsc()`.

```
1      row_ptrs[7] = { 0,       2,        5,       7,         10,
   12,        15 };
2  col_indexes[15] = { 0, 4,    0, 1, 5,  1, 2,    0, 3, 4,   4, 5,
   1, 4, 5       };
3    non_zeros[15] = { a, b,    c, d, e,  f, g,    h, i, j,   k, l,
   m, n, o       };
```

Listing 2.3: Sparse Compressed Row Example

Listing 2.3 shows the static allocation of a sparse matrix in CSR format. Whitespace is added for readability to group elements and pointers by rows. The format is very similar to the Compressed Coordinate format (Section 2.3.4). The *col_indexes*[] and *non_zeros*[] arrays are identical in both formats.

The *row_index* is not stored directly in the CSR format as in the COO format. In CSR the location of the start of each row in the *row_ptrs*[ ] array is stored as an index into the two *non_zeros*[ ] and *col_indexes*[ ] arrays. The *row_ptrs*[ ] array is size $(nrows + 1)$ rather than $nrows$ because it requires an extra entry in order to indicate the end of the last row.

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
non_zeros	=	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
col_indexes	=	0	4	0	1	5	1	2	0	3	4	4	5	1	4	5	
row_ptrs	=	0_0		2_1			5_2		7_3			10_4		12_5			15

Example 2.2: Matrix M in CSR representation

Example (2.2) shows the matrix M in CSR, this time using the representation which we will use for describing the transpose algorithms in the remainder of this document. In the example the subscript values in the *row_ptrs*[ ] array give the row number. These subscript values are shown only for the sake of clarity.

The CSR format is typically used when we wish to access the matrix by rows, which is common in certain matrix operations. In order to access row 3, say, we lookup the position of that row in the *row_ptrs*[ ] array. We then know that row 3 lies between *row_ptrs*[3] and *row_ptrs*[3 + 1] (i.e. locations 7 through 9). We can then read that row 3 has the values (h, i, j) with column indexes $(0, 3, 4)$ respectively.

2.3.6 Block Compressed Row Storage (BCRS) Format

Some types of linear systems of equations result in sparse matrices which are comprised of numerous small dense blocks of values in a regular pattern where all the blocks are the same size. The discretization of some partial differential equations which have several degrees of freedom often result in such matrices as do some matrices arising from Finite Element Analysis (FEA). Figure 2.4 shows a matrix where *all* the non-zero elements occur in

small 2×2 blocks, thus the number or rows/columns per block, $n_b = 2$. In other matrices, blocks may occur with different values of n_b.

$$\begin{bmatrix} 1 & 3 & 6 & 7 & 0 & 0 \\ 2 & 1 & 8 & 2 & 0 & 0 \\ 0 & 0 & 1 & 4 & 0 & 0 \\ 0 & 0 & 4 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 & 7 & 2 \\ 0 & 0 & 3 & 6 & 4 & 5 \end{bmatrix}$$

Figure 2.4: The Block Compressed Row Storage (BCRS) format

The Block Compressed Row Storage (BCRS) format is a modification of the Compressed Row Storage format where all the values of each block are stored contiguously together in the $non_zeros[\,]$ array. The $col_indexes[\,]$ array then just holds the column index of the top left element in the block. The $row_ptrs[\,]$ array thus becomes an array of pointers to blocks. The BCRS format has slightly lower memory usage than CRS. The $non_zeros[\,]$ array is the same length. However the length of the $col_indexes[\,]$ becomes $\frac{nnz}{{n_b}^2}$ and the length of the $block_ptrs[\,]$ array becomes $\frac{n}{n_b} + 1$ where n_b is the number of rows/columns per block. For a matrix with this structure it is implied that n_b is a factor of both n and m.

The benefit of the Block Compressed Row Storage format is that algorithms can exploit the blocked format in order to gain improved cache performance. Such algorithms would of course need to have been written to support matrices stored in the BCRS format and operate on blocks of size n_b. Variants of the BCRS format supports blocks of variable size in the matrix.

2.3.7 Compressed Diagonal Storage (CDS) Format

If a sparse matrix is *Banded*, in that all the non-zero elements are centered along the diagonal and subdiagonals then we can use the Compressed Diagonal Storage (CDS) Format. The *diagonal* is the line of matrices from

the top left of the matrix to the bottom right where the row and column indices are the same, $i = j$, shown in red in Figure 2.5. A *subdiagonal* is a line of elements that runs parallel to the diagonal. The Compressed Diagonal Storage scheme stores the diagonal in an array of size n. Each subdiagonal is stored in an additional array of size n. Thus, if all the non-zero elements of a matrix were located in the diagonal and the two subdiagonals either side of the diagonal then we could store the matrix in three arrays of size n corresponding to $d-1$, d and $d+1$.

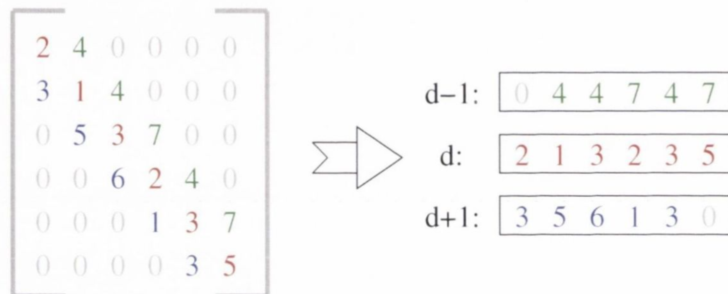


Figure 2.5: Compressed Diagonal Storage (CDS) Format

Figure 2.5 shows an example of a banded matrix, all the non-zero elements are along the diagonal and the lines beside the diagonal. Thus we can store all the non-zero elements in three arrays of size n. The Compressed Diagonal Storage format may include a number of additional zero elements which occur in the diagonal and subdiagonals. However additional arrays of meta-information are not required to identify the location of elements, meaning CDS often requires less memory than other sparse formats. Aside from lower memory usage than dense, the benefit of the CDS format over other sparse formats is that we know the location in memory of every element and can exploit this in our algorithms to improve cache performance.

2.3.8 The *Recursive Sparse Blocks* (RSB) Format

The Recursive Sparse Blocks (RSB) [Martone 10a, Martone 10b, Martone 11] Format is a cache friendly format for sparse BLAS operations. RSB parti-

tions the sparse matrix into quadrants using a *quad-tree* [Finkel 74, Wise 01] structure. A quad-tree is a tree data structure where each internal node has exactly four children. Figure 2.6 shows where matrix quadrants would be stored in a 1-level quad-tree. In the RBS format the Matrix is recursively divided into quadrants and the sparse sub-matrix blocks are stored in the leaves of the tree in standard COO or CSR format.

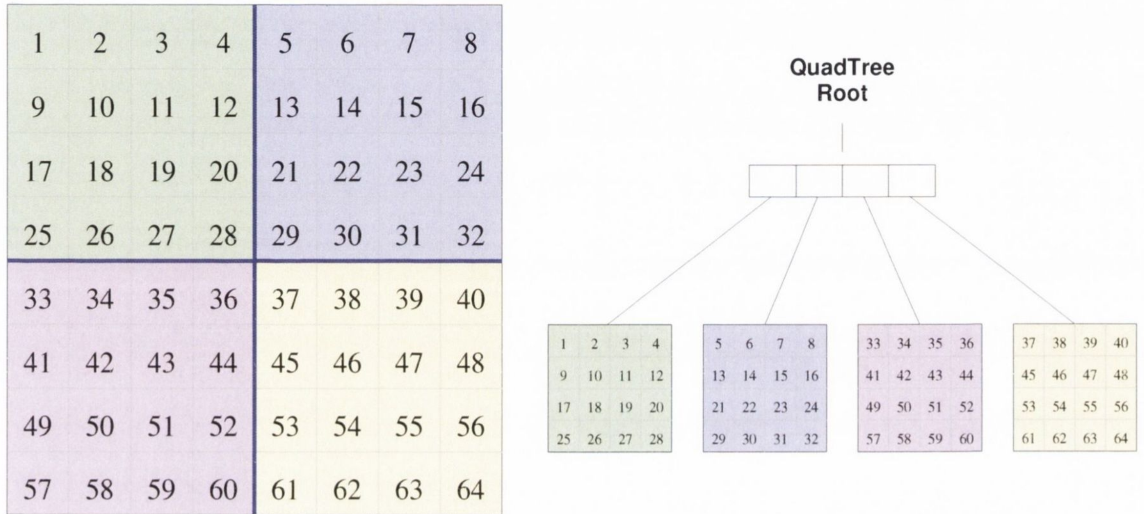


Figure 2.6: Recursive Sparse Blocks QuadTree

The procedure to convert to the Recursive Sparse Blocks format is expensive. The format is intended to improve the performance of Sparse Matrix-Vector Multiplication *SpMV* which is used repeatedly during iterative solvers, meaning that the cost of the conversion may be recouped over multiple multiplications.

2.4 Memory Hierarchy and Cache Performance

Over the past few decades computer processor speeds have increased rapidly year on year closely following those increases predicted by Moore's Law [Moore 65, Schaller 97]. Processors today are orders of magnitude

more powerful than those of a few decades ago. Unfortunately memory bandwidth and memory latency (how long it takes to bring data from main memory into the processor) have not been increasing at the same rate and have lagged behind. This has led to what is referred to as the *Memory Wall* [Wulf 95, McKee 04] or *Memory Gap* [Wilkes 01, Fernandes 02] — the gap in performance between processors and memory.

When development of numerical codes was becoming more common on early computer systems in the '50s and '60s, it only took a few processor cycles to bring data from main DRAM memory into the processor to perform calculations. Due to the fact that processor speeds have increased at a much higher rate than memory speeds, todays computer systems can take hundreds of processor cycles to fetch data from DRAM memory. Modern computer systems come with numerous levels of caches, prefetchers and other components to offset this gap in the speed of the processor and memory.

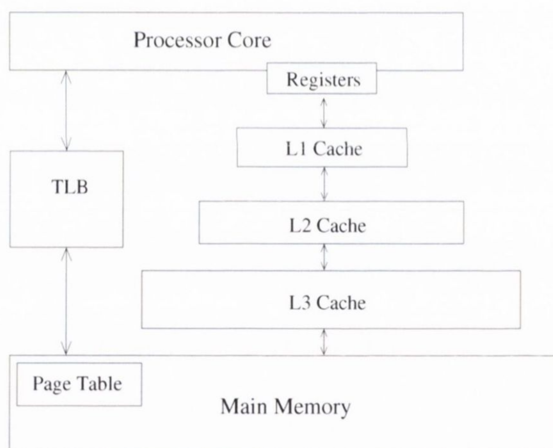


Figure 2.7: Memory Hierarchy

In order to produce efficient computer programs it is necessary to understand how these various components work, how they influence how programs operate and more importantly, how to exploit them [Goedecker 01, Drepper 07].

The memory hierarchy as shown in Figure 2.7 is important to understand

for our work on the In-Place Sparse Matrix Transpose. Caches are a very simple mechanism to conceal the gap in memory speeds. Caches are small amounts of fast memory which are close to the processor core. When the processor requests data from main DRAM memory, that data is brought into the processor. In fact, a whole *line* of data (this is often 64 bytes: 16 integers or eight doubles) is read in at once, incorporating the memory location requested and the data beside it in memory. This line of data is then stored in the cache. The theory is that if a piece of code had instructed the processor to load a particular piece of data (x), then it is probably likely to either (a) attempt to load that piece of data again soon (*Temporal Locality*) or (b) load another piece of data that is near x in memory (*Spatial Locality*).

Table 2.1: *Stoker:* Intel Xeon E7-4820 Cache Details

Cache	**Size**	**Associativity**	**Entries**
L1	32K	8-way	64 byte lines
L2	256K	8-way	64 byte lines
L3	18M	24-way	64 byte lines

The "line" of data will remain in the cache until it is replaced with another line of data from main memory. Caches have a *replacement policy* which decides when to replace lines of cached data. Addresses in main memory are mapped to locations in the cache with a certain level of *associativity*. In an *8-way* cache, when a new data line is read in, it replaces the oldest data line in the cache. Table 2.1 gives details of the sizes and associativity of the caches in our test system.

Table 2.2 gives details of the latency of cache misses in nanoseconds and processor cycles, at the different levels of processor cache. The latencies were determined using the Calibrator [Manegold , Yotov 05, Boncz 08] tool for Calibrating Cache Memory and TLB.

As we can see from Figure 2.7 and Tables 2.1 and 2.2 modern computers systems often now come with three levels of cache, L1, L2, and L3. The L1 cache is small (32K on *Stoker*, the system in Table 2.1) and is "close" to the processor, meaning that data from the L1 cache can be loaded into

Table 2.2: *Stoker:* Intel Xeon E7-4820 Cache Miss Latency from *Calibrator* tool

Level	Miss-Latency	Latency Cycles
L1	0.43 ns	1 cy
L2	5.26 ns	10 cy
L3	67.40 ns	134 cy

the processor in just a few cycles (one cycle according to *Calibrator*). The L2 cache is larger (256K on stoker) and slightly further away, taking a little longer to load data into the processor (ten cycles). The L3 cache is a lot larger (18M on stoker) and further away (134 cycles), though still nearer than main memory. The L3 cache is often shared between cores in a multi-core processor.

When the processor requests a piece of data that is in one of the caches it can load it relatively quickly depending on which level of cache it is in. If the data is not in the cache there is a *cache miss* and the data must be retrieved from a lower level cache, or main memory. This causes a *stall* in the processor as it waits for the data to become available, slowing down the program. Thus, cache hits are good and cache misses should be avoided where possible.

Another component is the *Translation Lookaside Buffer* (TLB). When a computer program is compiled, it hard codes the addresses in memory of various components of the program, the code, variables, statically allocated arrays, etc. These static hard coded addresses are known as *virtual addresses*. When the program runs, the processor creates a mapping between these virtual addresses and real addresses in memory that have been allocated to the program. This mapping is recorded in the *page table* which is stored in main memory. As main memory is slow, the processor uses a TLB, which is essentially a cache for the page table. If the processor attempts to look up a virtual address mapping in the TLB that is not cached, there is a TLB miss and the processor stalls (just like a cache miss) while waiting for the mapping to be retrieved from the page table in main memory. Thus TLB misses also influence the performance of

programs and algorithms. There has been considerable research in optimized matrix software which attempts to improve performance by reducing TLB misses [Goto 02, Goto 08b, Simecek 09]

Understanding how the Temporal and Spatial Locality of the caches and TLB operates aids us in developing algorithms which can be more cache efficient and can therefore perform better. Using Hardware Counters (see Section 5.9.1) we can get actual numbers of cache hits and misses, along with numerous other metrics, which allows us to analyse the performance of our applications and algorithms.

2.5 Complexity Analysis of Algorithms

When discussing and comparing algorithms it is very beneficial to use complexity analysis [Bruijn 70, Aho 74, Lewis 81, Greene 81, Sipser 96, Arora 09]. The Asymptotic Notation gives a general classification of the performance in (time/space) of an algorithm simply in terms of the size of its input (x).

In our discussion on algorithms we talk about the memory usage (space) and time complexity of the different algorithms. We do this because the memory usage and execution time of the algorithms are particularly dependent on the dimensions of the matrix, and in most cases increase and decrease depending on the relative dimensions of the matrix. The number of rows (n), the number of columns (m) and the number of non-zeros (nnz) in the matrix all influence the performance of the different algorithms.

In complexity theory we talk about the average or worst case *asymptotic* complexity of the algorithm. This means that for an algorithm with a particular input size (x) we omit all the various other factors involved and just talk about how the space (memory) or execution time of that algorithm grows proportionally to some function of the input size (x) as x increases to infinity. Because, as the input size grows larger and larger the size of x is the primary factor that influences the performance of the algorithm. It is then possible to discuss the worst case and average case performance depending on the input x.

Take a function $f(x) = 7x^2+4x+3$. As the size of the input (x) increases to infinity the x^2 term becomes the most dominant term in the equation. The coefficient, 7 and the other terms, $4x + 3$ have less and less influence on the relative size and become redundant. Asymptotic approximation is defined where we can state that there is some coefficient (a) of x^2 such that $a.x^2$ is always greater than or equal to $f(x) = 7x^2 + 4x + 3$. In this example, a coefficient of $a = 14$ would mean that $14.x^2$ is always greater than or equal to $7x^2 + 4x + 3$ for all positive inputs of $x \geq 1$. Thus we can say that $f(x)$ is asymptotically proportional to x^2.

There are a number of notations which we use to define complexity bounds on the algorithms. Depending on what we know about a particular function $f(x)$, we may say that the function has a best/average/worst case asymptotic complexity of; $\mathcal{O}(x^2)$, $\Omega(x^2)$, $\Theta(x^2)$ or $\sim(7x^2)$.

2.5.1 $\mathcal{O}(x)$ — Big-O: Upper Bound

Big-O Notation $(\mathcal{O})$ [Bachmann 23, Landau 24] is a very useful approximation that gives an upper bound on the complexity of an algorithm. It declares that an algorithm will never perform worse than this complexity.

We might say that a particular sorting algorithm has a complexity of $\mathcal{O}(x^2)$. This means that it will take no longer than time proportional to x^2. However Big-O does not define a lower bound on the complexity of the algorithm. The algorithm may complete in $\mathcal{O}(x \,.\, log(x))$ or even $\mathcal{O}(x)$ time, depending on the input. Big-O just gives an upper bound. Big-O is also not a tight bound, it is also true to say that this x^2 sorting algorithm is $\mathcal{O}(x^3)$ or even $\mathcal{O}(x^{100})$ as it will never perform worse than these bounds. In practice Big-O bounds are given as tight as possible (as tight as they can be proved).

2.5.2 $\Omega(x)$ — Big Omega: Lower Bound

Knuth popularized the Big-Omega Notation (Ω) [Knuth 76, Knuth 98] which provides a lower bound on an algorithm. It states that as an algorithm grows proportional to x it will always require *at least* this amount of

time/space proportional to x, never less. Ω does not put an upper bound on the complexity. Taking the previous sorting algorithm that was $\mathcal{O}(x^2)$ in the worst case. It may also be said that this same sorting algorithm is $\Omega(x \,.\, log(x))$ in all cases because it always performs at least $x \,.\, log(x)$ comparisons. Thus with $\mathcal{O}$ and Ω we can give upper and lower bounds to an algorithm.

2.5.3 $\Theta(x)$ — Big Theta: Double Bound

Big-Theta Notation (Θ) also popularized by Knuth [Knuth 76, Knuth 98] combines $\mathcal{O}$ and Ω to give both an upper and lower bound to the complexity of an algorithm. It states that a particular algorithm will always perform (operations/time/space) proportional to another simpler function, $g(x)$. Thus if we say an algorithm is $\Theta(x \,.\, log(x))$ it implies both $\mathcal{O}(x \,.\, log(x))$ and $\Omega(x \,.\, log(x))$. The algorithm will always perform proportional to $x \,.\, log(x)$, never x and never x^2.

This is a very tight bound on an algorithm which gives much more information about the algorithm. However it is not always possible to give a Big-Theta complexity. Some algorithms will vary in performance depending on the *content* of the input, not just the *size* of the input. A sorting algorithm may be $\mathcal{O}(x)$ in the best case if the input is already sorted. $\mathcal{O}(x \,.\, log(x))$ in the average case and may even be $\mathcal{O}(x^2)$ for particularly degenerate inputs. Thus, even though in each case the input size is the same, x, the algorithm takes different lengths of time due to the content of the input, so we can't have a Theta (Θ) complexity for this particular algorithm.

2.5.4 $\sim(x)$ — Tilde: Tighter Double Bound

Big-Theta notation (Θ) above gives a tight upper and lower bound on the asymptotic complexity of an algorithm, if we wish to compare algorithms which have the same asymptotic space/time complexity Big-Theta does not give enough information. This is because all the other components of the function are lost.

Tilde Notation ($\sim$) has been promoted by Sedgewick to address this [Sedgewick 11, Sedgewick 13]. Tilde Notation gives a tight upper and lower bound like Big-Theta, but gives an even tighter bound because the coefficients of the main component are retained. Thus a function $f(x) = 7x^2 + 4x + 3$ would have a Tilde Complexity of $\sim(7x^2)$. Comparing this to another function $g(x)$ which is $\sim(3x^2)$ we can say that $g(x)$ is more efficient. Theta notation would say that both algorithms are $\Theta(x^2)$ which hides this important distinction.

Summary

The four notations: Big-O ($\mathcal{O}$), Big Omega (Ω), Big Theta (Θ) and Tilde ($\sim$) are used throughout this document where appropriate. In general Theta Notation (Θ) is used where possible. In cases where algorithms have the same Theta complexity then Tilde ($\sim$) complexity may be used to distinguish between them.

When discussing the complexity of the Matrix algorithms there are two parameters which we use when discussing an $n \times n$ matrix with nnz non-zero elements. $\Theta(nnz)$ is used when discussing algorithms which depend on the *N*umber of *N*on-*Z*eros (nnz) in the matrix. $\Theta(n)$ is used when discussing algorithms which depend on the *N*umber of Rows (n). In the text $nrows$ is often used for clarity.

Chapter 3

Matrix Transpose

The matrix transpose operation is important in many areas such as Computational Chemistry [Rogers 03, Lewars 03, Sanders 08], Fast Fourier Transforms (FFT) [Cooley 65, Frigo 98, Lippert 98, Al Na'Mneh 05, Frigo 05], signal processing [Claasen 79, Padgett 09, El-Hadedy 10, Jie 10, Ravankar 11] and image processing [Portnoff 99, Baumstark 03, Na'mneh 06]

In the previous chapter we gave a general background on Linear Algebra and Dense and Sparse Matrices. In this chapter we give a detailed discussion on the Matrix Transpose operation itself, outlining the existing algorithms and research on this linear algebra operation. Section 3.2 gives an overview of the extensive research on the topic of Dense Matrix Transpose. Section 3.3 gives an overview of the Sparse Matrix Transpose. Particular focus is given to the two existing Sparse Transpose algorithms. The Out-of-Place algorithm is discussed in Section 3.4 and the Saad In-Place algorithm [Saad 94] in Section 3.5. Section 3.6 gives an overview of our experimental setup and explains our analysis methodology. Finally in Section 3.7 we evaluate the performance of the two existing Sparse algorithms in terms of memory usage and execution time.

3.1 The Matrix Transpose Operation

The Matrix Transpose [Cayley 59, Golub 96] is one of the basic linear algebra operations. The transpose of a matrix is defined as follows:

$$M_{ij}^{T} = M_{ji}$$

That is, the row and column indexes are exchanged. The element at row i, column j in the transpose M^T is the element at row j, column i in M. Given a matrix M (Example 3.1(a)), the transpose M^T (Example 3.1(b))

of that matrix can be constructed by swapping the elements of all columns in the matrix with the elements in the corresponding rows (and vice versa). The procedure can be seen in Example 3.1 where column 1 is swapped with row 1 and row 4 is swapped with column 4, etc.

$$M = \begin{pmatrix} \mathrm{a} & 0 & 0 & 0 & b & 0 \\ c & \mathrm{d} & 0 & 0 & 0 & e \\ 0 & f & \mathrm{g} & 0 & 0 & 0 \\ h & 0 & 0 & \mathrm{i} & j & 0 \\ 0 & 0 & 0 & 0 & \mathrm{k} & l \\ 0 & m & 0 & 0 & n & \mathrm{o} \end{pmatrix}$$

(a)

$$M^T = \begin{pmatrix} \mathrm{a} & c & 0 & h & 0 & 0 \\ 0 & \mathrm{d} & f & 0 & 0 & m \\ 0 & 0 & \mathrm{g} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathrm{i} & 0 & 0 \\ b & 0 & 0 & j & \mathrm{k} & n \\ 0 & e & 0 & 0 & l & \mathrm{o} \end{pmatrix}$$

(b)

Example 3.1: Sample Matrices M and its Transpose M^T

Alternatively we may describe the transpose as flipping the elements in the matrix across the top left to bottom right diagonal.

Repeating the transpose operation on the transpose of a matrix results in the original matrix:

$$(A^T)^T = A$$

A Symmetric Matrix is a matrix where the transpose of a matrix is identical to the original matrix.

$$\textit{Symmetric}: \; A = A^T$$

The transpose operation regularly occurs in linear algebra equations where the transpose of a matrix or vector is required for some calculation.

The original reference BLAS implementation does not actually include a procedure to perform the transpose operation. Logically there is no reason to need one, the procedures can simply access the matrix in a transposed order by swapping the row and column indexes. As such the procedures take arguments which tell the routine whether the matrix is stored in row-major or column-major ordering (see Section 2.3), the length of this

major dimension, and whether the matrix should be accessed in transposed form.

Such an approach is acceptable from a mathematical and theoretical perspective however as we saw in Section 2.4, memory access patterns can have a huge influence on how well an algorithm re-uses data in the caches and thus the performance of the algorithms. If a matrix is going to be accessed many times by a particular ordering then there could be a significant performance benefit from transposing the matrix so that it is stored in that ordering. It is for this reason that many optimized linear algebra and mathematical libraries which include implementations of the BLAS also include extra optimized transpose procedures. For example, the Intel Math Kernel Library (MKL) [Intel 93] comes with three dense transpose routines: `mkl_imatcopy()` for in-place transpose, `mkl_omatcopy()` for out-of-place transpose and `mkl_omatcopy2()` for out-of-place transpose with double-stride.

3.2 Dense Matrix Transpose

This section gives an overview of some of the extensive research into Dense Matrix Transpose. Although somewhat different to the problem of sparse matrix transpose, the work listed here demonstrates the importance of efficient matrix transpose algorithms, be they for dense or sparse matrices.

The Out-of-place transpose of a matrix stored in a Dense format (Section 2.3) is straightforward. Simply iterate through the matrix by rows (or columns depending on the major ordering) and copy each element in that row to the location of the corresponding column in memory. The copying can be performed by blocks in order to improve the efficiency of the transpose [Lam 91, Navarro 96, Gustavson 98, Kågström 06, Elmroth 04, Gustavson 12].

ALGORITHM 3.1: Dense Square Transpose

```
1 for ( 0 ≤ i < N ) do
2 |  for ( i+1 ≤ j < N ) do
3 |  |  tmp ← A[i][j];
4 |  |  A[i][j] ← A[j][i];
5 |  |  A[j][i] ← tmp;
6 |  end
7 end
```

3.2.1 *In-Place* Dense Matrix Transpose

In many situations we have a very large dense matrix stored in memory (or in external storage) and there is insufficient additional memory to hold a full copy of the matrix. The matrix then needs to be transposed in place using as little additional memory as possible. The problem of in-place transpose of a square or rectangular matrix has received considerable attention. The related problem of general in-place permutation has also received considerable attention [Durstenfeld 64, Floyd 72, Duijvestijn 72, Fraser 76, Melville 79, Feijen 87, Baker 92, Fich 95, Choi 95, Keller 02].

The In-Place transpose of a square matrix stored in a Dense format is logically straightforward, simply iterate through the upper triangular part of each row and swap the element at each location $A[i][j]$ with the element in the opposite column $A[j][i]$ by swapping the row and column indexes. Algorithm 3.1 shows the basic in-place transpose of a dense square matrix.

Although the algorithm iterates sequentially through the rows of the matrix where the rows are adjacent, the column access are not sequential and are dispersed throughout the matrix arrays causing the algorithm to access memory at strides of size n. This can cause considerable problems with cache performance [Gatlin 99].

Cache performance can be improved by transposing the dense square matrix in blocks. Figure 3.1 shows a simple example of the in-place block transpose of a dense matrix. The yellow blocks along the diagonal are transposed in place. The benefit of the block transpose can be seen when transposing the blue and green blocks. Take the green blocks for example. When reading the first block, 51 and 52 with be adjacent in memory, as

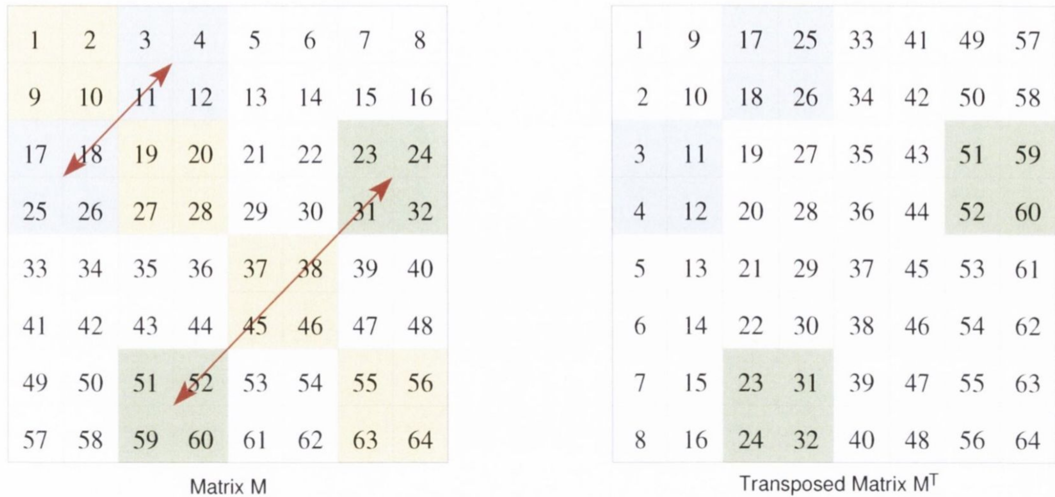


Figure 3.1: Block Transpose of Dense Matrix

will 59 and 60, so they should be read in together in a single cache line. The advantage with the block transpose is that in the destination block 23 and 24, and 31 and 32 will also be adjacent in memory, thus improving on the cache efficiency.

Chatterjee and Sen investigate the performance of six different algorithms for in-place transpose [Chatterjee 00]. They compare (1) the basic row major implementation (similar to Algorithm: 3.1) with (2) an in-place transpose designed to be efficient in terms of the basic I/O memory model of Aggarwal and Vitter [Aggarwal 87, Aggarwal 88]. Two further algorithms (3) and (4) are designed to be efficient in terms of their cache I/O model [Sen 02, Gatlin 99]. i.e. they transpose the matrix in blocks in a way that that is designed to give good cache performance. Algorithm (5) is a Cache-Oblivious algorithm from [Frigo 99]. The sixth algorithm (6) requires that the matrix is stored in their hierarchical matrix layout which uses sub-blocks arranged in a Morton order [Morton 66, Wise 01] layout (Section 2.3.3).

The results showed that the matrix arranged in their Morton ordered layout gave the best performance at about six times faster than the standard row-ordered transpose. This algorithm requires the user to modify the

layout of their matrix which may not be practical. More interesting from their results was that the two cache optimized algorithms are roughly four times faster than the naïve algorithm while still using the standard row-major matrix layout.

3.2.2 *In Place* Dense Rectangular Transpose

The In-place transpose of a Dense Rectangular matrix is more complicated. We cannot simply exchange elements by swapping their row and column indices as with a square dense matrix, because in a rectangular matrix the rows and columns are of different lengths. This means that a particular row will be at a different location within the array after the transpose. The position that each element needs to be moved to is still known. Given a matrix of size $(n \times m)$, element a_{ij} is at $A[im + j]$ in the original matrix and is moved to position $A[i + jn]$ in the transpose which is the location of a completely different element in another row and column. As we move elements during the transpose procedure we move elements in a *Cycle-Chasing* fashion. Elements are moved in this cycle-chasing chain until we eventually reach an element that should be moved to the original location a_{ij} where the chain started.

Cycle-Chasing In-Place Transpose

The in-place cycle-chasing algorithm was first described by [Berman 58] however there is a reference in [Pall 60] to a similar algorithm by Shooman from 1957. Using the two equations for the location x of an element in the matrix $(iM + j)$ and the location x' of the element in the transpose $(i + jn)$, we can define the permutation function $x' = \pi(x)$ such that:

$$
\begin{aligned}
&\textit{Permutation Function; Transpose of } x: \\
&x' = \pi(x) \qquad (3.1) \\
&\pi(im + j) \rightarrow i + jn
\end{aligned}
$$

Thus, for any particular element, 'x', we can define a simple for-

mula [Berman 58, Cate 77b] for the permutation of that element as:

$$\pi(x) = \begin{cases} nx \mod mn - 1 & \text{if } x \neq mn - 1, \\ mn - 1 & \text{if } x = mn - 1 \end{cases} \tag{3.2}$$

The next location in the chain is found by multiplying the current location x by n and taking the modulus of $(nm - 1)$ – the index of the last element in the array. Similarly the inverse may be defined as:

$$\pi^{-1}(x') = \begin{cases} mx' \mod mn - 1 & \text{if } x' \neq mn - 1, \\ mn - 1 & \text{if } x' = mn - 1 \end{cases} \tag{3.3}$$

[Berman 58] outlines an algorithm using these relations to transpose a rectangular matrix in place. The algorithm requires a flag for each element to record if it has been moved yet. Berman suggests using either the low order bit of the floating point value of the matrix elements (if exact precision is not required) or an extra work array of $\mathcal{O}(n \times m)$ bits.

[Windley 59] presents an algorithm by J.C. Gower which removes the $\mathcal{O}(n \times m)$ memory overhead at the cost of additional computation. Given that any permutation of a number of elements can be represented by a set of mutually exclusive cycles, meaning that each element will only be moved by a single cycle. The algorithm scans through the matrix and for each element at position x it calculates all the addresses in the cycle containing x. If any of those addresses is less than x, then that element has already been moved and can be skipped. Otherwise the algorithm starts an element moving cycle from x. A count of the number of elements moved can be used to detect when the algorithm has moved all elements.

[Windley 59] presents another algorithm which also does not require the $\mathcal{O}(n \times m)$ memory overhead of the [Berman 58] algorithm. This algorithm reduces the computational overhead of the Gower algorithm at a cost of an increase in reads and writes to memory. The algorithm calculates addresses in the cycle in reverse and moves elements such that they are in the correct order relative to the remaining unmoved elements. CACM Algorithm: 302 [Boothroyd 67] gives an implementation of an in-place transpose algorithm based on [Windley 59].

Pall and Seiden outline a method [Pall 60] using Abelian Groups of pre-calculating on paper the *cycle leaders* of different sized matrices before proceeding to transpose the matrix. A Cycle leader is the lowest addressed element in each cycle and thus will be the first element of each cycle encountered by the algorithm. They also give a procedure for calculating cycle lengths and demonstrate that the problem of calculating cycle leaders decomposes into one sub-problem for every divisor d of $(mn - 1)$. Their experiments show that this method gives much better performance than the algorithm of Shooman (which appears to be similar to [Berman 58]).

Laflin and Brebner present CACM Algorithm: 380 [Laflin 70b] which is an improvement over Algorithm: 302 [Windley 59, Boothroyd 67]. In this algorithm they exploit *dual cycles*, that is, the cycles starting at position x and at position $(nm - 1 - x)$. If x is the smallest value of a cycle loop then $(nm - 1 - x)$ is the largest value of a loop. The dual cycles thus can be shifted simultaneously to improve efficiency. If both values belong to the same cycle then this can be detected and handled efficiently. Algorithm: 380 [Laflin 70b] also uses an additional work array of size $|\frac{1}{2}(m + n)|$ to record which cycles have already been moved in order to improve efficiency.

Brenner presents CACM Algorithm: 467 [Brenner 73] which further improves on the previous in-place algorithms. Brenner proves a number of theorems using number theoretical analysis of the properties of the cycles in the in-place transpose. Brenner then uses those properties and a method similar to that in [Pall 60] to predict the location of cycle leaders to produce an algorithm which improves on Algorithm: 380 [Laflin 70b]. Results show that when using a work array of size $(^{n+m}/_{2})$, Algorithm: 467 is faster than Algorithm: 380 and Algorithm: 302. Considerably so when $(nm - 1)$ has many factors (hence many subcycles).

Cate and Twig give an in-depth analysis of the in-place cycle-chasing transpose of non-square matrices [Cate 77b] and present CACM Algorithm: 513 which improves on the performance of the previous algorithms using the numerical analysis properties they outline. Theorems from previous papers are reviewed and some new theorems presented. They give the equations of the permutation $\pi()$ and its inverse $\pi^{-1}()$ as shown above in

Equations 3.2 and 3.3. They also show the following:

- The longest cycle is the cycle containing $x = 1$ and has length L.
- The lengths of all other cycles are divisors of L.
- The number of cycles of a particular length can be calculated.
- Elements 0 and $mn - 1$ are fixed points (not moved under transposition), if m and n are odd then the midpoint $\frac{mn-1}{2}$ is also a fixed point.
- Two formulas are provided for calculating the number of fixed points in a particular transpose permutation.
- If there is a cycle at address x then there is also a cycle at address $nm - 1 - x$.
- In some cases the two cycles at x and $nm - 1 - x$ coincide and are part of the same cycle. In this case the length of the cycle is even and the addresses x and $-x$ are separated by half a cycle.

CACM Algorithm: 513 [Cate 77b] uses this cycle symmetry and the calculation of fixed points to improve on the performance over the previous algorithms 302 and 380. Results showed that the revised algorithm demonstrated a performance improvement between 25% and 35% over the previous algorithms. Leathers however provides a further analysis [Leathers 79] which shows that the earlier Algorithm: 467 [Brenner 73] performs better than Algorithm: 513 [Cate 77b] in most cases except where the modulus $(nm - 1)$ of the matrix is prime in which case Algorithm: 513 has a slight advantage.

Analysis of in-place transpose permutation [Knuth 71, Fich 95] shows that if the inverse of a permutation is known (as is the case with the in-place transpose) then it can be shown that the worst case running time of the permutation is $\mathcal{O}(n \,.\, log(n))$.

Other Approaches

[Dow 95] describes and evaluates 5 different algorithms for dense matrix transpose of rectangular matrices with particular focus on algorithms which are efficient on vector computers. The first algorithm, `V1` is the basic out-of-place algorithm as in Section 3.2 which copies every element sequentially to its correct location in a new array of size $m \times n$ and then copies the elements back to the original array in their new order. As discussed above, this out-of-place `V1` algorithm is generally quite fast however it has a significant memory overhead of $\Theta(m \times n)$.

Algorithms `V2` and `V3` modify the shape of the rectangular matrices so that they are square and then use efficient in-place blocked square transpose algorithms to transpose the matrix.

Algorithm `V2` is the *Pad Transpose*, extra space is added to the rows or columns (whichever is shorter) in order to make the matrix square. The pad method can only be used if there is sufficient additional space at the end of the array. $(max(n,m)^2 - nm)$ additional memory locations are required. In row-major format, if there are more rows than columns then each row needs to be padded. This is done by iterating in reverse through the array and shifting elements of each row towards the end of the array a number of places equal to the row number times the difference in row and column lengths. Row zero is not moved, row 1 is moved $(1 \times |m-n|)$, row 2 is moved $(2 \times |m-n|)$, row 3 is moved $(3 \times |m-n|)$, etc. The matrix is then transposed in place using an efficient blocked square transpose algorithm [Alltop 75, Ramapriyan 75, Buttari 07, Bikshandi 06]. After the transpose all the additional columns have become additional rows at the end of the matrix and can be ignored. If there are more columns than rows in row-major format then the above steps can be reversed. First transpose the square $(max(n,m) \times max(n,m))$ matrix which assumes that there are new padded rows at the end, then left shift all the elements of each row towards the front of the array leaving the padding at the end. The pad algorithm is particularly efficient when m and n are of similar magnitude, in this case the memory usage is much less than the out-of-place algorithm.

The pad method does require the additional memory to be directly at the end of the array which can make it impractical.

Algorithm V3 is the *Cut Transpose*, additional rows or columns are cut from the matrix to leave a square matrix which can be efficiently transposed in place [Alltop 75, Ramapriyan 75]. As such, the cut transpose is the opposite of the pad transpose. In row-major format, if there are more rows than columns ($m > n$), the additional rows are copied to the extra workspace area. The remaining ($n \times n$) square matrix is transposed in place with an efficient blocked algorithm then the square matrix is "padded" with extra columns as above by shifting elements in rows towards the end, expanding the matrix to the full ($m \times n$) size. Finally the extra rows are copied back from the workspace to their appropriate column. If there are more columns than rows ($m < n$) then the elements from the additional columns are copied to workspace and the columns are cut by shifting rows towards the top of the array. The square $m \times m$ matrix is transposed and the additional columns are copied from the workspace to their appropriate rows at the end of the array. The cut method requires ($|m-n| \times min(m, n)$) additional workspace memory. This additional memory is always less than that required by the out-of-place algorithm. The memory overhead is considerably less when m and n are similar in magnitude and the algorithm is also particularly efficient in this case. A major advantage of the *cut* method over the *pad* method is that the additional workspace memory does not need to be at the end of the matrix.

Algorithms V4 and V5 are rectangular block transpose algorithms [Eklundh 72, Alltop 75, Ramapriyan 75, Van Voorhis 77, Hegland 96] which can be used when m and n are composite (have a divisor d greater than 1 which is not prime). m and n can be made composite using the cut and pad techniques if required. Both algorithms partition the matrix into blocks and then transpose the blocks as a whole using the cycle-chasing method above [Berman 58, Windley 59]. These blocks can be efficiently transposed in this way on a vector computer. Algorithm V4 partitions the major dimension of the matrix such that there are m rows and d columns of size p where $dp = n$. After the sub-rows are transposed in cycles the elements

are reordered between sub-rows to their correct position. Algorithm `V5` partitions the matrix into blocks of size $d \times d$ where $dp = n$ and $dq = m$. The elements of each block are first transposed in place then the blocks are transposed as a whole. Diagonal blocks remain in-place.

Finally Dow compares these five algorithms to two implementations of the scalar cycle-chasing algorithms above. CACM Algorithm 467 [Brenner 73] and NAG F01CRF from the Numerical Algorithms Group Fortran Library [NAG 93]. The results of these experiments show that the five vector efficient algorithms outlined are at least an order of magnitude faster than the scalar cycle-chasing algorithms. For the matrices used in these experiments on the vector computer the out-of-place algorithm `V1` was generally the fastest with the cut method `V3` occasionally out performing it. The pad `V2` and cut `V3` algorithms perform well at a similar speed for matrices which are close to square. The pad algorithm does not perform well if a large amount of padding needs to be added. The blocked algorithms `V4` and `V5` do not perform as well as `V3` in these vector experiments. Algorithm `V5` performs better than `V4` as larger amounts of data are moved together with the larger blocks.

Cache Oblivious Dense Matrix Transpose Algorithms

In recent years *Cache Oblivious Algorithms* [Frigo 99, Tsifakis 04, Bader 07, Yzelman 11] have become popular as they can give improved performance without having to be tuned for a particular architecture or memory hierarchy and cache sizes. Cache oblivious algorithms recursively partition the problem into blocks. At each level of recursion they divide the problem into smaller and smaller blocks. After a number of recursions the algorithm will have blocks which are small enough to fit in all the different levels of cache. Thus the cache oblivious algorithms do not need to be tuned for particular architecture or cache sizes. Experimental analysis [Chatterjee 00, Yotov 07] has shown that these cache oblivious transpose algorithms do improve on the performance of the naïve row or column based algorithms, however they still fall short of the performance of tuned cache-aware blocked algorithms.

Out-of-Core Dense Matrix Transpose

The ability to transpose a matrix in-place is important when the matrix is large and there is insufficient extra storage available to store a full copy of the matrix. In some cases a matrix may be so large that it does not all fit in main memory, in this case the matrix must be transposed *out-of-core* where the matrix is stored in external storage (hard disk / tape) and only small parts of the matrix can be held in memory at any one time. Transposing a square matrix out-of-core is simpler than a rectangular matrix as row and column locations are invariant however, it is more difficult than transposing a square matrix in memory. The out-of-core transpose is a complicated problem which has received much attention.

Eklundh presents an efficient out-of-core algorithm [Eklundh 72, Eklundh 73] for transposing large square matrices of size $2^n \times 2^n$. The algorithm assumes that the entire $2^n \times 2^n$ is stored in external storage. It requires an additional in-memory working area of at least 2^{n+1} in order to store at least two rows of the matrix in memory at a time. The algorithm reads in two pairs of rows from the matrix at a time, swaps certain elements between the rows and writes the rows back to external storage. The algorithm reads different pairs of rows over multiple passes continuing to swap elements until all elements have been moved to their correct transposed location. Thus the algorithm can transpose an out-of-core matrix in n passes or fewer. If there is additional in-core memory available the algorithm can processes multiple (2^j) rows at a time to improve efficiency. A similar algorithm was also presented by [Schumann 72, Schumann 73] using sequential access devices compared to the direct (random) access devices of [Eklundh 72].

Delcaro outlines a method [Delcaro 74] based on the Eklundh algorithm for transposing large square and non-square matrices in external storage. The algorithm requires the row and column dimensions m and n to have a large number of factors. The matrix is partitioned into blocks based on these factors and transposed. Twogood also extends the Eklundh algorithm to the general case [Twogood 76] where $2^i (j \geq 1)$ of its rows will fit into main memory and analyses its performance for two-dimensional

image filtering. Alltop presents a three step algorithm [Alltop 75] which is another extension to that of Eklundh to support the transpose in-core and out-of-core of large square and rectangular matrices by augmenting the matrix such that its dimensions have a large common divisor d. The matrix is padded in both directions to increase the size of d and the matrix is then partitioned similar to Delcaro into a square $d \times d$ matrix of blocks of size $\frac{n}{d} \times \frac{m}{d}$ which are then transposed. The Alltop algorithm requires additional storage of $(2nm/d)$. Ramapriyan presents a generalization [Ramapriyan 75] of Eklundh's algorithm which can transpose out-of-core matrices which are not square powers of two and which are also non-square $(m \times n)$. Van Voorhis presents a further generalization [Van Voorhis 77] of the Alltop algorithm which removes the requirement of a factor of two for the matrix dimensions in external storage and also combines the last two steps of the three step algorithm.

Ari describes two improvements [Ari 79] to Eklundh's algorithm. The first reduces the number of accesses to external storage at a cost of an increase in the amount of data transferred. The second shows how the efficiency of the algorithm can be improved by using a small amount of extra external storage if available. Goldbogen presents PRIM [Goldbogen 81] another in-place out-of-core transpose algorithm which can transposes an $n \times m$ matrix in a series of iterative transformations of the entire matrix. Unlike the algorithm of Eklundh which permutes single elements, Goldbogen also permutes blocks of elements. Twigg describes an algorithm [Twigg 83] for transposing large matrices stored in external files. The algorithm is based on sort-merge using a variant of the balanced tape merge algorithm [Lorin 75] to transpose the matrix by transposing the matrix in chunks into intermediate files which are then merged together.

Kaushik et al. give a review [Kaushik 93] of a number of the out-of-core in-place matrix transpose algorithms based on Eklundh's algorithm [Eklundh 72] and propose another variation based on tensor products [Fraser 76, Johnson 92, Johnson 93] which improves efficiency by reducing the number of disk accesses required by the algorithm. Results show that the new single radix algorithm considerably reduces both disk

I/O time and computation time and thus total execution time over a range of matrix sizes compared to that of Eklundh. The single radix algorithm reads pairs of contiguous rows whereas Eklundh reads non-continuous rows which results in a greater number of disk accesses.

Suh presents an improvement of the out-of-core algorithm [Suh 02] which improves performance over previous algorithms by reducing the number of I/O operations and eliminating the index computation. I/O is reduced by writing the data onto disk in predefined patterns and balancing the number of disk read and write operations. The index computation time, an expensive operation involving two divisions and a multiplication, is eliminated by partitioning the memory into read and write buffers. Krishnamoorthy also presents an algorithm [Krishnamoorthy 04] which improves performance of the out-of-core transpose by minimising the number of I/O operations. This is done by using the I/O characteristics of the system to determine optimal block sizes for read, write and communication such that the total execution time is minimised.

Parallel Dense Transpose

The parallel transpose is another variant of the in-place dense transpose which has received a lot of attention, particularly for parallel applications such as FFT [Cooley 65, Lippert 98, Jie 10, Al Na'Mneh 05] where the dense matrix is partitioned and different sections of the matrix are distributed across multiple processors/nodes.

Choi describes a parallel transpose algorithm [Choi 95] for use in the PUMMA library for the parallel multiplication of transposed matrices which are distributed across numerous processors. The algorithm uses non-blocking message passing to transfer matrix blocks which are arranged in a cyclic data distribution. Hegland introduces a new parallel transpose split algorithm [Hegland 96, Calvin 96] which can be used for parallel matrix transpose as part of FFTs on the Fujitsu VPP 500 vector computer. The algorithm achieves a third of peak performance using 32 processors. Wapperom presents a further improvement to the split transpose

method [Wapperom 06] for three dimensional Fourier Transforms. Data is split along two dimensions to allow for a higher degree of parallelism, the algorithm also modifies the all-to-all communication to be performed in groups.

Lippert presents a parallel transpose algorithm [Lippert 98] targeted at SIMD (Single Instruction Multiple Data) systems, in particular those with the high speed APE/Quadrics interconnect. The algorithm, which is also intended for use with Fast Fourier Transforms shows improved performance on interconnected systems with rigid next-neighbour connectivity and lack of local addressing. He and Ding investigate the performance [He 02] of in-place multi-dimensional array transposition with the vacancy tracking algorithm while using OpenMP, MPI and hybrid MPI/OpenMP for communication. On a single node OpenMP outperforms MPI and across a cluster the hybrid MPI/OpenMP outperforms MPI. Al Na'Mneh presents an adaptive matrix-transpose algorithm [Al Na'Mneh 05] for transposing matrices, which is based on all-to-all communication on symmetric multiprocessors. The algorithm reduced overhead by adaptively choosing a suitable radix based on a number of factors. Experimental results show the transpose algorithm gives increased performance for six-step One-Dimensional Fast Fourier Transforms. Ravankar presents another algorithm [Ravankar 11] for parallel matrix transpose on a Torus Array Processor which has a time complexity of $\mathcal{O}(n)$. The algorithm uses the matrix-matrix multiply-add (MMA) operation for transposing the matrix which is carried out in $5n$ time-steps.

3-Dimensional Matrix Transpose

3D matrices occur in many problem domains such as seismic and medical imaging. If we think of a 2D matrix as a square then a 3D matrix can be thought of as a cuboid. The 3D Matrix transpose operation changes the axis order of the cuboid. It may simply swap two of the axes or it may rotate all three axes depending on the requirement of the operation. Figure 3.2 shows a three axis rotation from XYZ to YZX. The transpose

continues to swap row/column elements according to the axis rotation, thus in Figure 3.2 the element at location (i, j, k) would be moved to location (j, k, i) and that element would be moved to location (k, i, j), etc.

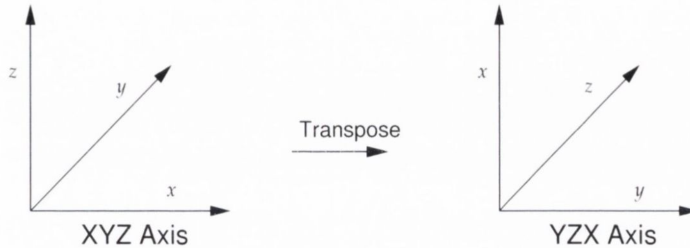


Figure 3.2: 3D Transpose: Rotate XYZ axis to YZX

Wapperom presents a variation of the split transpose method [Wapperom 06] for three dimensional Fourier Transforms. Data is split along two dimensions to allow for a higher degree of parallelism, the algorithm also modifies the all-to-all communication to be performed in groups. El-Moursy presents an algorithm [El-Moursy 08] for parallel transposition of 3-Dimensional matrices on multicore architectures. The algorithm exploits the software managed memory hierarchy of SIMD architectures such as the Cell Broadband Engine.

3.3 Sparse Matrix Transpose

In the previous section we gave an overview of the considerable research into the problem of dense matrix transpose. In the following sections we give an overview of the research into the problem of sparse matrix transpose [Pissanetzky 84]. We also give a detailed description of the existing out-of-place and in-place transpose algorithms and experimentally analyse the performance of the algorithms.

Sparse matrix transpose is the procedure of transposing a matrix which, due to the high proportion of zeros in the matrix is stored in one of the compact storage formats outlined in Section 2.3 such as Compressed Sparse Row. The procedure for transposing a matrix stored in row-major

format is also identical to the procedure for converting the matrix to the column-major Compressed Sparse Column format.

When a matrix is stored in a compact format we do not know the exact location in memory of every element in the matrix. This makes it more difficult to produce cache efficient sparse matrix algorithms.

As with dense matrices there are two main ways of transposing a sparse matrix. The most straightforward method is the out-of-place technique described in Section 3.4. The out-of-place method creates an entirely new empty matrix in memory, then each element is copied to its correct transposed location in this new matrix. The second method is the in-place technique described in Section 3.5 which reduces the memory overhead of the algorithm by transposing the matrix in place. Section 3.6 gives an overview of our experimental setup then Section 3.7 shows the results of the experimental analysis of these two existing algorithms.

3.4 *Out-of-Place* (OOP) Sparse Transpose

As with dense matrices, the straightforward method to transpose a sparse matrix is to copy the elements to their transposed location in a second separate set of matrix arrays. The out-of-place sparse transpose algorithm can be loosely compared to an out-of-place bucket sort algorithm. It is a simple fast algorithm, however the memory overhead of the algorithm is extremely large as it doubles the memory required for the matrix. For the largest matrix in our sample collection the OOP algorithm requires a memory overhead of 4,699 MiB resulting in a total memory usage of at least 9,398 MiB when performing the transpose.

This *Out-of-Place* method appears to be the most commonly used sparse matrix transpose algorithm. Variations of the out-of-place algorithm described in Section 3.4.3 are implemented in numerous packages. For example, the `Sparskit2` [Saad 94] package contains two Fortran implementations of the sparse matrix transpose. The `CSRCSC()` subroutine for converting from CSC to CSR format (which is the same as transpose) implements the OOP algorithm similar to Algorithm 3.2. The Bebop

Sparse Matrix Converter [Demmel 05, Vuduc 05] is a library for converting sparse matrices between different storage and file formats which includes an out-of-place transpose routine. The `CHOLMOD` [Chen 08] package which comes as part of Tim Davis's SuiteSparse [Davis 05b] collection of sparse matrix packages, includes a number of `cholmod_transpose()` procedures which implement the out-of-place algorithm for transposing and permuting different types of matrix (real, double, complex, integer, pattern, symmetric, unsymmetric, etc.). The `HSL` [Group 63, Gould 04] Mathematical Software Library from the Numerical Analysis Group is a closed source, commercial, collection of FORTRAN [Backus 56, Backus 57] codes for large scale scientific computation. HSL includes two routines for performing sparse matrix transpose. According to the library documentation, transpose algorithm `MC38()` requires three output arrays to be allocated. A double array of size nnz, integer array of size nnz and an integer array of size $(m + 1)$ - this would indicate that `MC38()` uses the out-of-place algorithm.

There are far fewer research publications focused on the topic of sparse transpose than dense transpose. One of the first descriptions of the out-of-place sparse matrix transpose is from McNamee who presents `TRSPMX()` as part of CACM Algorithm: 406 [McNamee 71] which is a collection of linear algebra routines for sparse matrices. Further remarks [Sipala 77, Gustavson 78a] correct some initial errors. Gustavson presents `HALFPERM()` which is a variation on the out-of-place transpose which can be used twice in order to perform the full sparse matrix permutation PAQ^{-1} on the matrix A. The algorithm presented is described as being similar to a distribution count sort. Experimental results using HALFPERM for permutation show that it performs up to ten times faster than TRSPMX.

3.4.1 Parallel Sparse Matrix Transpose

When dealing with very large linear algebra problems the matrix can be partitioned across multiple distributed memory nodes in order to solve the problem in parallel. If a transpose operation is required as part of the calculation, such as matrix-transpose-vector multiplication ($A^T v$), it is

possible to perform the calculation without transposing by accessing the elements through transposed indexing. However it may be more efficient to perform the calculation if there was a quick and simple method to transpose the distributed matrix. Thus a distributed parallel transpose sparse matrix transpose is an interesting problem

Hendrickson investigates the problem of partitioning sparse unsymmetric and rectangular matrices to balance work between nodes and keep communication costs low [Hendrickson 98]. Results show that multilevel partitioning methods give the best performance.

Kruskal investigates techniques for the parallel manipulation of sparse matrices [Kruskal 89], a number of algorithms are considered including the parallel transpose of sparse matrices. The algorithms are considered from the perspective of a shared memory MIMD (Multiple Instruction Multiple Data) system. To transpose a matrix in row major format (in this case a variation of the CSR format Section 2.3.5), it is converted to a canonical format where the matrix is stored in triplets similar to COO (Section 2.3.4). The matrix in canonical format is transposed by swapping row and column indexes in each element, the elements are reordered by index using a radix sort, the numbers of elements in each row computed and the matrix converted back to row major CSR variant.

Buluc again looks at the matrix-transpose-vector multiplication problem [Buluç 09]. The Compressed Sparse Blocks (CSB) storage format is introduced. Storing the sparse matrix in the CSB format allows both Ax and $A^T x$ to be computed efficiently. This blocked CSB storage format also allows the sparse matrix A to be efficiently permuted out-of-place into the transpose A^T.

Gonzalez introduces a parallel out-of-place sparse matrix transpose algorithm [Gonzalez-Mesa 13] which uses *Transactional Memory* [Herlihy 93] which supports atomic group load and store instructions thus easing parallelisation and ensuring correctness. Results showed that the parallel algorithm using transactional memory exhibits improved performance over the baseline.

3.4.2 Sparse Matrix Transpose Unit

Stathis et. al. describe the *Sparse Matrix Transpose Unit* [Stathis 04] which is a proposed hardware co-processor for vector computers. The unit is designed for efficient transposition of sparse matrices which are stored in the hierarchical sparse matrix (HiSM) storage format [Stathis 03a] which is similar to the Recursive Sparse Blocks format (Section 2.3.8). The unit has internal register memory of size $s \times s$, thus the matrix is stored in hierarchical HiSM blocks of size $s \times s$. The performance of the unit was evaluated using their D-SAB Sparse Matrix Benchmark Suite [Stathis 03b]. Results show that the transpose unit exhibits speedups of up to 32 times compared to those of the standard compressed row storage format with an average speedup of 17 times.

As we have seen before, it is more efficient to transpose sparse matrices stored in hierarchical or blocked formats. With a hierarchical format and specialised hardware, the sparse matrix transpose unit is highly efficient. However such a blocked hierarchical transpose is only useful for matrices already stored in such a format as converting a matrix stored in a compressed sparse column or row format to the HiSM format is expensive.

3.4.3 Description of Out-of-Place Transpose Algorithm

In this section we describe the out-of-place sparse matrix transpose algorithm and analyse how it operates. Section 3.7 presents results of runtime performance analysis of the algorithm.

Sparse Transpose Input Matrix M in CSR Format

We first describe the structure of the sparse matrix M stored in the CSR format as shown in Data Structure 3.1. This structure is the standard input to all the sparse matrix transpose algorithms presented in this document. The structure contains three integers $(nrows, ncols, nnz)$ which give the dimensions of the matrix, one real array $(non_zeros[\,])$ which contains

DATA STRUCTURE 3.1: Matrix M in CSR format:
This is the Input and Output for all transpose algorithms

Input: Matrix M in the CSR representation - Containing:
- *nrows* - the number of rows in the matrix M
- *ncols* - the number of columns in the matrix M
- *nnz* - the number of non-zero values in the matrix M
- *row_ptrs*[ ] - array of row pointers in M [nrows+1]
- *non_zeros*[ ] - array of element values in M [nnz]
- *col_indexes*[ ] - array of element column indexes in M [nnz]

the values of the matrix elements and two integer arrays (*row_ptrs*[ ] and *col_indexes*[ ]) which define the layout/structure of the matrix. Example 3.2 shows a representation of the matrix M stored in CSR format.

nrows	=	6															
ncols	=	6															
nnz	=	15															
row_ptrs	=	0_0		2_1			5_2		7_3			10_4		12_5			15
non_zeros	=	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
col_indexes	=	0	4	0	1	5	1	2	0	3	4	4	5	1	4	5	
index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Example 3.2: Matrix M in CSR representation

Out-of-Place Sparse Matrix Transpose Algorithm

Algorithm 3.2, gives a basic pseudo-code implementation of the out-of-place transpose. The algorithm takes as input the 'C' structure M representing the CSR matrix as outlined in Data Structure 3.1 and will have contents similar to that shown in Example 3.2. The output of the OOP algorithm is a completely new matrix structure containing M^T as shown in Example 3.3. First (lines 4-14) the new compressed row pointers array *new_row_ptrs*[ ] is created by counting the number of elements in each column (from the *column_index*) and performing a cumulative sum on lines 16-18 of those counts. The *new_row_ptrs*[ ] array will contain the starting row indices of matrix M^T. By definition, a transpose reorders columns to rows and hence

ALGORITHM 3.2: Out-Of-Place sparse matrix transpose

Input: Matrix M as in Data Structure 3.1
Output: New matrix M^T (the transpose of M) in the CSR representation - Containing:
new_nrows - the number of rows in the matrix M^T [= old_ncols]
new_ncols - the number of columns in the matrix M^T [= old_nrows]
new_nnz - the number of non-zero values in the matrix M^T [= nnz]
new_row_ptrs[ ] - the new array of row pointers in M^T
new_non_zeros[ ] - the new array of element values in M^T
new_col_indexes[ ] - the new array of element column indexes in M^T
/* Output arrays are allocated and initialized to zero (not shown for brevity) */

```
1  /* Swap row & column dimensions of the transpose */
2  new_nrows ← ncols;
3  new_ncols ← nrows;
4  /* Count the number of indexes in each column and store in new_row_ptrs[] — offset by 2 */
5  /* Offset by 1 as row x + 1 starts after row x. */
6  /* Offset by 2 to sidestep need to shift indices in new_row_ptrs[] at end of algorithm. */
7  for ( 0 ≤ row < nrows ) do
8  |   for ( old_row_ptrs[row] ≤ k < old_row_ptrs[row + 1] ) do
9  |   |   col ← old_col_indexes[k];
10 |   |   if ( (col + 2) < (new_nrows + 1) ) then
11 |   |   |   new_row_ptrs[(col + 2)] ← new_row_ptrs[(col + 2)] + 1;
12 |   |   end
13 |   end
14 end
15 /* Cumulative sum of new_row_ptrs[] */
16 for ( 0 ≤ row < new_nrows ) do
17 |   new_row_ptrs[row + 1] ← new_row_ptrs[row + 1] + new_row_ptrs[row];
18 end
19 /* Loop through each 'old' row */
20 for ( 0 ≤ row < nrows ) do
21 |   for ( old_row_ptrs[row] ≤ k < old_row_ptrs[row + 1] ) do
22 |   |   /* Copy each element to it's correct new_row position in the transposed matrix */
23 |   |   col ← old_col_indexes[k];
24 |   |   pos ← new_row_ptrs[(col + 1)];          /* offset by 1 */
25 |   |   new_non_zeros[pos] ← old_non_zeros[k];
26 |   |   new_col_indexes[pos] ← row;
27 |   |   new_row_ptrs[(col + 1)] ← new_row_ptrs[(col + 1)] + 1;
28 |   end
29 end
```

the *new_row_ptrs*[ ] array is created by counting the number of differing old column values and accumulating them.

Note: While constructing the *new_row_ptrs*[ ] array we save the column

counts at a position that is offset by two from the column index as can be seen by the $(col + 2)$ array index on line 11. We then access the array offset by one as can be seen by the $(col + 1)$ array index on lines 24 and 27.

Generally when building the $row_ptrs[\,]$ array we store the column counts at a position offset by one. This is because we are looking for the start positions of the new rows. Row $x + 1$ starts after row x and all the rows before, thus we store the count of the number of elements in row x in position $x + 1$. During the algorithm we use the entries in the $new_row_ptrs[\,]$ array to point to the next free slot in each new row and increment the entries as we copy elements to their new position. At the end of the algorithm $new_row_ptrs[x]$ would point to the start of row $x + 1$ and we would need to correct the array by shuffling elements to the right.

If we construct the array offset by two elements, at the end of the algorithm the array will contain the correct entries and the reshuffle is not necessary. The additional arithmetic and control flow (line 10) for the offset by two approach should be optimised by the compiler. Experimental evaluation shows that it is slightly faster is practice. The *if* statement on line 10 ensures we don't index past the end of the array. It is not necessary to count the number of elements in the last column as we know the total number of elements.

Once we have the new $new_row_ptrs[\,]$ array, we traverse through all the nnz values of the matrix row-by-row (lines 20-29). Each element in the row is copied to its correct position in the transposed matrix. The position is found by using the current $column_index$ of the element to index into the $new_row_ptrs[\,]$ array (offset by one) which gives the index in the $new_non_zeros[\,]$ and $new_col_indexes[\,]$ arrays for the element. The non_zero value is copied and the old_row_index becomes the new_column_index. The $new_row_ptrs[\,]$ is incremented so that the next element to be copied to that new row will be put in the next free position in the row.

The resultant transposed matrix M^T in CSR format is shown in Example (3.3).

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
non_zeros	=	a	c	h	d	f	m	g	i	b	j	k	n	e	l	o	
col_indexes	=	0	1	3	1	2	5	2	3	0	3	4	5	1	4	5	
new_row_ptrs	=	0_0			3_1			6_2	7_3	8_4				12_5			15

Example 3.3: Transposed Matrix M^T in CSR representation
(Integer dimensions remain unchanged)

3.4.4 Analysis of Out-of-Place Algorithm

The OOP algorithm performs the transpose in-order, row by row. A beneficial side effect of the order in which the OOP algorithm copies matrix elements is that the values within each new row of the transposed matrix will also be in order (of column index) within the rows. The OOP algorithm is simple and generally fast, running with an asymptotic complexity of $\Theta(nnz + n)$. The input matrix is accessed sequentially which gives good cache locality. Although fast, the algorithm does require $\sim(3nnz + n)$ additional memory. This actually translates to $(12nnz + 4n)$ *bytes* if we assume 8-byte double non-zeros and 4-byte integer indices, meaning OOP requires 100% of the size of the Matrix in overhead[a].

This memory overhead may be acceptable for small matrices, however for larger matrices finding this additional memory may prove difficult, or indeed impossible. Thus, we need an algorithm which performs reasonably well in terms of both space and time.

Experimental analysis of the out-of-place algorithm is shown in Section 3.7.

3.5 The *In-Place* (IP) Sparse Transpose

The space complexity of the OOP algorithm can be reduced by using an *In-Place* (IP) transpose algorithm. As the name suggests, an IP algorithm performs the transpose using the original matrix arrays without making

[a] Rectangular matrices require slightly above or below 100% for different size *row_ptrs*[]

an additional copy of the *non_zeros*[ ] and *col_indexes*[ ] arrays. One approach to the sparse in-place transpose algorithm is to use a *"cycle-chasing"* technique to transpose the matrix within its storage structure. This cycle-chasing is similar to the cycle-chasing used for dense rectangular in-place transpose. However due to the sparsity of the matrix it is not possible to pre-calculate the positions of the elements in the matrix and hence the positions visited by the cycles.

There is very little research literature dealing with the topic of sparse in-place transpose. The Sparskit2 package by Youcef Saad [Saad 94] contains an implementation of an in-place cycle chasing algorithm in Fortran [Backus 56, Backus 57]. The `TRANSP()` subroutine implements the Saad-IP algorithm which we discuss in detail in Algorithm 3.3 in Section 3.5.1.

`HSL` [Group 63, Gould 04] is a mathematical software library from the Numerical Analysis Group, it is a closed source, commercial, collection of FORTRAN codes for large scale scientific computation. HSL includes two routines for performing sparse matrix transpose, `MC38()` mentioned above which uses an out-of-place algorithm and the routine `MC46()` which is an in-place algorithm. Unfortunately the algorithms have not been published. According to the documentation, `MC46()` requires two arrays of size $(m+1)$ in order to perform the transpose. As we will discuss further in Chapter 4, the algorithm seems to only address the first of the three problems for an in-place sparse transpose with a memory overhead $\Theta(n)$. The `MC46()` routine still requires $\Theta(nnz)$ additional space in order to record which elements have been moved and the location of free slots in the matrix.

3.5.1 The Saad In-Place Transpose Algorithm

The pseudo-code for the Saad-IP transpose algorithm is shown in slightly simplified pseudo code split across Algorithm 3.3 (a) (Part I: Saad-IP Initialize) and Algorithm 3.3 (b) (Part II: Saad-IP Main Loop).

The Saad-IP Algorithm 3.3 (a) first expands (lines 4-9) the *row_ptrs*[ ] array into a newly allocated, full *tmp_row_indexes*[ ] array of size $\Theta(nnz)$

ALGORITHM 3.3 (a): The Saad *In-Place* sparse transpose - PART I: Initialize

Input: Matrix M as in Data Structure 3.1
Output: Matrix M containing M^T (the transpose of M) in the CSR representation, with:
  $new_row_ptrs[\,]$ - new array of pointers to row starts $[ncols + 1]$

```
1  /* Moved Flag */
2  #define IS_MOVED −1
3  /* Expand row_ptrs[] into temporary row_indexes[nnz] array */
4  Allocate: tmp_row_indexes[nnz];
5  Initialize: tmp_row_indexes[];          /* Initialize to zero */
6  for ( 0 ≤ row < nrows ) do
7  |   for ( row_ptrs[row] ≤ x < row_ptrs[row + 1] ) do
8  |   |   tmp_row_indexes[x] ← row;
9  |   end
10 end
11 /* Don't need contents of row_ptrs any more – but need to re-allocate new_row_ptrs[] for non-square
   matrices */
12 Free: row_ptrs[];
13 Allocate: new_row_ptrs[ncols + 1];
14 Initialize: new_row_ptrs[];          /* Initialize to zero */
15 /* Count number of entries in each new row (old col) - To build row offsets */
16 for ( 0 ≤ i < nnz ) do
17 |   col ← col_indexes[i];
18 |   new_row_ptrs[ (col + 1) ] ← new_row_ptrs[ (col + 1) ] + 1;
19 end
20 /* Cumulative sum across new_row_ptrs[] */
21 for ( 1 ≤ i ≤ ncols ) do
22 |   new_row_ptrs[i] ← new_row_ptrs[i] + new_row_ptrs[i − 1];
23 end
```

(requiring ~(4 nnz) *bytes*). Essentially converting matrix M from the CSR format into the CCO (Compressed Coordinate) format. Algorithm 3.3 (a), like the OOP Algorithm 3.2, then builds the compressed $new_row_ptrs[\,]$ array (lines 14-21), which indicates the starting indices of the rows in the transposed matrix M^T.

The Cycle chasing part of the algorithm is then shown in Algorithm 3.3 (b). A counter cur_x is used to traverse the nnz values while performing the in place transpose. Counter cur_x starts from the 0th index which is indicated by the arrow 'p' in Example 3.4 with the nnz value a, row_index 0 and column_index 0. This first element is copied to a temporary location

ALGORITHM 3.3 (b): The Saad *In-Place* sparse transpose - PART II: Main Loop

```
1  /* Loop through every element in the matrix */
2  cur_x ← 0;        /* First Element */          cycles ← 0;       /* Num elements processed */
3  while ( true ) do
4      /* Backup current element */
5      src_val ← non_zeros[cur_x];
6      src_col ← col_indexes[cur_x];
7      src_row ← tmp_row_indexes[cur_x];
8      /* Flag that we have taken out this element, so it doesn't need to be moved any more */
9      tmp_row_indexes[cur_x] ← IS_MOVED;
10     repeat
11         cycles ← cycles + 1;        /* Cycle Counter++ */
12         /* Find the transposed position of element in 'src' — next unused space in its new row */
13         dst_x ← new_row_ptrs[src_col];
14         dst_val ← non_zeros[dst_x];         /* Save the element @ dst_x in 'dst' */
15         dst_col ← col_indexes[dst_x];
16         non_zeros[dst_x] ← src_val;         /* Put 'src' in 'dst_x' */
17         col_indexes[dst_x] ← src_row;       /* Old row → New col */
18         new_row_ptrs[src_col] ← new_row_ptrs[src_col] + 1;        /* ← dst_x + 1 */
19         if ( tmp_row_indexes[dst_x] = IS_MOVED ) then
20             goto MOVED;        /* 'dst_x' was an empty slot – indicating the end of a chain */
21         end
22         src_val ← dst_val;        /* Copy 'dst' into 'src' */
23         src_col ← dst_col;
24         src_row ← tmp_row_indexes[dst_x];
25         tmp_row_indexes[dst_x] ← IS_MOVED;        /* Set flag on 'dst_x' */
26         /* Loop until we have gone through all nnz elements */
27     until ( cycles > nnz );
28     goto END;
29     /* MOVED– Find the next un-moved element to start chasing */
30     MOVED:
31     cur_x ← cur_x + 1;
32     if ( cur_x ≥ nnz ) then
33         goto END;        /* We have run off the end of the array — Jump to the end */
34     end
35     if ( tmp_row_indexes[cur_x] = IS_MOVED ) then
36         goto MOVED;        /* cur_x has already been moved, loop again */
37     end
38 end
39 /* END– Right-shift row offsets to make new_row_ptrs[] */
40 END:
41 for ( ncols ≥ i > 0 ) do
42     new_row_ptrs[i] ← new_row_ptrs[i − 1];
43 end
44 new_row_ptrs[0] ← 0;        Free: tmp_row_indexes;
   M→row_ptrs ← new_row_ptrs;
```

src = | b | 4 | 0 | dst = | | |

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
non_zeros	=	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
col_indexes	=	0	4	0	1	5	1	2	0	3	4	4	5	1	4	5	
new_row_ptrs	=	0_0			3_1			6_2	7_3	8_4				12_5			15
tmp_row_indexes	=	0	–	1	1	1	2	2	3	4	3	4	4	5	5	5	
		↑p	↑q							↑r							

Example 3.4: Algorithm 3.3 - Saad-IP Circuit Chasing Step: 1

"*src*". The *tmp_row_indexes*[ ] value for this element is no longer needed, so this array can be re-used to mark that the element has already been examined (using the IS_MOVED flag). Looking up the column_index (0) in *new_row_ptrs*[ ] shows that this element is already in the correct position for the transposed row, so it is copied back to the same location. The algorithm jumps from line 21 to line 31 with a GOTO, where *cur_x* is incremented to the next position in the matrix, index 1. This location is still $\leq nnz$ (line 33), and has not been moved yet (line 36), so the algorithm loops back to line 4.

src = | b | 4 | 0 | dst = | i | 3 | 3 |

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
non_zeros	=	a	–	c	d	e	f	g	h	b	j	k	l	m	n	o	
col_indexes	=	0	–	0	1	5	1	2	0	0	4	4	5	1	4	5	
new_row_ptrs	=	0_0			3_1			6_2	7_3	8_4				12_5			15
tmp_row_indexes	=	0	–	1	1	1	2	2	3	4	3	4	4	5	5	5	
		↑p	↑q							↑r							

Example 3.5: Algorithm 3.3 - Saad-IP Circuit Chasing Step: 2

The algorithm starts cycle chasing again on line 4 at array position 1 where $cur_x = 1$, indicated by arrow 'q' in Example 3.4. The element at 'q' with nz value b is copied into "*src*", and the destination position of the element in the transposed matrix is found on line 14 to be position 8 as indicated by arrow 'r' which currently holds nz value i. The element at position 'r' is copied to the "*dst*" temporary location (lines 14-16). The

element that was at 'q' is copied from "*src*" to position 'r' (lines 17-19) as shown in Example 3.5. The position 'r' was not the end of a chain (line 20), so the element that was at 'r' is copied from "*dst*" to "*src*" (lines 23-26). The algorithm loops back continuing to cycle chase the element from position 'r' which is now in "*src*" until it finds an element with a destination in the original row 0 which will be copied to position 'q' to end the chain.

The Saad algorithm continues chasing the next unmoved element until all elements are moved to their correct *new_row* in the transposed matrix.

3.5.2 Analysis of Saad In-Place Algorithm

Unlike the OOP algorithm the IP cycle chasing transpose does not work sequentially through the matrix. Therefore elements are not transposed "in-order". This results in a transposed matrix where all the elements are in their correct transposed row, however it is unlikely that they are in the correct order by column index within that row.

In many cases this may not be a concern. If elements are required to be in row order, this can be achieved by performing an additional sorting step after the transpose using a technique similar to standard sorting algorithms. This sorting step can also be done in place. In Section 4.5 we describe our algorithm based on QuickSort [Hoare 61, Hoare 62] and Insertion Sort [Knuth 98] which was used for the sorting phase of the experiments presented in Chapters 3, 4, 5 and 6. . The additional sorting step was included in all the transpose timing results such that every transpose algorithm resulted in the same output from the same input.

The Saad-IP algorithm as described above in Algorithm 3.3 is implemented in the Sparskit2 package [Saad 94]. The Saad-IP algorithm exhibits asymptotic time complexity of $\Theta(nnz + n)$ time, however it is more complicated than the OOP algorithm and accesses the elements in random order as it jumps around chasing the cycles which results in poor locality for cache reuse, so in practice is generally slower than the OOP algorithm. As we see in Figure 3.3 from the experimental evaluation in Section 3.7, when dealing with large matrices and where memory is tight, performing

the transpose in place only requires $(4nnz)$ $bytes$[b] of additional memory compared to the $(12nnz + 4n)$ $bytes$ for the OOP algorithm. The performance cost may be an acceptable trade-off, however, the memory overhead is still $\Theta(nnz)$ which can prove significant for large matrices and for many matrices increases at a higher rate than the number of rows (n). For the largest matrix in our collection the OOP algorithm requires 4,698 MiB and the Saad algorithm requires 1,531 MiB.

To address this memory overhead we have developed a collection of new in-place transpose algorithms which reduce the asymptotic memory overhead to $\Theta(n)$ additional storage (see Chapters 4 and 5). Our HyperPartition algorithm (Chapter 6) requires significantly less memory overhead in practice.

3.6 Performance Evaluation of Algorithms

For the purpose of comparison with our new in-place matrix transpose algorithms, in Section 3.7 we present an empirical evaluation of the two existing sparse matrix transpose algorithms previously described in Sections 3.4.3 and 3.5.1. We first outline the methodology used in our experiments and analysis.

This section describes our experimental setup and how the algorithms were analysed. We discuss what experiments were run, the sample input data used, what measurements were recorded and how they were recorded. We also discuss how the data was analysed and presented.

3.6.1 Matrix Collections – Sample Input Matrices

In order to evaluate the different transpose algorithms and to demonstrate the advantages of our proposed new algorithms, we performed an extensive set of experiments on the algorithms. It is important to understand and evaluate how the algorithms function in real world applications rather than in synthetic simulations. We therefore used a set of matrices taken from

[b] $4nnz$ for square matrices, rectangular matrices may require slightly or more or less.

real world applications for our experiments. 259 sample input matrices were collected in the MatrixMarket [Boisvert 95, Boisvert 97] file format (Appendix C) taken from The University of Florida Sparse Matrix Collection [Davis 94, Davis 09, Davis 11a, Davis 11b] maintained by Tim Davis. Tables in Appendix A give detailed information of a sample of the matrices used.

Our main interest is in the performance of our new transpose algorithms on large matrices as this is where the reduction in memory overhead is most desirable. Some low end embedded systems may still be concerned with memory usage of medium size matrices, therefore for these experiments we chose sample matrices from the Florida collection with more than 1 million (1,000,000) non-zero values. A matrix of this size would require roughly 12 MiB to store in memory assuming the matrix is stored in CSR format with 32-bit row and column indices and double precision non-zero values. A total of 259 input matrices were used in the experiments to evaluate the performance of the algorithms. The matrix collection includes 129 symmetric matrices out of the 259. These symmetric matrices are stored with only the lower triangle represented. It is not necessary to transpose a symmetric matrix as the transpose of a symmetric matrix is the same matrix. We included symmetric matrices in our test suite simply treating them as triangular matrices. The transpose of a triangular matrix often occurs in calculations such as in the upper and lower triangular solves after a Cholesky [Golub 96, Stewart 01] decomposition $M = LU$, where the lower triangular L is the transpose of the upper triangular U, i.e. $L^T = U$. Including symmetric matrices as triangular matrices increases the number of input matrices for our experiments.

The largest sample input matrix *nlpkkt240* is a 27,993,600 × 27,993,600 matrix with 401,232,976 non-zero values and requires 4,699 MiB to simply just store it in memory in the CSR format.

Tables in Appendix A give detailed information on a sample of the matrices used in our experiments. Table A.1 shows the dimensions and structure of the matrix. Table A.2 lists the problem domains from which the matrices were produced. Additionally, Tables A.3 and A.4 show the

memory usage and execution time for the algorithms when transposing the sample matrices.

3.6.2 Experimental Setup

The experiments were run on "*Stoker*" a 32 core machine with 4 × Intel octo-core Xeon E7 4820 processor @2.00GHz with 18 MiB Cache and 128 GiB RAM running Debian 6.0. Each core uses two-way simultaneous multi-threading so there is a total of 64 hardware threads. The implementations were compiled with the `Intel C/C++/Fortran` Compiler version 12.0 with the following optimisation flags:

```
-O3 -fast -openmp
```

The PAPI [Browne 00] system was used for collecting performance metrics and information from the machine's hardware counters (See Section 5.9.1).

A subset of the experiments were repeated on other machines which produced results with very similar trends to the experiments run on Stoker. For comparability, consistency and clarity only the results from stoker machine are presented here.

The two main performance properties we are interested in are: The algorithm's execution time and the memory usage of the algorithm.

Algorithm Execution Time

Measurements of algorithm execution time were conducted as follows: Each of the input matrices was read in from the MatrixMarket file into a sparse coordinate structure which was transformed into the Compressed Sparse Row (CSR) format structure. Each algorithm was then used to transpose the algorithm forwards and backwards 19 times. In the case of the in-place algorithms, the cycle chasing and sorting steps were timed separately. The timer was started just before calling the cycle chasing transpose routine and stopped and recorded as soon as it completed. The timer was again started just before calling the sorting routine and stopped as soon as it completed.

For the out-of-place algorithm there is just one step which is measured by the timer. Except where noted, all execution time results presented in this document include the time for both the cycle-chasing phase and the sorting phase combined. The algorithm (cycle chasing) time includes the time for the actual transpose and all steps relating to the transpose — such as building arrays before/after the transpose, counting elements, cycle-chasing, etc.

When conducting time sensitive experiments, background activity on the experimental machine such as scheduled tasks and system interrupts can interfere with the running programs, causing delays and inaccurate timing results. To offset this problem we chose to measure the time for each algorithm and input 19 times. The median value for the 19 timings was used for analysis and generating the graphs. The value of 19 was chosen as it was deemed large enough to satisfactorily reduce the number of outliers in the timing results and is an odd number which has a single median value. The median was used as an average rather than the mean as the mean would also include the timings of any outliers that did occur. The median value should also be more representative than simply taking the fastest value of the 19 runs.

In practice the timing results for the algorithms were very consistent as can be seen in Figure 7.2 on Page 192. This figure shows each of the individual 19 timing results for the HyperPartition Transpose with RadixSort for different numbers of buckets. For example the Serial algorithm indicated with the red plus ('+') has a block of 19 markers at roughly seven seconds when using two buckets and another block of 19 markers just under five seconds when using four buckets. For the serial algorithm, the timings for the 19 runs are almost identical. Some slight variation between the individual timings can be seen in the parallel algorithm shown with the blue ('×'), particularly with 16,384 buckets. The variation in the parallel version is possibly due to other activity on the machine which has more of an affect when running across 32 cores however it is also likely to be a result of different distributions of OpenMP threads across the CPU cores in relation to the location in memory of the data. HyperThreading may

also be an influence if two threads are allocated to a single core.

Analysing Memory Usage

For these experiments we are interested in the *memory overhead* of the algorithms. The *additional memory* on top of that required to simply store the matrix in memory that the algorithms allocate in order to perform their function. The memory overhead required by each algorithm can be calculated from the dimensions of the matrix, however for completeness (and to ensure correctness) the memory allocated by the algorithms was also measured directly. This was done by overloading all memory allocation and deallocation routines in order to track memory usage. The peak memory in use during the algorithm was then compared to the baseline usage for each matrix. Analysis tools such as Valgrind [Nethercote 03, Nethercote 07] were also used to test the algorithm implementations for memory leaks.

3.6.3 Presentation of Data

The graphs in this document have been generated with GNUplot [Williams 90] and show the execution time, memory overhead, cache performance and other measurements of the algorithms presented. As discussed in Section 3.6.2, the memory overhead is the *additional* memory allocated by the algorithm. We are developing algorithms which improve on the memory usage and execution time, therefore: *in all graphs, lower values are better.*

When discussing the size of a matrix we may talk about the number of rows (n), the number of columns (m) or the number of non-zero elements (nnz) in the matrix. For our purposes the number if non-zeros (nnz) is the most appropriate as it generally has the greatest effect on the memory usage and execution time of the algorithms. Our graphs show the performance of the algorithms relative to the number of non-zeros in the matrix.

Due to the very wide range in the number of non-zero elements in the matrices (from 1 Million to 401 million elements) and the even greater relative difference in memory usages and execution time values for the matrices, it is not possible to directly graph the actual experimental results

together. The graphs would be highly skewed and distorted. As such, we have done a number of things to improve legibility of the graphs. First, we have elected where possible to display all the memory and execution time results of all the algorithms relative to the measured results of the Saad In-Place algorithm. Hence for every graph, the measurements of the Saad algorithm are all on a horizontal line at ($y = 1$). For each matrix the results of the other algorithms are then plotted either above or below this line showing that that algorithm performs better or worse than the Saad algorithm for that input matrix, and by what proportion. This applies to all the metrics - Memory / Execution Time / Cache Misses / etc.

The second way we improve legibility is again due to the wide range in the number of non-zeros and the fact that most of the matrices fall closer to the lower end of the range. Even with a log-x axis this results in data points clustered to the left of the graph. In order to improve legibility, we have split the graph along the x-axis with a stretched logarithmic scale from 1,000,000 to 16,000,000 over the left $^3/_4$ of the graph and then a shorter logarithmic scale from 16,000,000 to 420,000,000 over the right $^1/_4$ of the graph.

One disadvantage of the format in which we have chosen to display the results is that it is not possible to directly see the exact measurements and values of memory usage and execution time that the algorithms have for the various input matrices. In many cases the display format also hides the vast improvement in memory usage and runtime that these new algorithms make over the existing algorithms. Table A.3 and Table A.4 show the exact algorithm memory overhead and actual execution time respectively of a number of the larger sample matrices used in the experiments.

3.7 Evaluation of Sparse Transpose Algorithms

Figure 3.3 shows the additional memory overhead of the Out-of-Place algorithm (Alg: 3.2) relative to the Saad In-Place Algorithm (Alg: 3.3). There is a clear similarity and linearity between the algorithms showing that both have a space complexity of $\Theta(nnz)$ with the OOP algorithm requiring

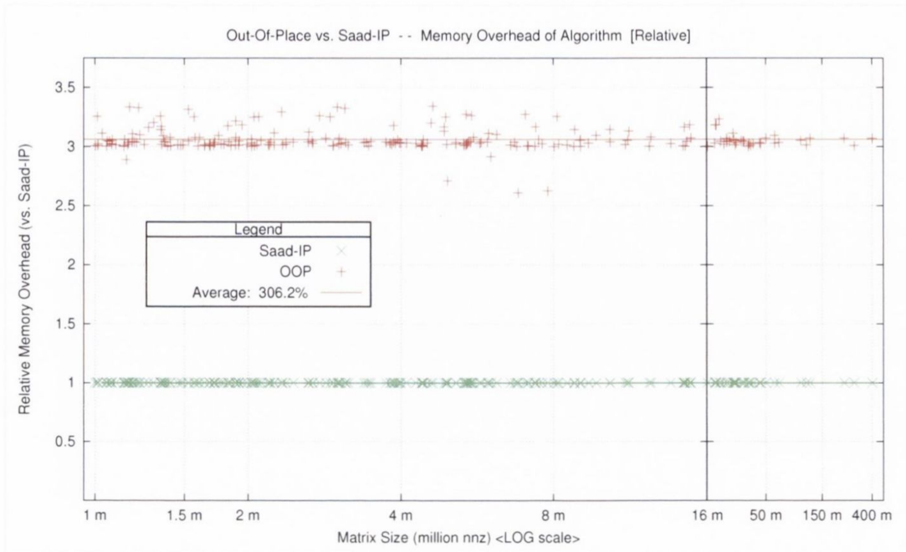


Figure 3.3: Memory overhead of the Out-of-Place (OOP) algorithm relative to the Saad In-Place algorithm. OOP uses roughly 3 times more memory than Saad-IP. For the largest matrix (*nlpkkt240*) with 401 Million non-zeros, OOP requires 4,699 MiB and Saad requires 1,531 MiB in memory overhead.

roughly three times more memory than that of Saad. Variability is due to the differences in lengths of the $row_ptrs[\,]$ array $(nrows + 1)$ compared to the $col_indexes[\,]$ and $non_zeros[\,]$ arrays (nnz). For the largest matrix used in these experiments, (*nlpkkt240*, shown on the far right of the graphs) the OOP algorithm requires 4,698 MiB of additional memory whereas the Saad-IP algorithm requires 1,531 MiB. This is a considerable overhead of 100% and 33% of the original matrix size respectively for both algorithms. More details of matrix memory requirements can be found in Table A.3.

The Algorithm runtime of the Out-of-Place algorithm is shown in Figure 3.4 in comparison to that of the Saad-IP algorithm. OOP is faster in the majority of cases, however there are actually a number of inputs where the OOP algorithm is considerably slower than Saad.

The graphs in Figure 3.3 and Figure 3.4 show both the appeal and drawback of the OOP algorithm. It is simple and (generally) fast, but the graphs show that this comes at a great cost of 100% additional storage, which simply may not be feasible in many cases.

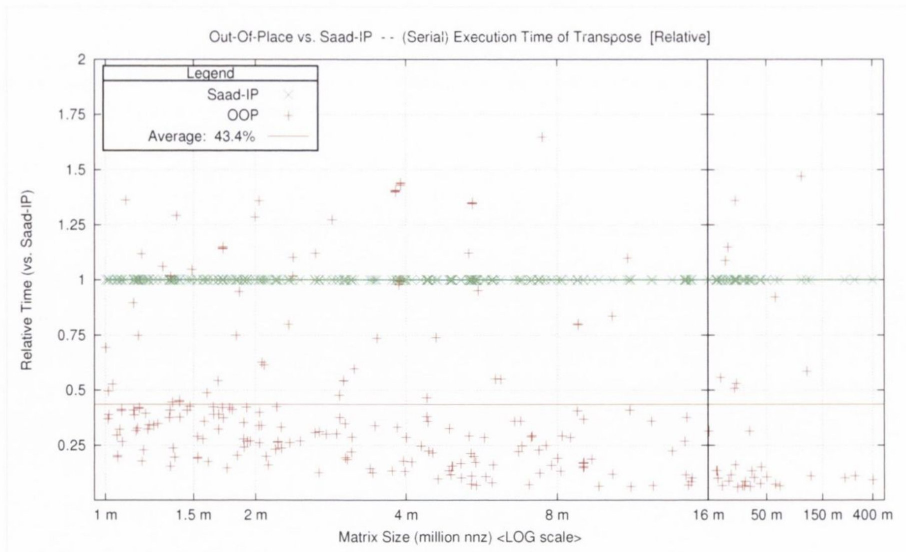


Figure 3.4: Runtime of the Out-of-Place (OOP) algorithm relative to the Saad In-Place algorithm. OOP is generally faster but is slower on some inputs.

3.8 Summary

This chapter reviewed existing research on matrix transpose. There is a large body of existing research on dense matrix transpose with many algorithms, each with their own advantages and disadvantages. However there has been far less focus on transposing sparse matrices. We have reviewed the two major sparse matrix transpose algorithms from the literature and provided an experimental evaluation of both which will be used for comparison in subsequent chapters.

In the forthcoming chapters we present our new Sparse In-Place Transpose algorithms, which reduce the asymptotic space complexity of the sparse transpose while also reducing the execution time compared to the existing in-place algorithm.

Chapter 4

Space Efficient *In-Place* Sparse Matrix Transpose

In the previous chapter we introduced the Matrix Transpose operation and outlined the research and existing algorithms on the topic. We gave an in-depth analysis of the two main algorithms for transposing Sparse Matrices:

1) The Out-of-Place Sparse Matrix Transpose Algorithm (3.2) which has a time complexity of $\Theta(nnz + n)$ and transposes the matrix by making a complete new copy of the full sparse matrix structure in memory and then copies each individual element to its correct location in the transposed matrix. The total space complexity of the Out-of-Place algorithm is $\sim(3nnz + n)$ meaning the Out-of-Place algorithm can require up to 4,698 MiB in memory overhead for our largest matrix.

2) The Saad In-Place Sparse Matrix Transpose Algorithm (3.3) from the package Sparskit2 [Saad 94] which also has a time complexity of $\Theta(nnz + n)$. The Saad algorithm requires an additional temporary array ($tmp_row_indexes[\,]$) of size (nnz) for the row indexes and transposes the matrix in-place using a cycle-chasing technique to move elements. Thus the total space complexity of the Saad In-Place algorithm is $\sim(nnz)$, only 33% of the Out-of-Place algorithm. However even at this level, Saad requires 1,531 MiB of additional memory for the largest matrix.

Performing the transpose in-place with reduced space complexity, the Saad algorithm still requires a considerable memory overhead. As matrices grow larger, this memory overhead will also continue to grow in proportion to nnz. For the great majority of real-world sparse matrices the number of rows (n) in the matrix is far less than the number of non-zeros (nnz). Thus an in-place transpose which only requires $\Theta(n)$ additional space could make a considerable reduction in the memory overhead.

In this chapter we analyse the problems with performing the in-place

sparse matrix transpose operation with only $\Theta(n)$ additional memory and present a number of solutions to these problems.

4.1 In-Place Transpose with Reduced Memory

There are three very important problems which need to be solved in order to reduce the space complexity of the in-place transpose from $\Theta(nnz)$ to $\Theta(n)$.

Problems to solve in order to reduce Space Complexity:

(a) How to find the *old_row_index* of each element in $\Theta(n)$ space

(b) How to record that an element has been processed in $\Theta(n)$ space

(c) How to determine the next free slot in each row in $\Theta(n)$ space and $\Theta(1)$ time

As we move elements from row to row during the cycle chasing transpose the first problem is (a) how to find the old row index of each element. In this chapter and the subsequent chapters we present a number of solutions to this problem.

Problems (b) and (c) are linked, they both relate to the way in which we record that elements have been processed/moved, however the problems manifest themselves in two separate ways. Section 4.1.2 outlines our solution to problems (b) and (c).

Our solutions to these three problems not only allow us to reduce the space complexity of the in-place transpose from $\Theta(nnz)$ to $\Theta(n)$, they also considerably reduce the memory overhead and algorithm execution time in practice as can be seen from the extensive experimental analysis throughout this document.

4.1.1 Finding the *old_row_index* in $\Theta(n)$ Space

The first problem (a) that needs to be addressed in order to reduce the space complexity of the in-place sparse transpose is how to find the *old_row_index* of each element in the matrix.

Consider the original CSR representation of the matrix M from Example (1.1) shown again in Example (4.1). When performing the in place transpose using the cycle-chasing technique (see Sections 3.5 and 4.2), on a matrix in CSR format, at each *jump*[a] in the cycle we know certain information about the element at that position. We know the *position* of the element: p, the value ($non_zeros[p]$) of the non zero element and we know the column index ($col_indexes[p]$) of the element. This *col_index* will become the *new_row_index* of the element in M^T. The information we are missing which can not be looked up *directly* is the current *old_row_index* of the element.

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
non_zeros	=	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
col_indexes	=	0	4	0	1	5	1	2	0	3	4	4	5	1	4	5	
row_ptrs	=	0_0		2_1			5_2		7_3			10_4		12_5			15

Example 4.1: Matrix M in CSR representation

As shown in Section 3.5 the Saad-IP (Algorithm 3.3), solves this problem by expanding the $old_row_ptrs[\,]$ array of size $(n+1)$ into an nnz length array of row indexes ($tmp_row_indexes[\,]$) requiring an additional $\Theta(nnz)$ of memory. As we have seen previously, this complexity can translate to 1,531 MiB of memory overhead. We wish to reduce that overhead.

Usually, when we are accessing the matrix by rows, we look up row 'r' in $row_ptrs[\,]$ in order to find the start, 's', and end, 'e' index of the row within the $non_zeros[\,]$ and $col_indexes[\,]$ arrays. However going the other direction, starting with a random element 'p', in the $non_zeros[\,]$ and $col_indexes[\,]$ arrays, and trying to find the row index of that element is

[a] When we move an element to another row we "jump" to that location in the matrix

more difficult.

Our approach is to perform the in-place transpose using the $row_ptrs[\,]$ array itself, without expanding $row_ptrs[\,]$ into the $tmp_row_indexes[\,]$ array. This eliminates the $\Theta(nnz)$ space requirement in memory overhead and reduces it to $\Theta(n)$. We propose a number of solutions to this problem which are outlined in the following sections and chapters.

Our solutions to finding the old_row_index in $\Theta(n)$ space:

- Binary Range Search Lookup - Section 4.3
- Radix Table Lookup - Section 4.4
- Corresponding Row Table Lookup - Chapter 5

The Binary Range Search and Radix Table Lookup algorithms search the $old_row_ptrs[\,]$ array directly in order to reduce the memory overhead. Although this search does come at an additional cost to the time complexity of the algorithms, in practice the execution time of the Radix Table Lookup algorithm is similar to that of Saad with some variation (see the performance results in Section 4.4.5).

The Corresponding Row algorithm outlined in Chapter 5 uses an additional table of size $\Theta(n)$ in order to look up the old row index in constant $\mathcal{O}(1)$ amortized time in order to maintain the $\Theta(nnz + n)$ runtime of the in-place transpose.

4.1.2 Determine if an Element has Already Been Processed in $\Theta(n)$ Space and $\Theta(1)$ Time.

The second and third problems `(b)` and `(c)` that need to be addressed for the $\Theta(n)$ in-place sparse matrix transpose both relate to how to determine whether or not an element has already been processed. By "processed" we mean that we have either moved the element during a cycle-chasing chain so it does not need to be moved again or we have scanned the element and

found that it does not need to be moved. In either case, we know that element does not need to be checked again.

As we iterate through the elements in the matrix using the cycle-chasing algorithm (see Sections 3.5 and 4.2), we determine if each element is in the correct row in the transposed matrix. If not, we move that element to an area of the matrix arrays corresponding to the correct new row. We continue to move elements in a cycle-chasing manner until the cycle ends. We then continue scanning through the elements in subsequent rows to ensure they are in the correct row. Problem `(b)` arises as we are scanning through the elements in each row, at this point it is important to know whether or not an element has already been processed/moved so that we can skip over it or move it and start cycle-chasing. Problem `(c)` arises when moving an element to another row, at this point it is important to know which elements have already been processed/moved to that row so that we can move the element to the location of the next "free" slot in that row.

How Saad determines if an element has been processed

The Saad in-place Algorithm (3.3) solves problem `(a)` using the $tmp_row_indexes[\,]$ array. When the algorithm is processing an element, it reads its row index from the $tmp_row_indexes[\,]$ array and either moves the element if necessary, starting a cycle, or leaves it in the current row. In both cases the element's $col_index[\,]$ will be updated to contain the old_row_index that was read. Once an element has been processed, the row index value in the $tmp_row_indexes[\,]$ array is no longer needed.

For problem `(b)`, in order to indicate that an element has been processed (moved or updated in place) the Saad algorithm sets a special value (in this case *-1*) in the $tmp_row_indexes[\,]$ array for that element. The Saad algorithm can scan through the elements and skip those with a value of *-1* in the $tmp_row_indexes[\,]$ array.

The Saad algorithm solves problem `(c)` using the $new_row_ptrs[\,]$ array. The $old_row_ptrs[\,]$ array is no longer needed after expanding it

into the $tmp_row_indexes[\,]$ array, thus it can reuse the old array as the $new_row_ptrs[\,]$ array. Thus the Saad algorithm requires that the length of the $row_ptrs[\,]$ array is $max(nrows, ncols)$. When the algorithm needs to move an element to a new row it looks up the index of the start of that row in the $new_row_ptrs[\,]$ array and moves the element to that location. It then increments that index in $new_row_ptrs[\,]$ by one to indicate that the next free slot in that row is the next position in the array. Thus when Saad needs to move an element it can look up the location of the next free position in each row in the $new_row_ptrs[\,]$ array.

The Saad algorithm solves problems `(a)` and `(b)` in $\Theta(nnz)$ space and $\Theta(1)$ time, and solves problem `(c)` in $\Theta(n)$ space and $\Theta(1)$ time. We need to solve both problems in $\Theta(n)$ space and $\Theta(1)$ time.

One possible method to record that an element has been processed in order to solve problem `(b)` would be to create an additional data structure such as a bit vector with one entry for each item. However, this would require an additional $\Theta(nnz)$ space, or nnz bits to be precise. Although for most sparse matrices an additional nnz bits is probably not so very large in comparison to the sparse matrix itself, the space overhead may nonetheless be significant. Furthermore, an additional data structure of $\Theta(nnz)$ is not very satisfying from an theoretical point of view, because the overall space complexity of the algorithm will include an $\Theta(nnz)$ term.

Another solution would be to store a flag in the entries of $col_indexes[\,]$. We cannot use a specific value, such as the -1 used by Saad, as this would over-write the index value already stored there. Instead we could use a high-order bit, or negate the column index value. If the column index is a signed value, then negating the index to indicate a moved value will work. But if the column index is not a signed value, then commandeering the high order bit will reduce the range of column indices, and thus limit the maximum matrix dimension by a factor of two. It is particularly for such large matrices where we wish to have a memory efficient in-place transpose.

Record that an element has been processed in $\Theta(n)$ space and $\Theta(1)$ time

As we have seen, it is not possible to have a "processed" flag for each individual element in the matrix as this would require $\Theta(nnz)$ space. However, having a flag for each individual element is not actually necessary, we just need to be able to distinguish those elements which have been processed from those that have not yet been processed.

Consider again how elements are processed and moved during the cycle-chasing procedure. We process elements sequentially, row by row. When we move an element we move it to the first "free slot" in the new row. This means that all the processed and unprocessed elements will be contiguous together in each row. The processed elements (if any) will be grouped together at the start of the row and all the remaining unprocessed elements will be grouped together towards the end of the row. Consequently, if we record the start and end of the groups of processed and unprocessed elements in each row we will be able to identify which elements have and have not been processed.

As with the Out-of-Place and Saad-IP algorithms, in our $\Theta(n)$ in-place algorithm, we need to construct a $new_row_ptrs[\,]$ array of size $(n+1)$ as part of the transpose process in order to indicate the start of each new row in the matrix. Unlike the Saad algorithm we do not have an nnz length array of $tmp_row_indexes[\,]$ so we need to keep the $old_row_ptrs[\,]$ array in order to look up the row index to solve problem `(a)`. Therefore we will need to allocate a new array of size $(n+1)$ for the $new_row_ptrs[\,]$.

With the $new_row_ptrs[\,]$ array we know the position of the start of each row. This is also the position of the start of the group of processed elements in the row. If we then store the position of the first unprocessed element in each row in a second $(n+1)$ size array called $row_offsets[\,]$ we can then delimit the positions of the processed and unprocessed elements in each row as follows:

Elements in row i:	$new_row_ptrs[\,i\,]$	$\rightarrow$	$new_row_ptrs[\,i+1\,]-1$
Processed in row i:	$new_row_ptrs[\,i\,]$	$\rightarrow$	$row_offsets[\,i\,]-1$
UnProcessed in row i:	$row_offsets[\,i\,]$	$\rightarrow$	$new_row_ptrs[\,i+1\,]-1$

We can use these two arrays of size $(n + 1)$ to solve both problems (b) and (c) as follows. At the start of the algorithm, both arrays are initialized to point to the start of each new row. The $new_row_ptrs[\,]$ array is not modified during the algorithm and always points to the start of the row. When scanning through a row ('i'), we scan through the unprocessed elements starting at $row_offsets[\,i\,]$. If that element does not need to be moved we update its $col_index[\,]$ entry and increment $row_offsets[i]$ to indicate it has been processed and to indicate the position of the next unprocessed element. If the element needs to be moved to another row ('y') we do not want to disturb elements that we have already moved to row 'y' so we find the position of the first unprocessed element in the row from $row_offsets[y]$. We move the element to that position, increment $row_offsets[\,]$ and continue chasing the element we just took out of row y.

All look ups and updates take constant $\Theta(1)$ time. Thus, using these two arrays of size $(n + 1)$ we have solved problems (b) and (c) in $\Theta(n)$ space and $\Theta(1)$ time. It would be preferable to reduce the need for two additional arrays, however for the matrices in the test suite which have more than 65,536 non-zero elements, it is not possible to combine the two arrays. Both arrays are required, the $row_offsets[\,]$ array needs to be incremented to keep track of the first free slot in each row and the elements that have already been moved. The $new_row_ptrs[\,]$ array needs to remain unchanged in order to keep track of the start of each row. Thus, the minimum overhead that we can perform the in-place sparse matrix transpose is $\sim(2n)$ additional memory.

In the next section we show our generic $\Theta(n)$ in-place sparse matrix transpose algorithm which incorporates our solutions to the three problems that need to be solved to reduce the memory overhead from $\Theta(nnz)$ to

$\Theta(n)$.

4.2 Generic *In-Place* Sparse Transpose

Algorithm 4.1 shows our new, Generic $\Theta(n)$ In-Place Cycle-Chasing Sparse Transpose algorithm which demonstrates our solutions to the three problems of transposing a matrix in-place in $\Theta(n)$ memory overhead as discussed in Section 4.1. This Generic Transpose is the basic template which we will base our in-place transpose algorithms on.

Our Generic Algorithm performs the cycle-chasing transpose in a similar manner to Saad-IP as outlined in Algorithm 3.3 in that it performs the same cycle-chasing element movements. The algorithm demonstrates how we use the *new_row_ptrs*[] and *row_offsets*[] arrays discussed in Section 4.1.2 to mark and skip over elements which have already been processed/moved.

The basic procedure of Algorithm 4.1 is: Lines (18-19) – Loop through each new row and each (unmoved) element in that new row. Lines (21-23) – copy that element into "src". Lines (26-31) – find the destination of the element in "src" and copy the element at that destination into "dst". Lines (33-34) – copy element in "src" to the destination. Lines (36-38) – copy element "dst" into "src". Line (24) – keep cycle chasing the element in "src" until the end of the cycle is reached.

The Generic In-Place algorithm calls two external routines in order to find the *row_index* of the element, problem `(a)` outlined in Section 4.1.1. It is these external routines which will change in the forthcoming transpose algorithms. The first is on Line (16), `initialize_lookup_table()`. This placeholder calls a routine which initializes additional lookup tables, if any are required to perform the *row_index* lookup. The second is on Lines (23 and 31), `lookup_row_index()`. This placeholder calls a routine which looks up the *row_index* using the current position in the *non_zeros*[] and *col_indexes*[] arrays as a key. The `lookup_row_index()` procedure potentially uses other data structures initialized earlier with `initialize_lookup_table()`.

The two algorithms take different arguments depending on how they

ALGORITHM 4.1: Generic $\Theta(n)$ *In-Place* Sparse Transpose w/ Row Index Lookup

Input: Matrix M as in Data Structure 3.1

Output: Matrix M containing M^T in CSR, with: *new_row_ptrs*[ ] - new row pointers [cols+1]

```
1  /* Allocate Arrays — Initialized to Zero */
2  Allocate: new_row_ptrs[new_nrows + 1];          Allocate: row_offsets[new_nrows];
3  /* Count number of elements in each new column - offset by 1 */
4  for ( 0 ≤ index < nnz ) do
5  |  col ← col_indexes[index];
6  |  if ( col ≤ (new_nrows − 1) ) then
7  |  |  new_row_ptrs[col + 1] ← new_row_ptrs[col + 1] + 1;
8  |  end
9  end
10 /* Cumulative sum to get new_row_ptrs[] */
11 for ( 1 ≤ row ≤ new_nrows ) do
12 |  new_row_ptrs[row] ← new_row_ptrs[row] + new_row_ptrs[row − 1];
13 |  row_offsets[row] ← new_row_ptrs[row] ;          /* Copy to row_offsets[] */
14 end
15 /* Initialize Lookup Table (if required) */
16 initialize_lookup_table ( )
17 /* Loop through each 'new row' */
18 for ( 0 ≤ row < new_nrows ) do
19 |  for ( row_offsets[row] ≤ x < new_row_ptrs[row + 1] ) do
20 |  |  /* Take out element */
21 |  |  src_nz ← non_zeros[x];
22 |  |  src_col ← col_indexes[x];
23 |  |  src_row ← lookup_row_index( x );
24 |  |  while ( src_col ≠ row ) do
25 |  |  |  /* While element in 'src' does not belong in original 'row' — Cycle Chase element in
   |  |  |     'src' */
26 |  |  |  dst_row ← src_col;
27 |  |  |  dst_x ← row_offsets[dst_row];          /* 'src' should be at position 'dst_x' in
   |  |  |                                             'dst_row' */
28 |  |  |  /* Take out the element at 'dst_x' */
29 |  |  |  dst_nz ← non_zeros[dst_x];
30 |  |  |  dst_col ← col_indexes[dst_x];
31 |  |  |  dst_row ← lookup_row_index( dst_x );
32 |  |  |  /* Put the element we are chasing 'src' into destination slot 'dst_x') */
33 |  |  |  non_zeros[dst_x] ← src_nz;
34 |  |  |  col_indexes[dst_x] ← src_row;
35 |  |  |  /* Put the element in 'dst' in 'src' so we can chase it next */
36 |  |  |  src_nz ← dst_nz;
37 |  |  |  src_col ← dst_col;
38 |  |  |  src_row ← dst_row;
39 |  |  |  /* Increment row_offsets[ */
40 |  |  |  row_offsets[dst_row] ← row_offsets[dst_row] + 1;
41 |  |  end
42 |  |  /* Put 'src' into original position 'x' in the 'row' we started with */
43 |  |  col_indexes[x] ← src_row;
44 |  |  non_zeros[x] ← src_nz;
45 |  end
46 |  Free: M→row_ptrs;      M→row_ptrs ← new_row_ptrs;      Free: row_offsets;
47 end
```

build the lookup tables and perform the lookup. The `lookup_row_index()` routine takes as arguments; x on line 23 and dst_x on line 31. These are the positions in the matrix of the elements whose old_row_index needs to be looked up.

The following sections discuss how we can construct data-structures which are used by these routines in the Generic Transpose to improve on the space and time complexity of the Sparse In-Place Transpose.

4.3 *In-Place* Transpose with Binary Range Search

One option to find the row_index in the $row_ptrs[\,]$ array is to scan through the array looking for the row where index 'p' (the position of the element in the nnz arrays) falls between the start 's' and end 'e' of that row. This would however take $\mathcal{O}(n)$ time and would potentially need to be done for every row index lookup for each of the nnz elements we move, which would be completely infeasible.

A better alternative is to use a technique similar to Binary Search[Knuth 98] we can search the $row_ptrs[\,]$ array in $\Theta(log(n))$ time in order to find 'x', the row_index of the element at index 'p'.

4.3.1 Binary Range Search

Our modified binary search algorithm, Binary Range Search, shown in Algorithm 4.2, is similar to the standard binary search in that it repeatedly bisects the array to find the location of the key. Unlike the standard technique where one searches for key/value pairs which may or may not be in the array, all the keys which we may search for ($0 \leq p < nnz$) are "covered" in the $row_ptrs[\,]$ array. We are looking for the position (index) in the $row_ptrs[\,]$ array which covers the range in which that key value falls. For instance, in the $row_ptrs[\,]$ array in Example (4.1), if we binary search for 10, the routine should return 4, given that:

ALGORITHM 4.2: Index Lookup using Binary Range Search Algorithm

Input: *old_row_ptrs*[ ], *old_nrows*, *key*
Output: *row_index*

```
1  /* Drop to sequential search when array length (high − low) ≤ 20 */
2  LIMIT ← 20;
3  /* Binary search array between 'low' and 'high' */
4  low ← 0;
5  high ← (old_nrows − 1);
6  /* While array longer than 'LIMIT' */
7  while ( (high − low) > LIMIT ) do
8  |   mid ← (low + high)/2;
9  |   if ( key < old_row_ptrs[mid] ) then
10 |   |   high ← mid − 1;
11 |   end
12 |   else
13 |   |   low ← mid;
14 |   end
15 end
16 /* Drop to sequential scan at 'LIMIT' - scanning upwards from 'low' */
17 while ( key ≥ old_row_ptrs[low] ) do
18 |   low ← low + 1;
19 end
20 return( low − 1 );        /* index is 1 below first element greater than 'key' */
```

$$row_ptrs[4](8) \ \leq \ 10 \ < \ row_ptrs[4+1](12)$$

The modified binary range search shown in Algorithm 4.2 uses the bisecting technique to find the location 'x' of the key in the *row_ptrs*[ ] array such that $row_ptrs[x] \ \leq \ key \ < \ row_ptrs[x+1]$. The location '$x$' is then returned as the *row_index* of the element we are looking for. For efficiency, the algorithm drops to a sequential scan below a certain LIMIT. This is in order to improve branch prediction. Sequential search causes $\mathcal{O}(1)$ branch mispredictions whereas binary search causes $\mathcal{O}(log(n))$ branch mispredictions. Thus for smaller values of n, sequential search is usually faster [Uht 97, Brodal 05, Kaligosi 06, Biggar 08b]. A range of values for LIMIT were were tested; $4, 8, 16, 20, 32$, etc. Here we have chosen an array length of 20 for LIMIT as it gave a good average performance in our environment over a number of sample inputs – other values may prove more

efficient on other platforms. The best choice for LIMIT was not investigated in detail as the focus of the work was on the transpose algorithm rather than search optimisation. Improving this parameter would have little impact on the overall performance of the Binary Range Search Transpose as the key lookup is $\mathcal{O}(log(n))$ while we require $\mathcal{O}(1)$.

4.3.2 Cycle-Chasing Transpose with Binary Range Search

Algorithm 4.3 shows the in-place cycle chasing sparse matrix transpose with Binary Range Search. This algorithm is a modification of our Generic in-place transpose algorithm (4.1) for use with the Binary Range Search (Algorithm 4.2).

The Cycle Chasing Binary Range Search Transpose Algorithm (4.3) simply calls the `binary_range_search()` algorithm on lines (21 and 29). The binary search algorithm does not require any table initialization as it searches the $row_ptrs[\,]$ array directly. The only arguments required by `binary_range_search()` are the $old_row_ptrs[\,]$ array, the length (old_nrows) of the array and the '$key = p$' we are searching for. The range search algorithm (4.2) will always return a row index between 0 and old_nrows for any key: $0 \leq key < nnz$. The algorithms presented here assume that the matrix is correct and valid. Input validation could be included in practice but should be separate from the transpose algorithm for efficiency.

Applying the binary range search technique to the cycle chasing algorithm results in the same row_index values as obtained from the expanded $tmp_row_indexes[\,]$ in the Saad-IP algorithm. Hence the transpose with binary range search (Algorithms 4.3 and 4.2) and Saad-IP (Algorithm 3.3), perform the transpose operation using the exact same cycle chasing transformations.

The Binary Range Search Transpose requires a total of two additional arrays ($new_row_ptrs[\,]$ and $row_offsets[\,]$ mentioned earlier) of size $\Theta(n)$, which gives a total memory complexity of $\sim(2n)$ in order to perform the

ALGORITHM 4.3: Sparse Transpose with Binary Row Index Range Search

Input: Matrix M as in Data Structure 3.1

Output: Matrix M containing M^T in CSR, with: *new_row_ptrs*[ ] - new row pointers [cols+1]

```
 1 /* Allocate Arrays — Initialized to Zero */
 2 Allocate: new_row_ptrs[new_nrows + 1];            Allocate: row_offsets[new_nrows];
 3 /* Count number of elements in each new column - offset by 1 */
 4 for ( 0 ≤ index < nnz ) do
 5 |   col ← col_indexes[index];
 6 |   if ( col ≤ (nrows − 1) ) then
 7 |   |   new_row_ptrs[col + 1] ← new_row_ptrs[col + 1] + 1;
 8 |   end
 9 end
10 /* Cumulative sum to get new_row_ptrs[] */
11 for ( 0 ≤ row ≤ nrows ) do
12 |   new_row_ptrs[row] ← new_row_ptrs[row] + new_row_ptrs[row − 1];
13 |   row_offsets[row] ← new_row_ptrs[row] ;        /* Copy to row_offsets[] */
14 end
15 /* Loop through each 'new row' */
16 for ( 0 ≤ row < new_nrows ) do
17 |   for ( row_offsets[row] ≤ x < new_row_ptrs[row + 1] ) do
18 |   |   /* Take out element */
19 |   |   src_nz ← non_zeros[x];
20 |   |   src_col ← col_indexes[x];
21 |   |   src_row ← binary_range_search( old_row_ptrs[], old_nrows, x );
22 |   |   while ( src_col ≠ row ) do
23 |   |   |   /* While element in 'src' does not belong in original 'row' — Cycle Chase element in
   |   |   |   'src' */
24 |   |   |   dst_row ← src_col;
25 |   |   |   dst_x ← row_offsets[dst_row];        /* 'src' should be at position 'dst_x' in
   |   |   |   'dst_row' */
26 |   |   |   /* Take out the element at 'dst_x' */
27 |   |   |   dst_nz ← non_zeros[dst_x];
28 |   |   |   dst_col ← col_indexes[dst_x];
29 |   |   |   dst_row ← binary_range_search( old_row_ptrs[], old_nrows, dst_x );
30 |   |   |   /* Put the element we are chasing 'src' into destination slot 'dst_x') */
31 |   |   |   non_zeros[dst_x] ← src_nz;
32 |   |   |   col_indexes[dst_x] ← src_row;
33 |   |   |   /* Put the element in 'dst' in 'src' so we can chase it next */
34 |   |   |   src_nz ← dst_nz;
35 |   |   |   src_col ← dst_col;
36 |   |   |   src_row ← dst_row;
37 |   |   |   /* Increment row_offsets] */
38 |   |   |   row_offsets[dst_row] ← row_offsets[dst_row] + 1;
39 |   |   end
40 |   |   /* Put 'src' into original position 'x' in the 'row' we started with */
41 |   |   col_indexes[x] ← src_row;
42 |   |   non_zeros[x] ← src_nz;
43 |   end
44 |   Free: M→row_ptrs;      M→row_ptrs ← new_row_ptrs;      Free: row_offsets;
45 end
```

transpose. Even with this extra storage requirement the binary search variant only exhibits asymptotic space complexity of $\Theta(n)$ compared to the $\Theta(nnz)$ required by Saad-IP, which can be a significant saving. This space saving does however come at a cost of an additional $\Theta(log(n))$ overhead in time complexity for every row index lookup, which occurs at every jump in the cycle, giving us a total time complexity of $\Theta(nnz \,.\, log(n))$.

There are a number of techniques and algorithms which could be used as improvements to the binary search. We could possibly use some type of *Binary Tree*[And 62], *BTree* [Bayer 70, Comer 79] or *Trie* [Willard 84, Sinha 04] to reduce the lookup time. However these techniques would require additional structures in addition to the $row_ptrs[\,]$ array and may not be suitable for the "search modification" which we require (for all keys $0 \leq key < nnz$ return the row in which the key falls). In addition, these techniques still have an asymptotic time complexity greater than the direct lookup of Saad-IP (Algorithm 3.3). Hashing techniques [Knott 75] can considerably improve key lookup time however, hashes generally do not maintain the order of the keys, which is a feature we require.

4.3.3 Memory Overhead of in-place Sparse Matrix Transpose with Binary Range Search

The memory usage of the Binary Range Search Transpose (Algorithm 4.3) compared to that of Saad (Algorithm 3.3) is shown in Figure 4.1. The Transpose with Binary Range Search algorithm clearly requires less memory overhead for all input matrices and indeed requires considerably less for a majority of matrices.

On average, the in-place transpose with Binary Range Search requires a memory overhead of just 14% of that of Saad, with the majority of the input matrices requiring even less than this. A handful of matrices in the test suite have a very low sparsity pattern with very low numbers of non_zero elements in each row. This means that the number of rows (n) for these few matrices is much closer in size to the number of non_zeros (nnz). These handful of matrices pull up the average relative overhead overall as

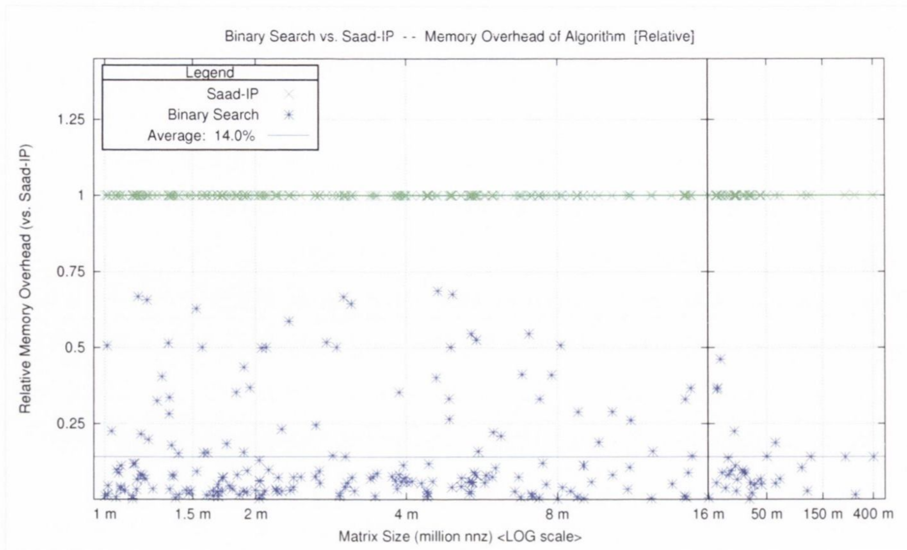


Figure 4.1: Memory overhead of the Binary Range Search transpose algorithm compared to the Saad-IP algorithm. The new algorithm requires considerably less memory than Saad, just 14% on average with most inputs requiring even less.

the reduced $\Theta(n)$ space complexity is less beneficial for these matrices.

4.3.4 Algorithm Execution Time of in-place Sparse Matrix Transpose with Binary Range Search

Figure 4.2 shows the execution time of the Binary Range Search algorithm compared to the execution time of Saad. The Binary Search algorithm performs rather poorly compared to Saad, taking over twice as long as Saad on average to transpose the matrices in the test suite. The additional $\Theta(log(n))$ complexity added to every row index lookup, pushing the time complexity to $\Theta(nnz\,.\,log(n))$ overall causes the execution time performance to deteriorate drastically.

The Binary Range Search Transpose Algorithm is the in-place transpose algorithm with the minimum memory overhead (without modifying the matrix structure) that we can have. It uses just 14% of the memory overhead of Saad. The increase in time complexity is very costly. This runtime may be an acceptable trade-off in very isolated instances if sufficient

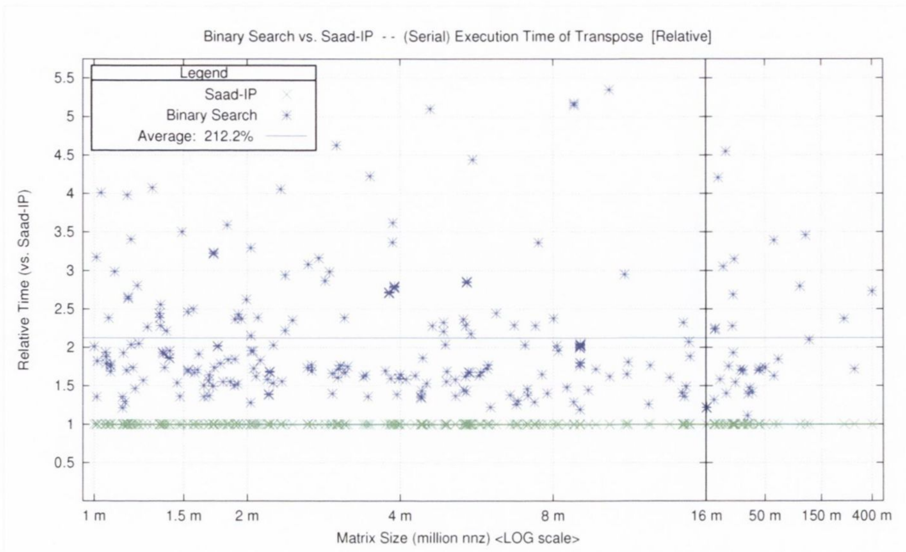


Figure 4.2: Algorithm Execution Time of Binary Range Search Transpose relative to the Saad algorithm. The Binary Range Search algorithm is clearly slower with the $\Theta(nnz \, . \, log(n))$ time complexity apparent.

memory is not available to use the Saad algorithm.

The next section describes an algorithm that uses slightly more additional memory to reduce this runtime overhead while maintaining the $\Theta(n)$ space complexity in memory overhead.

4.4 *In-Place* Sparse Matrix Transpose with Radix Lookup Table

One technique which can improve the runtime performance of the binary search transpose algorithm is to use a radix lookup table to give a shortcut index into the $row_ptrs[\,]$ array. The $\Theta(log(n))$ binary searches of the $row_ptrs[\,]$ array can be reduced, or in many cases avoided using this shortcut. Although using the radix table does not help asymptotically (there is still a non constant lookup time in the worst case), in practice the lookup table exhibits a much faster runtime than the basic binary search.

We build a radix lookup table (an associative array) indexing into

$row_ptrs[\,]$. For each key 'p' (Where: $0 \leq p < nnz$) we right shift the key by $radix_offset$ bits and thus the most significant 'k' bits of the key 'p' are used as indices into the radix table. In turn, the values in the radix table provide us with the index into $row_ptrs[\,]$ which is (ideally) the index of, or index just below, the row_index we are looking for. As such, with a single lookup, the radix table gives us a shortcut into the $row_ptrs[\,]$ array at a position close to the key we are searching for.

4.4.1 Building the Radix Table

Algorithm 4.4 shows the `build_radix()` algorithm which is used to initialize the radix lookup table.

Although the keys that will be searched for in the radix table are in the range $0 \leq key < nnz$ we do not want to have a lookup table that is proportional to $\Theta(nnz)$. We want to keep our memory overhead proportional to $\Theta(n)$. In order to facilitate this we create a radix table proportional to $\Theta(n)$ by creating a radix table of size: *the power of* 2 *just less than or equal to* n. i.e.

$$table_size = 2^{\lfloor log_2 n \rfloor}$$

We choose a $radix_offset$ by which to shift the keys such that they fit in the range:

$$0 \leq (key \gg radix_offset) < table_size$$

This has the disadvantage that a small proportion of the $radix_table$ may be left empty. In retrospect, using:

$$radix_table_size = nnz \gg radix_offset$$

where the offset is chosen to give a table size proportional to $\Theta(n)$ would have given a more appropriate table size.

Using a table size of $\Theta(n)$ maintains the total space complexity of $\Theta(n)$, however it increases the memory overhead by 50% over the Binary Range

Search Transpose. It is also valuable to know if using smaller table sizes proportional to $\frac{n}{2}$, $\frac{n}{4}$ or $\frac{n}{8}$ would be of benefit to performance. Thus the `build_radix()` algorithm (4.4) also takes a *len_mod* parameter which adds to or subtracts from the power-of-two of the size of the radix table. Hence halving or doubling the size of the table each time as appropriate.

Take, for example a matrix with $n = 321{,}826$ rows; $\lfloor log_2(321{,}826) \rfloor = 18$ and $2^{18} = 262{,}144$, therefore for this matrix a radix table size of 262,144 is $\Theta(n)$. Similarly, using the "*len_mod*" parameter we could choose radix table sizes proportional to the number of rows, n, for this 321,826 row matrix as follows:

$$\begin{array}{lcccccr}
\lfloor log_2 n \rfloor - 2 & \rightarrow & \sim(^n/_4) & = & 2^{16} & = & 65{,}536 \\
\lfloor log_2 n \rfloor - 1 & \rightarrow & \sim(^n/_2) & = & 2^{17} & = & 131{,}072 \\
\lfloor log_2 n \rfloor + 0 & \rightarrow & \sim(n) & = & 2^{18} & = & 262{,}144 \\
\lfloor log_2 n \rfloor + 1 & \rightarrow & \sim(2n) & = & 2^{19} & = & 524{,}288 \\
\lfloor log_2 n \rfloor + 2 & \rightarrow & \sim(4n) & = & 2^{20} & = & 1{,}048{,}576
\end{array}$$

The algorithm first (lines 3-5) determines the number of bits required to store the largest *row_index* and the largest *non–zero array index*. Using these, along with the *len_mod* radix size modifier, the algorithm determines (lines 7-20) the parameters for the radix table: *radix_len* — the size of the radix table in bits, *radix_size* — the size of the radix table in bytes and *radix_offset* — the offset in bits that keys will be shifted. The radix table is built (lines 24-29) by scanning backwards through the *old_row_ptrs*[ ] array, each of the radix keys are generated by right shifting the pointer index from the *row_ptrs*[ ] array by *radix_offset* bits. The table is filled in reverse so that it contains the first index in *row_ptrs*[ ] which produces the radix key k. Lines 31-35 then scan forward through the radix table to fill in any holes that may be present.

As we scan backwards through the *row_ptrs*[ ] array we fill the radix table with the values $(5, 4, 2, 0)$ corresponding to rows $(5, 4, 2, 0)$ respectively. This gives the radix table as shows in Example 4.3.

ALGORITHM 4.4: Build Radix Lookup Table

Input: old_nrows, $old_row_ptrs[\,]$, len_mod
Output: $radix_table[\,]$, $radix_offset$

```
1  /* Determine size of radix table and offset */
2  row_bits ← log2(old_nrows);
3  nz_bits ← log2(old_row_ptrs[old_nrows]);
4  diff_bits ← (nz_bits − row_bits);
5  /* Change exponent of 2 by len_mod for radix length of ~ 2^(k−len_mod) where n ≃ 2^k */
6  if ( ( row_bits + len_mod ) < 0 ) then
7      radix_offset ← nz_bits;          /* Size of radix table would be too small */
8      radix_len ← 1;
9  end
10 else
11     if ( diff_bits < len_mod ) then
12         radix_offset ← 0;            /* Size of radix offset would be too big */
13         radix_len ← (nz_bits + 1);
14     end
15     else
16         radix_offset ← (diff_bits − len_mod);
17         radix_len ← (row_bits + 1 + len_mod);
18     end
19 end
20 /* Allocate table */
21 Allocate: radix_table[radix_size + 1];
22 /* Fill radix table in reverse so the table contains the lowest row_ptrs[] 'index' corresponding to 'key' */
23 for ( old_nrows > i > 0 ) do
24     key ← old_row_ptrs[i] >> radix_offset;
25     radix_table[key] ← i;
26 end
27 radix_table[0] ← 0;
28 radix_table[radix_size + 1] ← (old_nrows − 1);
29 /* Fill in holes in the radix table - Scan and replace zero with previous index */
30 for ( 1 ≤ i < radix_size ) do
31     if ( radix_table[i] = 0 ) then
32         radix_table[i] ← radix_table[i − 1];
33     end
34 end
```

4.4.2 Radix Table Lookup

The `lookup_row_index()` routine for the radix table lookup is shown in Algorithm 4.5. Firstly, the key is right shifted by $radix_offset$ bits in order to lookup the radix table and get the shortcut index, 'i', into the $row_ptrs[\,]$

Taking our running example from Examples 1.1 and 4.1, the radix table is built as follows:

Row: 5	$\rightarrow$	$12 \gg 2$	$= 3$	$\rightarrow$	$radix[3]$	$\leftarrow 5$
Row: 4	$\rightarrow$	$10 \gg 2$	$= 2$	$\rightarrow$	$radix[2]$	$\leftarrow 4$
Row: 2	$\rightarrow$	$5 \gg 2$	$= 1$	$\rightarrow$	$radix[1]$	$\leftarrow 2$
Row: 0	$\rightarrow$	$0 \gg 2$	$= 0$	$\rightarrow$	$radix[0]$	$\leftarrow 0$

Example 4.2: Building the Radix Table (in reverse)

radix_table	=	0_0	2_1	4_2	5_4			
old_row_ptrs	=	0_0	2_1	5_2	7_3	10_4	12_5	15

Example 4.3: Radix Table for Matrix M

ALGORITHM 4.5: Index Lookup using a Radix Lookup Table

Input: *radix_table*, *radix_offset*, *old_row_ptrs*[ ], *key*
Output: *row_index*

```
1 x ← (key >> radix_offset);
2 i ← radix_table[x];
3 while ( key ≥ old_row_ptrs[i] ) do
4 |  i ← i + 1;
5 end
6 /* Key is at old_row_ptrs[] at index i or greater */
7 return ( i − 1 );
```

array. Taking this shortcut, the algorithm then starts scanning through the *row_ptrs*[ ] array from that position, 'i', until it finds a value greater than *key*. The *row_index* is the index just before this first location in the array where $row_ptrs[i] > key$, hence we return the value $(i - 1)$ as the *row_index*.

Example (4.3), shows the radix table (*radix_table*[ ]) indexing into the *row_ptrs*[ ] array for our running example matrix M from Examples (1.1 and 4.1). Suppose we are given a key $p = 6$ and we want to look up the row index in the *old_row_ptrs*[ ] array. We first right-shift 'p' by *radix_offset*, in this case $radix_offset = 2$, which gives us $i = 1$, the index into *radix_table*. Next, $radix_table[i = 1] = 2$ gives us the index into

the $row_ptrs[\,]$ array which is below the row_index we are searching for. We start scanning through the array from $row_ptrs[i = 2] = 5$. The next location is $row_ptrs[i = 3] = 7$, this is $\geq key = 6$ hence we stop scanning and return $(i - 1) = (3 - 2) = (2)$ which is the row_index of the element at location 6.

4.4.3 Cycle Chasing Transpose with Radix Table Lookup

The Cycle Chasing Transpose with Radix Table Lookup (Algorithm 4.6) is similar to the Generic in-place algorithm (4.1). There is a call to `build_radix_table()` (Algorithm 4.4) on line 16 to build the radix table, which passes as arguments old_nrows, $old_row_ptrs[\,]$ and len_mod. This returns the $radix_table[\,]$ and $radix_offset$. On lines 23 and 31 the $radix_table[\,]$ is used to lookup the row index by calling `radix_lookup()` (Algorithm 4.5) with parameters; $radix_table[\,]$, $radix_offset$, $old_row_ptrs[\,]$ and x as the lookup key on line 23 and dst_x as the key on line 31.

Similar to the binary search, the radix search needs two $\Theta(n)$ sized arrays for storing the $new_row_ptrs[\,]$ and $row_offsets[\,]$ indices. It also needs a $radix_table[\,]$ which is of similar size to the $row_ptrs[\,]$ array: our experiments have shown that a radix table of size about $n/2$ gives a good trade-off between memory overhead and performance (see Sections 4.4.4 and 4.4.5).

The Radix Table approach requires slightly more memory than binary search (roughly $\sim(2n + n/2)$ for Radix compared to $\sim(2n)$ for Binary) however the asymptotic space complexity is still of order $\Theta(n)$.

Technically the Radix Table lookup algorithm could have a worst case performance of $\mathcal{O}(n)$ for matrices which are particularly degenerate, which is actually a worse complexity than the Binary Range Search method. However, in practice the Radix Lookup technique performs much better than the Binary Search method as can be seen from Figure 4.3 below.

The number of elements scanned during lookups when transposing each matrix with a radix table size of $n/2$ were counted and averaged. The input matrix with the highest average scan length had an average scan length of 2.46 elements, all other matrices had lower averages. This means that most

ALGORITHM 4.6: Sparse Transpose with Radix Table Row Index Lookup

Input: Matrix M as in Data Structure 3.1

Output: Matrix M containing M^T in CSR, with: *new_row_ptrs*[ ] - new row pointers [cols+1]

```
1  /* Allocate Arrays — Initialized to Zero */
2  Allocate: new_row_ptrs[new_nrows + 1];          Allocate: row_offsets[new_nrows];
3  /* Count number of elements in each new column - offset by 1 */
4  for ( 0 ≤ index < nnz ) do
5  |   col ← col_indexes[index];
6  |   if ( col ≤ (nrows − 1) ) then
7  |   |   new_row_ptrs[col + 1] ← new_row_ptrs[col + 1] + 1;
8  |   end
9  end
10 /* Cumulative sum to get new_row_ptrs[] */
11 for ( 0 ≤ row ≤ nrows ) do
12 |   new_row_ptrs[row] ← new_row_ptrs[row] + new_row_ptrs[row − 1];
13 |   row_offsets[row] ← new_row_ptrs[row] ;        /* Copy to row_offsets[] */
14 end
15 /* Initialize Radix Lookup Table */
16 (radix_table[], radix_offset) ←
   build_radix_table( old_nrows, old_row_ptrs[], len_mod )
17 /* Loop through each 'new row' */
18 for ( 0 ≤ row < new_nrows ) do
19 |   for ( row_offsets[row] ≤ x < new_row_ptrs[row + 1] ) do
20 |   |   /* Take out element */
21 |   |   src_nz ← non_zeros[x];
22 |   |   src_col ← col_indexes[x];
23 |   |   src_row ← radix_lookup(radix_table[], radix_offset, old_row_ptrs[], x);
24 |   |   while ( src_col ≠ row ) do
25 |   |   |   /* While element in 'src' does not belong in original 'row' — Cycle Chase element in
   |   |   |   'src' */
26 |   |   |   dst_row ← src_col;
27 |   |   |   dst_x ← row_offsets[dst_row];        /* 'src' should be at position 'dst_x' in
   |   |   |   'dst_row' */
28 |   |   |   /* Take out the element at 'dst_x' */
29 |   |   |   dst_nz ← non_zeros[dst_x];
30 |   |   |   dst_col ← col_indexes[dst_x];
31 |   |   |   dst_row ←
   |   |   |   radix_lookup( radix_table[], radix_offset, old_row_ptrs[], dst_x );
32 |   |   |   /* Put the element we are chasing 'src' into destination slot 'dst_x') */
33 |   |   |   non_zeros[dst_x] ← src_nz;
34 |   |   |   col_indexes[dst_x] ← src_row;
35 |   |   |   /* Put the element in 'dst' in 'src' so we can chase it next */
36 |   |   |   src_nz ← dst_nz;
37 |   |   |   src_col ← dst_col;
38 |   |   |   src_row ← dst_row;
39 |   |   |   /* Increment row_offsets[] */
40 |   |   |   row_offsets[dst_row] ← row_offsets[dst_row] + 1;
41 |   |   end
42 |   |   /* Put 'src' into original position 'x' in the 'row' we started with */
43 |   |   col_indexes[x] ← src_row;
44 |   |   non_zeros[x] ← src_nz;
45 |   end
46 |   Free: M→row_ptrs;     M→row_ptrs ← new_row_ptrs;     Free: row_offsets;
47 end
```

keys were found after scanning 3 elements or less, which are likely to all be read in together in a single cache line. When the algorithm accesses the first element, in the majority of cases the elements following that element in the arrays will also be brought into the cache. Certainly in the majority of cases all these lookup scans will be completed by just reading a single cache line from memory.

It would be possible to use a binary search instead of the sequential scan in the radix lookup. This would give a guaranteed worse case lookup of $\mathcal{O}(log(n))$. However this would introduce more branches and random array look ups into the algorithm and in practice the (short) sequential scan performs better.

Hence, with the radix lookup we do not have a good worst case performance complexity, however for the general case, and for the matrices in our experiments, the radix lookup performs very well with a memory overhead of just $\Theta(n)$.

4.4.4 Memory Usage of Radix Lookup Table Transpose

The memory usage of the Radix Lookup Transpose (Algorithm 4.6) compared to that of Saad (Algorithm 3.3) is shown in Figure 4.3. The graph shows the memory usage of the Radix Table Lookup Transpose with a radix table of size $n/2$. The algorithm requires less memory overhead for all input matrices and indeed requires considerably less for a majority of matrices. On average the Radix Table $n/2$ Transpose requires just 16% of the memory overhead of Saad. Comparing to Figure 4.1 shows that the Radix Table Lookup algorithm requires roughly 25% more additional memory than Binary Range Search.

For the graph in Figure 4.3 we have chose a Radix Table size of $n/2$ as this table size was shown from experimentation to give a good trade-off between memory overhead and runtime performance. Figures 4.4 and 4.5 show the memory usage of the Transpose with Radix Table Lookup with table sizes $n/16$, $n/8$, $n/4$, n, $2n$, $4n$, $8n$ and $16n$. For the smaller table sizes,

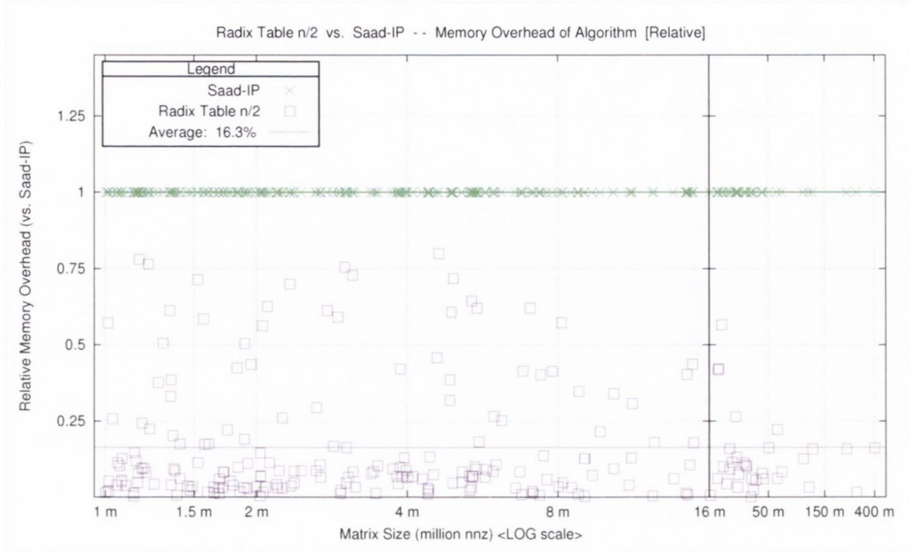


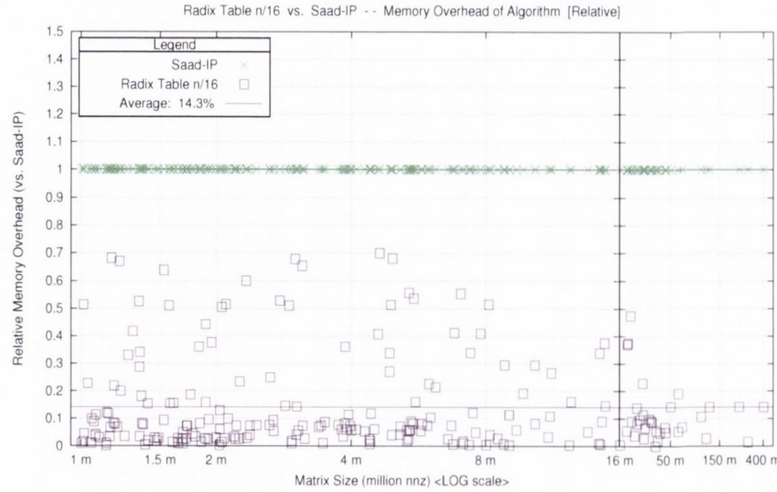
Figure 4.3: Memory overhead of the ($^n/_2$) Radix Table Lookup transpose algorithm compared to the Saad-IP algorithm. The Radix Table algorithm requires considerably less memory than Saad with an average of 16% of Saad. Most inputs require even less than this.

$^n/_{16}$ and $^n/_8$ the memory usage is very close to that of the Binary Range Search Transpose. As the table size increases, the memory overhead for some of the matrices starts to approach (and surpass) the memory required for Saad. For the larger table sizes, $4n$, $8n$ and $16n$, the overhead for some of the input matrices starts becoming considerably larger than Saad.

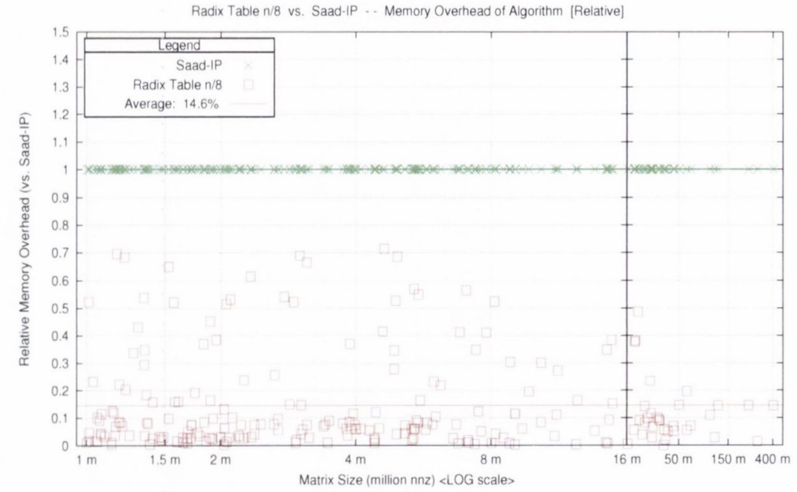
4.4.5 Execution Time of Radix Lookup Table Transpose

Figure 4.6 shows the execution time of the Radix Search ($^n/_2$) algorithm compared to the execution time of Saad. The Radix Search algorithm shows runtime results which are comparable with Saad. These results are with memory overheads that are asymptotically less ($\Theta(n)$ *vs.* $\Theta(nnz)$) than Saad and much less in practice for most inputs, 16% on average.

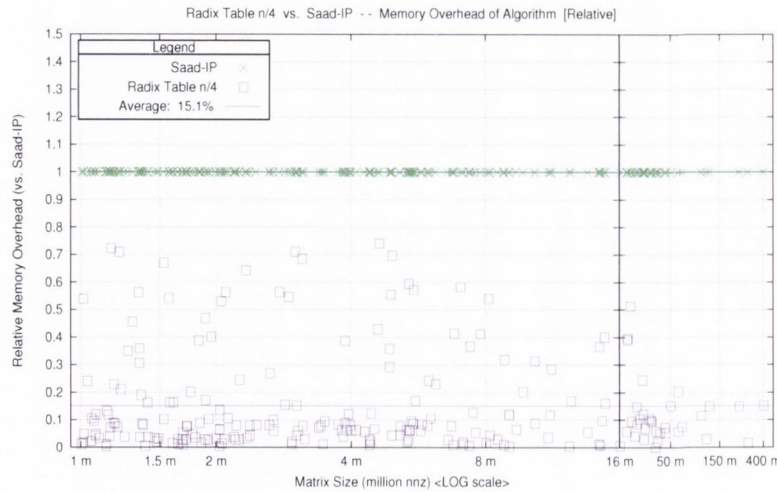
With a table size of about $^n/_2$ the average runtime of the radix table compared to Saad is 98.6% of the runtime of Saad for the 259 matrices in



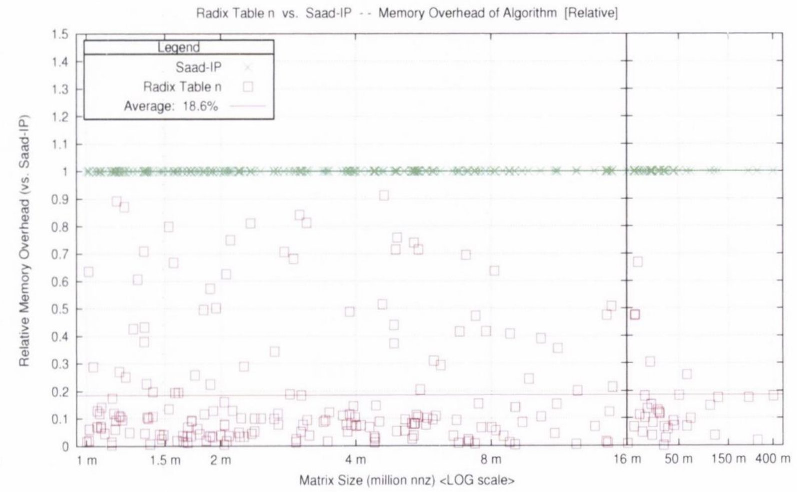
(a) Memory Radix Table $\frac{n}{16}$ - Average: 14%



(b) Memory Radix Table $\frac{n}{8}$ - Average: 15%

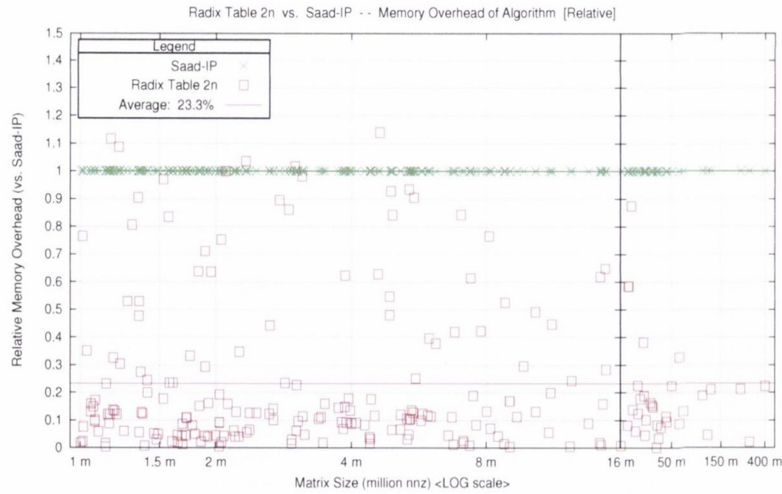


(c) Memory Radix Table $\frac{n}{4}$ - Average: 15%

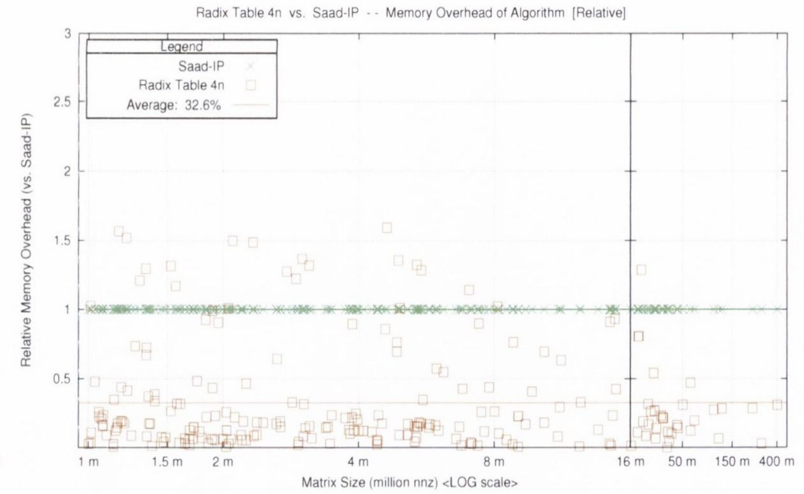


(d) Memory Radix Table n - Average: 19%

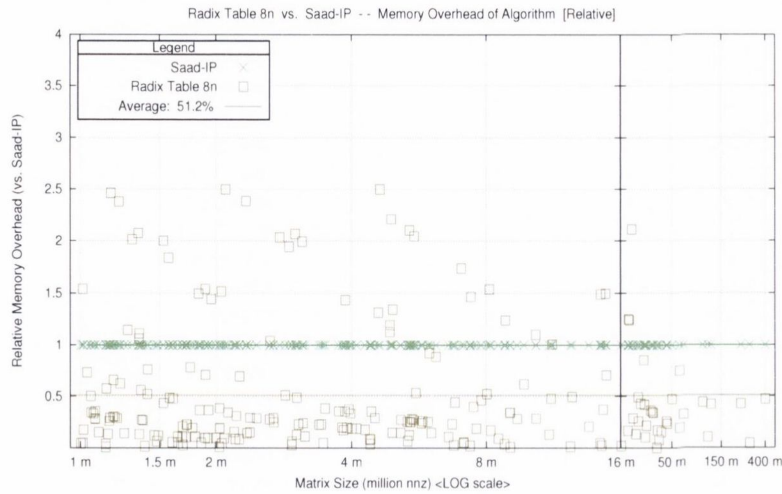
Figure 4.4: Memory Usage of Radix Table Sizes: $\frac{n}{16}$, $\frac{n}{8}$, $\frac{n}{4}$, n



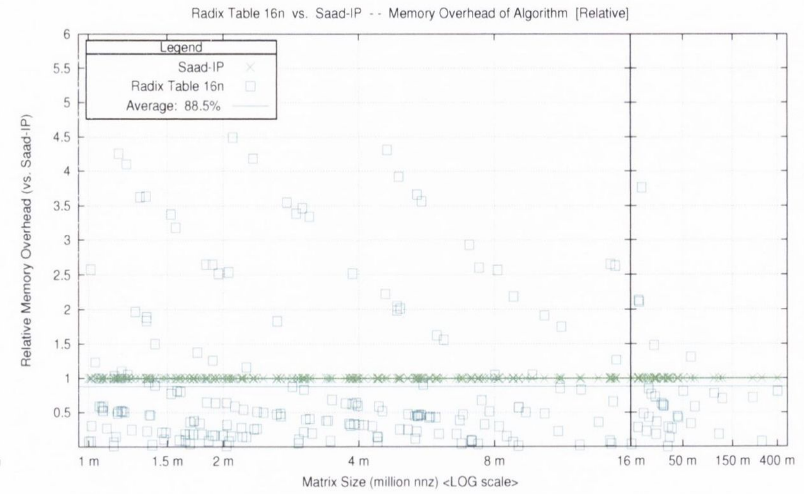
(a) Memory Radix Table $2n$ - Average: 23%



(b) Memory Radix Table $4n$ - Average: 33%



(c) Memory Radix Table $8n$ - Average: 51%



(d) Memory Radix Table $16n$ - Average: 88%

Figure 4.5: Memory Usage of Radix Table Sizes: $2n$, $4n$, $8n$, $16n$

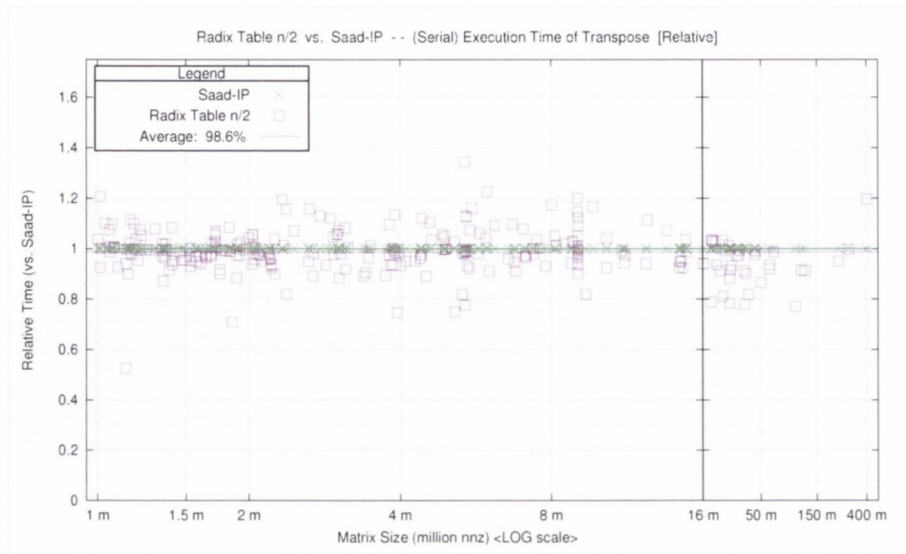


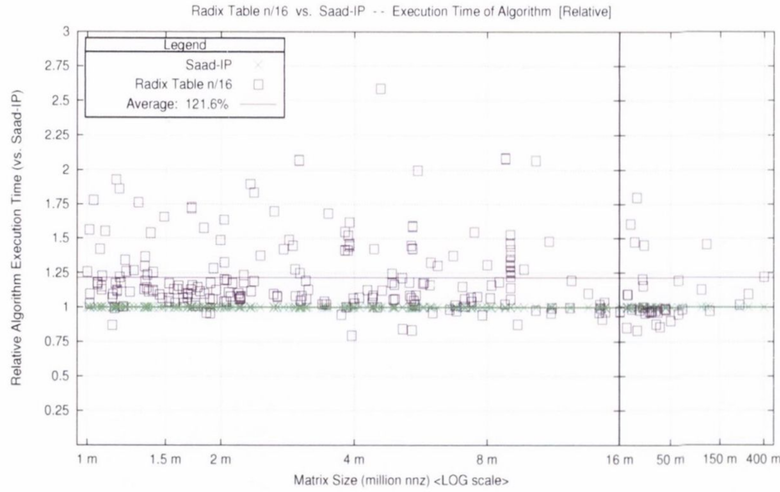
Figure 4.6: Algorithm runtime of ($^n/_2$) Radix Table Lookup Transpose compared with the Saad algorithm. ($^n/_2$) Radix is very much comparable with Saad with an overall runtime of 98% on average. Radix Table is slightly slower in some cases but also noticeably faster in a few cases.

the test suite.

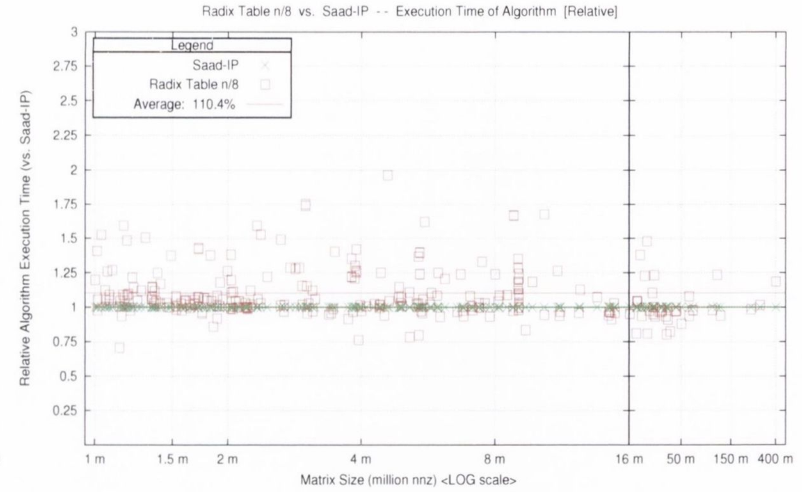
Figures 4.7 and 4.8 show the runtime of the Sparse Transpose with Radix Table lookup for different table sizes of $^n/_{16}$, $^n/_8$, $^n/_4$, n, $2n$, $4n$, $8n$ and $16n$. For smaller table sizes $^n/_{16}$ and $^n/_8$ the runtime of the Radix Transpose is worse than Saad at 120% and 109% respectively. As the table size increases to a size proportional to n the performance improves. However for the larger table sizes of $8n$ and $16n$ the performance deteriorates again as the larger radix tables cause more cache misses.

Table sizes n, $2n$ and $4n$ have slightly better performance than $^n/_2$, however this comes at higher memory usages, with the memory overhead for Radix Transpose approaching that of Saad, even surpassing it for $2n$ and $4n$ for some matrices.

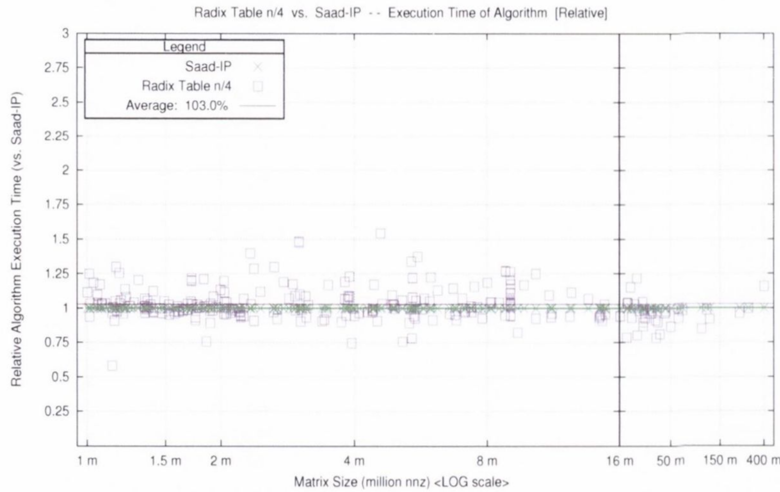
As such, a table size of about $^n/_2$ is recommended as giving reasonable performance for reasonable memory usage (which is still $\Theta(n)$).



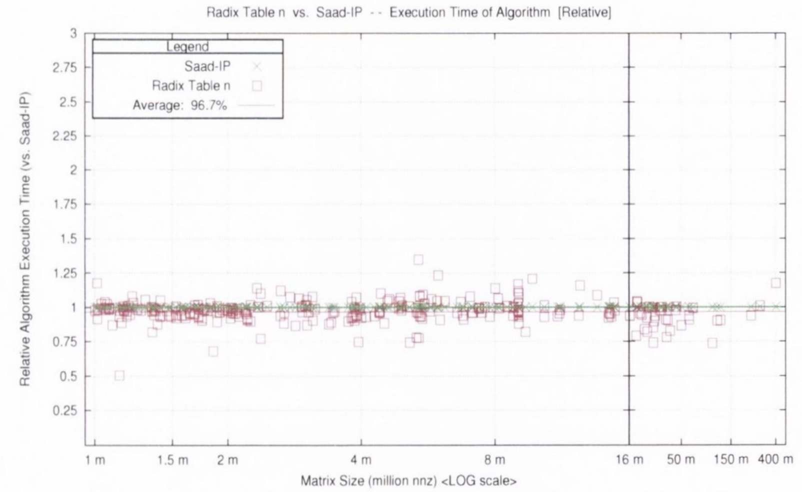
(a) Execution Time Radix $\frac{n}{16}$ - Average 122%



(b) Execution Time Radix $\frac{n}{8}$ - Average 110%

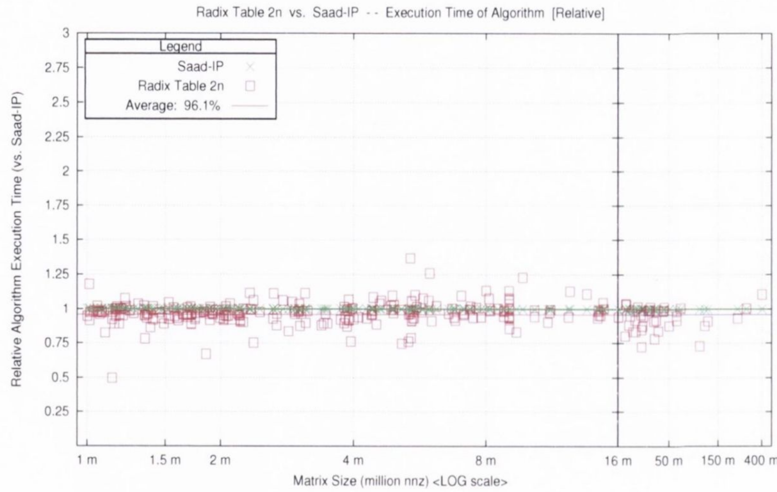


(c) Execution Time Radix $\frac{n}{4}$ - Average 103%

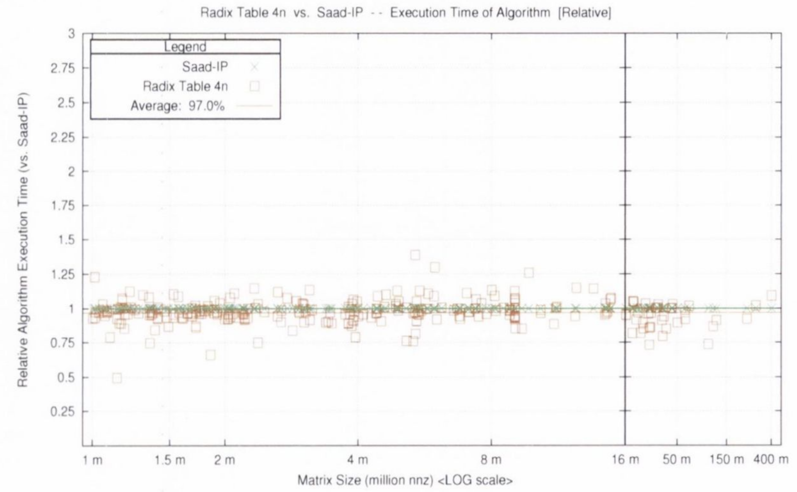


(d) Execution Time Radix n - Average 97%

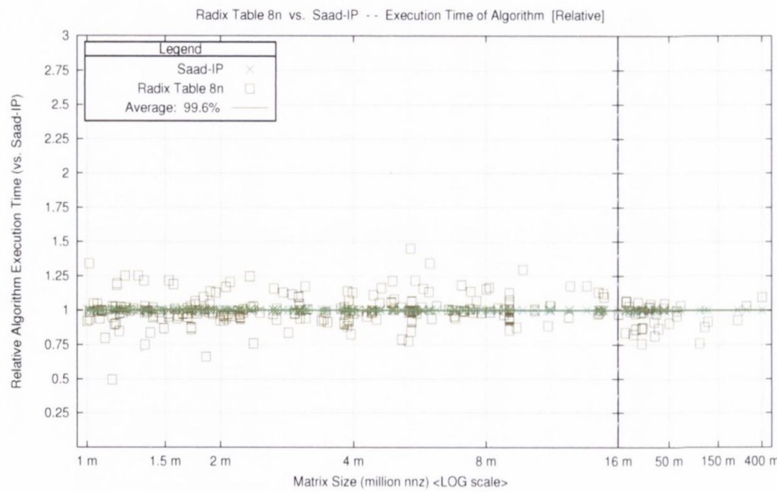
Figure 4.7: Algorithm Runtime of Radix Table Sizes: $\frac{n}{16}$, $\frac{n}{8}$, $\frac{n}{4}$



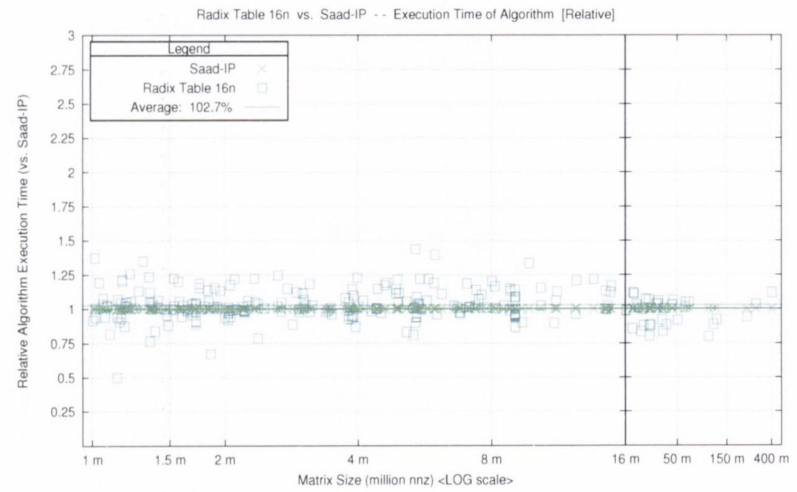
(a) Execution Time Radix $2n$ - Average 96%



(b) Execution Time Radix $4n$ - Average 97%



(c) Execution Time Radix $8n$ - Average 100%



(d) Execution Time Radix $16n$ - Average 103%

Figure 4.8: Algorithm Runtime of Radix Table Sizes: $2n$, $4n$, $8n$, $16n$

4.5 Ensuring In-Row Ordering

Our in-place cycle-chasing transpose algorithm moves elements to the correct transposed row of the matrix *in-place* with ($\Theta(n)$) additional storage. However, just like the Saad in-place algorithm (3.3), due to the way the elements are moved during the cycle-chasing, the elements do not necessarily end up in their correct position within each row in terms of column index order. A side benefit of the way the OOP algorithm (3.2) copies elements is that they automatically end up in their correct position in the transposed row.

In some cases we do not require elements to be in their correct order in the rows, however in most cases it is required or at least preferred. Therefore, for all the in-place algorithms we include an additional step which sorts elements into their correct order in the transposed rows. To give a balanced representation, the runtime for this additional sorting phase is included in the total runtime presented in all the results in this document (except where noted).

Thus the full in-place transpose procedure is as follows: In Phase-I, one of the in-place cycle chasing algorithms is used to move elements to their correct rows in the transposed matrix. In Phase-II, the elements in each individual row are rearranged to ensure that they are in column order.

4.5.1 Sorting Rows with Two Array QuickSort

We can ensure that the elements in the matrix are in their correct order within each row by sorting the values in the two $non_zeros[\,]$ and $col_indexes[\,]$ arrays together at the same time based on the values in the $col_indexes[\,]$ array. Our algorithm for this post-sorting pass is shown in Algorithm 4.7, it uses a technique based on QuickSort [Hoare 61, Hoare 62, Knuth 98] to sort two arrays based on the contents of one.

Algorithm 4.7 takes as input the two arrays; $non_zeros[\,]$ and $col_indexes[\,]$ along with two integers $left$ and $right$ to delimit the section (row) of the arrays to sort. We use a *median of three* to select the pivot. Elements in the

ALGORITHM 4.7: Two Array QuickSort (Median of Three)

Input: Unsorted Row in: *cols*[ ], *vals*[ ], *left*, *right*
Output: The Sorted Row, with: *cols*[ ] and *vals*[ ] sorted by *cols*[ ]

```
1  /* Set the ISort limit and set the initial positions of indexes l and r at left and right */
2  LIMIT ← 32;
3  l ← (left − 1);
4  r ← right;
5  /* QuickSort if array length is greater than LIMIT */
6  if ( right > (left + LIMIT) ) then
7  |   /* Find the mid-point of the array */
8  |   midd ← ( (left + right) / 2 );
9  |   /* Find median of the three elements at left, midd and right and swap them so that they are
   |      in-order */
10 |   if ( cols[ left ] > cols[ midd ] )        then
   |   exchange( cols[ ], vals[ ], left, midd );
11 |   if ( cols[ left ] > cols[ right − 1 ] )  then
   |   exchange( cols[ ], vals[ ], left, (right − 1) );
12 |   if ( cols[ midd ] > cols[ right − 1 ] )  then
   |   exchange( cols[ ], vals[ ], midd, (right − 1) );
13 |   /* Put the median value in the right position */
14 |   exchange( cols, vals, midd, right );
15 |   /* This median value in right is our pivot */
16 |   pivot ← cols[right];
17 |   /* Search the array from left and right for values that are less than and greater than the pivot */
18 |   while ( true ) do
19 |   |   while ( cols[ ++l ] < pivot ) do noop;      /* Do nothing in loop */;
20 |   |   while ( pivot < cols[ −−r ] ) do noop;
21 |   |   /* If the two pointers have overlapped, the array is partitioned - break out of loop */
22 |   |   if ( l ≥ r ) then break;
23 |   |   /* Otherwise exchange the elements */
24 |   |   exchange( cols[ ], vals[ ], l, r );
25 |   end
26 |   /* Put the pivot back in position - 'l' is an element with a column index greater than pivot */
27 |   exchange( cols[ ], vals[ ], l, right );
28 |   /* Array has been partitioned - Recursively call quicksort on each of the partitions. */
29 |   two_array_quicksort( cols[ ], vals[ ], left, (l − 1) );
30 |   two_array_quicksort( cols[ ], vals[ ], (l + 1), right );
31 end
32 else
33 |   /* Otherwise, if the array is shorter than LIMIT, use Insertion Sort to sort the array */
34 |   two_array_isort( cols[ ], vals[ ], left, right );
35 end
```

two arrays are swapped using the `exchange`$(cols[\,], vals[\,], a, b)$ macro. The QuickSort algorithm partitions the arrays and recursively calls itself until the array length is below LIMIT at which point it switches to InsertionSort for efficiency. From brief experimentation a LIMIT of 32 was chosen for the point the algorithm drops to InsertionSort, other values may work better on other systems.

4.5.2 Sorting Sub-Rows with Two Array Insertion Sort

Insertion Sort generally makes more comparisons and moves elements a greater number of times than QuickSort. However when working with very small arrays InsertionSort is more efficient as it has much fewer branch mispredictions [Brodal 05, Biggar 08b]. Also, the short arrays are in the cache and it can do the comparisons and moves very quickly. Calling InsertionSort for small arrays also reduces the number of recursive calls that need be made to QuickSort, thus reducing the number of function calls and the size of the call stack.

Our implementation of The Two Array InsertionSort algorithm is show in Algorithm 4.8. Insertion Sort takes the same arguments as QuickSort, the two matrix arrays $non_zeros[\,]$ and the $col_indexes[\,]$ that we wish to sort along with two integers; $left$ and $right$ which delimit the location of the partition in the two arrays that needs to be sorted.

4.5.3 Execution Time of Sorting

Figure 4.9 shows the total execution time for each matrix for performing the Saad in-place cycle-chasing transpose followed by the post sorting phase of the algorithm. This graph shows the cycle chasing and sorting runtime when performing the transpose with the Saad algorithm. Other algorithms show a similar graph.

The graph uses a stacked area graph to show the differences in runtime between the Saad cycle chasing phase and the sorting phase. The *x-axis* has no scale, it is just the matrices listed one after the other with constant

ALGORITHM 4.8: Two Array InsertionSort

Input: Unsorted Row in: *cols*[ ], *vals*[ ], *left*, *right*
Output: The Sorted Row, with: *cols*[ ] and *vals*[ ] sorted by *cols*[ ]

```
1  /* Loop through the arrays */
2  for ( (left + 1) ≤ i ≤ right ) do
3      /* If we find a col index with a lower value than the index proceeding it */
4      if ( cols[ i ] < cols[ i − 1 ] ) then
5          /* Take the current element out, and put the previous element in it's place */
6          cur_col ← cols[ i ];
7          cur_val ← vals[ i ];
8          cols[ i ] ← cols[ i − 1 ];
9          vals[ i ] ← vals[ i − 1 ];
10         /* Scan backwards until we find an element greater than the current element or reach end
           of array */
11         j ← (i − 1);
12         while ( (j > left) && (cur_col < cols[ (j − 1) ]) ) do
13             /* Shift each previous element along one place to the right */
14             cols[ j ] ← cols[ j − 1 ];
15             vals[ j ] ← vals[ j − 1 ];
16             j ← ( j − 1 );
17         end
18         /* Finally, put the current element at that location */
19         cols[ j ] ← cur_col;
20         vals[ j ] ← cur_val;
21     end
22     /* Continue Scanning through the array */
23 end
```

width in order of matrix size (nnz). The *y-axis* shows the runtime of the algorithm in nanoseconds per non-zero matrix element. This runtime per element is found by taking both the total cycle chasing algorithm runtime and the total sort time for each matrix and dividing that time by the number of elements in the matrix (nnz).

Figure 4.9 shows that the cycle chasing algorithm takes by far the majority of the total runtime. With the sort-time accounting for just a small proportion of the total runtime. For a small number of matrices the sorting phase does take a slightly larger proportion of the runtime, however, overall the execution time of the cycle chasing is dominant.

The Saad Algorithm (3.3) from the previous chapter, the Binary Range Search Algorithm (4.3) and Radix Table Search Algorithm (4.6) from this

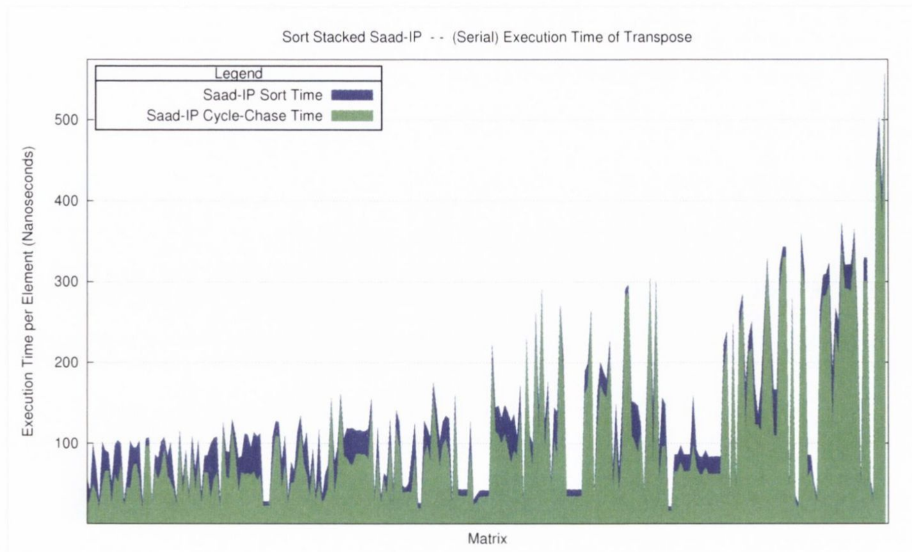


Figure 4.9: Sort Time stacked on top of Algorithm Time

chapter along with the Corresponding Row Algorithm (5.3) presented in the next chapter, all do the same cycle-chasing element movements for each input matrix. Thus the sorting step has the exact same input and performs the exact same sorting operations, taking almost exactly the same time in each case. The only difference in total runtime between the algorithms is in the time for the cycle-chasing transpose.

Section 7.1 will investigate methods to improve the runtime performance of the sorting phase.

4.5.4 Runtime Complexity of Sorting Phase

After transposing the matrix with the cycle chasing transpose the Two Array QuickSort/InsertionSort algorithm is used to sort the rows of the matrix so that they are in column order within the rows.

The sorting algorithm is called $\mathcal{O}(n)$ times in total, once for each of the (n) rows in the matrix. The average number of elements per row is $(\frac{nnz}{n})$. The maximum number of elements in any row is bounded by (n). Sorting a row of (n) elements with QuickSort is $\mathcal{O}(n \, . \, log(n))$. The worst case complexity occurs when the rows being sorted each contain (n)

values. There can be $(\frac{nnz}{n})$ such rows, therefore the overall complexity is $\mathcal{O}(\frac{nnz}{n} \,.\, n \,.\, log(n))$. This gives our worst case complexity for sorting the matrix as $\mathcal{O}(nnz \,.\, log(n))$.

4.6 Conclusion

Both the binary search and radix search methods of in place sparse transpose reduce the memory overhead to $\Theta(n)$.

The Sparse Cycle Chasing Transpose with Binary Range Lookup Algorithm (4.3) transposes the matrix with the least memory overhead of just $\sim(2n)$. This translates to an average memory overhead of 14% of Saad for the 259 matrices. However this memory reduction does come with a cost of increased runtime over Saad to $\Theta(nnz \,.\, log(n) + n)$.

The Sparse Cycle Chasing Transpose with Radix Table Lookup Algorithm uses slightly more memory at $\sim(2n + {}^{n}/_{2})$ for the ${}^{n}/_{2}$ table size which translates to an average of 16% of the memory usage of Saad. The Radix Table algorithm does not maintain the $\Theta(nnz + n)$ runtime complexity of Saad, however in practice it performs the transpose in 98.6% of the time of Saad.

In the next chapter we introduce a novel technique that reduces the runtime of our in place algorithm to $\Theta(nnz + n)$, without forfeiting the reduction in the asymptotic space complexity to $\Theta(n)$.

Chapter 5

Corresponding Row Cycle-Chasing Transpose

Reconsider the running example of the sparse matrix M from previous chapters, repeated below as Example 5.1 and shown again in the CSR sparse matrix storage format in Example 5.2.

Examining our new space efficient transpose algorithms introduced in Chapter 4 we identified that the main overhead contribution to the increased execution time while performing the cycle chasing permutation is finding the row index of the element as we jump from location to location. In this section we introduce our novel technique which finds this row index in a constant amortized $\mathcal{O}(1)$ time while maintaining the reduced space overhead of $\Theta(n)$ of our previous algorithms.

This gives us an in-place algorithm with the same $\Theta(nnz+n)$ time overhead as the Out-of-Place (Algorithm 3.2) and Saad-IP (Algorithm 3.3) while reducing the memory overhead to $\Theta(n)$ compared to the overhead of those existing algorithms; $\Theta(nnz+n)$ for Out-of-Place and $\Theta(nnz)$ for Saad.

$$M = \begin{pmatrix} a & 0 & 0 & 0 & b & 0 \\ c & d & 0 & 0 & 0 & e \\ 0 & f & g & 0 & 0 & 0 \\ h & 0 & 0 & i & j & 0 \\ 0 & 0 & 0 & 0 & k & l \\ 0 & m & 0 & 0 & n & o \end{pmatrix}$$

Example 5.1: Sample Matrix M

5.1 Constant-Time Row Index Lookup

The novel technique is based on a number of key insights about how the cycle chasing algorithm operates:

1. The $new_row_ptrs[\,]$ array divides the destination positions into n groups, where n is the number of new rows in the matrix M.
2. At every jump during cycle chasing we will jump to one of these n rows.
3. There is only one location in each new row that we can jump to – the first "available" slot in the row as indicated by the current value in $row_offsets[\,]$.
4. Therefore, it follows that at any one time, we are only concerned with the row index *corresponding to* those n possible locations we might jump to. The other $(nnz - n)$ locations are not currently important.

Thus from the above, if we could employ a *corresponding row* lookup table of size $\Theta(n)$ to hold the row index for these n locations then it would be possible to perform the cycle chasing in-place sparse transpose with only $\Theta(n)$ space overhead and maintaining the $\Theta(nnz + n)$ time complexity of the existing algorithms.

The challenge is to maintain this table after items are moved during the permutation. To avoid increasing the complexity of the overall algorithm we need to do all updates in constant amortized time.

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
non_zeros	=	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
col_indexes	=	0	4	0	1	5	1	2	0	3	4	4	5	1	4	5	
row_ptrs	=	0_0		2_1			5_2		7_3			10_4		12_5			15

Example 5.2: Matrix M in CSR representation

$src =$ [a_v | 0_c | 0_r] $dst =$ [$_v$ | $_c$ | $_r$]

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
old_row_ptrs	=	0_0		2_1			5_2		7_3			10_4		12_5			15
non_zeros	=	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
col_indexes	=	0	4	0	1	5	1	2	0	3	4	4	5	1	4	5	
		↑p															
new_row_ptrs	=	0_0			3_1			6_2	7_3	8_4				12_5			15
row_offsets	=	0_0			3_1			6_2	7_3	8_4				12_5			15
corresp_table	=	0_0			1_1			2_2	3_3	3_4				5_5			

Example 5.3: Corresponding Row - Step 1

Building the corresponding row table and maintaining it while searching and performing lookups is a little tricky. Example (5.3) shows the structure of the matrix M during the cycle chasing with corresponding row. We have the same 3 arrays from Example 2.2; $old_row_ptrs[\,]$, $col_indexes[\,]$ and $non_zeros[\,]$ along with the two arrays ($new_row_ptrs[\,]$ and $row_offsets[\,]$) required for our $\Theta(n)$ generic in-place transpose outlined in Algorithm 4.1. Example (5.3) also shows a new **corresp_table**[ ] of size $\Theta(n)$ which is required for the corresponding row transpose algorithm. The $corresp_table[\,]$ array stores the row index (from the $old_row_ptrs[\,]$ array), similar to the Saad Algorithm (3.3), but only for the positions of the first element in each new row corresponding to the indexes in the $new_row_ptrs[\,]$ array.

5.2 Using the Corresponding Row

In order to lookup the corresponding row table we need to know the current new_row of the element. We already know the new_row for the first element in a cycle because we are traversing the matrix through each of the new rows. We can see this in Example 5.3. The transpose starts at element 0 as marked by arrow 'p'. As this is the first element we transpose, we already know that it is in new row '0' so we can lookup the element's old_row_index from the corresponding row table as $corresp_table[0] = 0$. Which for element 'p' gives us: $old_row_index = 0$.

When we leave an element in its old position we still need to update its *col_indexes*[ ] entry with its *new_col_index* value. We get this new column index from the element's *old_row_index* which we found by looking it up in the *corresp_table*[ ] array. For element '*p*' $old_row = new_row = 0$ so in this case the update does not change the value in the array.

$src = b_v \; 4_c \; 0_r \qquad dst = \mathrm{i}_v \; 3_c \; -_r$

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
old_row_ptrs	=	0_0		2_1			5_2		7_3			10_4		12_5			15
non_zeros	=	a	b	*c*	*d*	*e*	*f*	*g*	*h*	i	*j*	*k*	*l*	*m*	*n*	*o*	
col_indexes	=	0	4	0	1	5	1	2	0	3	4	4	5	1	4	5	
			↑q							↑r							
new_row_ptrs	=	0_0			3_1			6_2	7_3	8_4				12_5			15
row_offsets	=		1_0		3_1			6_2	7_3	8_4				12_5			15
corresp_table	=	0_0			1_1			2_2	3_3	3_4				5_5			

Example 5.4: Corresponding Row - Step 2

However, for subsequent elements in the cycle, it is a little more difficult. Example 5.4 shows the next element that the algorithm processes, at position 1 marked with arrow '*q*' and value *b*. Again we can directly read the *old_row_index* from $corresp_table[1] = 0$ as this is the first element in the cycle. Element '*q*' does not belong in this *new_row*. From its *old_col_index* (4) we know it belongs in new row 4, so we need to chase this element. We save the value, column and row indexes to '*src*'. The value and column index come straight from the arrays and the row index comes from the corresponding row table lookup. We can find the position of the next free slot in row 4 by looking up $row_offsets[4] = 8$. Thus, element '*q*' needs to be moved to position 8, marked with label '*r*'. We need to take out the element at '*r*' with value *i* and store it in the temporary '*dst*' variable. We know its value and old column index, however we can not directly read the element's *old_row_index* from the arrays. We need to know what *new_row* the element is **currently** in so that we can read the row index from the corresponding row table. The trick is that we know what *new_row* this element is in because it is the *new_row* that the previous element in the

cycle *should* be in. In other words (this is the key to the corresponding row algorithm): we use the *old_col_index* of the previous element in the cycle to index into the corresponding row table to find the element's *old_row_index*. For element '*r*' we can find its new row from $corresp_table[4] = 3$.

In Example 5.5 we can see that the element in '*dst*' which was taken from position '*r*' with the value '*i*' has been updated to have row index '3'. From this point the cycle chasing continues as normal copying the element in '*src*' with value '*b*' to position '*r*' (swapping row and column indexes). The algorithm then copies the element '*i*' from '*dst*' to '*src*' and continues chasing the element in '*src*'.

5.3 Search and Update Corresponding Row Table

The procedure for searching and updating the corresponding row table is shown in Algorithm 5.1. The algorithm requires five arguments; The *corresp_table*[ ] which we will show how to build later, the *old_row_ptrs*[ ] and the number of old rows *old_nrows*, the *new_row* that we are searching for and the index *idx* of the element in the matrix. The search is a straight-forward lookup using the *new_row* key to index into the *corresp_table*[ ] which returns the *old_row_index* corresponding to that *new_row* in the matrix. The *old_row_index* we found is returned at the end of the algorithm.

Every time we search the corresponding row table, we also need to update the contents of the table – "If necessary". The update procedure is show in lines 3-8 of Algorithm 5.1. The table only needs to be updated if the next element '*x*' in the new row after the current element ('*r*') is in a different *old_row* to the current element. The algorithm looks at the pointer for the next *old_row* in the *old_row_ptrs*[ ] array (old_row 4 in the case of element '*r*'). If this pointer has a value *less than* the index of the next element after '*r*' (the element at '*x*' with value '*j*' in this case) then the corresponding row table needs to be updated to contain the index of this new row. Otherwise the next element is in the same row and does not need to be updated. In order to handle empty rows, the algorithm scans through the *old_row_ptrs*[ ] array to find an index pointer *row_ptrs*[*z*] that

has a value greater than the position of the element.

$src =$ b_v 4_c 0_r $dst =$ i_v 3_c 3_r

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
old_row_ptrs	=	0_0		2_1			5_2		7_3			10_4		12_5			15
non_zeros	=	a	–	c	d	e	f	g	h	b	j	k	l	m	n	o	
col_indexes	=	0	–	0	1	5	1	2	0	0	4	4	5	1	4	5	
			$\uparrow_q$							$\uparrow_r$	$\uparrow_x$	$\uparrow_y$					
new_row_ptrs	=	0_0			3_1			6_2	7_3	8_4				12_5			15
row_offsets	=			$\mathbf{2_0}$	3_1			6_2	7_3		$\mathbf{9_4}$			12_5			15
corresp_table	=	$\mathbf{1_0}$			1_1			2_2	3_3	$\mathbf{3_4}$				5_5			

Example 5.5: Corresponding Row - Step 3

In Example 5.5 we can see that both element 'r' at position 8 and the next element ('x') at position 9 are both in *old_row* 3 so the corresponding row entry for *new_row* 4 does not (yet) need to be updated. However, at some later point in the algorithm we will need to move another element to *new_row* 4 which will be put into the arrays at position 9 ('x'). The element at 'x' will need to be taken out and again we lookup its *old_row_index* to be $corresp_row[4] = 3$, however, this time when updating the corresponding row table, the next element in *new_row* 4 with label 'y' at index 10 is in *old_row* 4 therefore the table will need to be updated with this value: $corresp_table[4] \leftarrow 4$.

ALGORITHM 5.1: Searching and Updating the *corresponding_row_table*

Input: *corresp_table*[], *old_row_ptrs*[], *old_nrows*, *new_row*, *idx*
Output: *old_row_index*, Updated *corresp_table*[]

```
    /* idx is the index in non_zeros[] and col_indexes[] of the element we are moving */
1   /* Get the old_row_index corresponding to new_row */
2   old_row_index ← corresp_table[new_row];
3   /* Update the corresponding row table if necessary, skipping over empty rows, if necessary */
4   z ← (old_row_index + 1);
5   while ( (z ≤ nrows) && ( (idx + 1) ≥ old_row_ptrs[z]) ) do
6   |   corresp_table[new_row] ← corresp_table[new_row] + 1;
7   |   z ← z + 1;
8   end
9   return(old_row_index);
```

Examples 5.4 and 5.5 also show the $row_offsets[\,]$ array being updated every time we process an element in the matrix so that the array points to the next "unmoved" element in that particular row.

The corresponding row transpose performs the row index lookup (Algorithm 5.1) in constant $\Theta(1)$ time amortized over the whole transpose algorithm. Each time the `corresp_srch_upd()` routine is called to look up a row index it will check if the table needs to be updated for that row. The while loop will only be entered in the subset of lookups where the table entry needs to be updated. The entry for the row will only need to be updated if the next element in that new row is in a different *old row*. Each time there is a lookup, the conditional on the while will be tested and the loop will only be entered when we cross a row boundary. There can only be $\mathcal{O}(n)$ row boundaries in the array which could potentially be encountered while transposing the nnz elements of the matrix.

When the update does actually enter the loop it will update the entry for the row, then when it tests the loop condition again it will generally find that no more updates are necessary and exit the loop. Thus most updates will only execute the contents of the loop a single time. The only times the loop will execute more than once is when there are empty rows in the matrix. A single update could potentially loop over a large proportion (x) of the row boundaries in $new_row_ptrs[\,]$. However this would mean that there were only $(n - x)$ boundaries left that could potentially cause subsequent lookups to enter the while loop.

Thus over all the $\Theta(nnz)$ lookups, the while loop will only iterate a possible $\mathcal{O}(n)$ times in total. This gives an amortized $\mathcal{O}(1)$ complexity for each lookup over the nnz lookups. Therefore the full cost of all the (nnz) corresponding row lookups and updates during the full corresponding row transpose is no more than $\mathcal{O}(nnz + n)$. This is no more than the time complexity of the cycle-chasing transpose. Thus, searching and updating the corresponding row does not increase the complexity of the corresponding row transpose which is $\Theta(nnz + n)$, just like the existing OOP and Saad sparse transpose algorithms.

ALGORITHM 5.2: Building the *corresponding_row_table*

Input: *old_nrows*, *new_nrows*, *old_row_ptrs*, *new_row_ptrs*
Output: *corresp_table*[] - The Corresponding Row Table

```
1  /* Allocate Corresponding Row Table */
2  Allocate: corresp_table[new_nrows + 1];
3  /* 'j' starts at the end of old_row_ptrs[] */
4  j ← old_nrows;
5  /* for each 'x' in new_row_ptrs[] (in reverse) */
6  for ( new_nrows ≥ x ≥ 0 ) do
7      /* Scan backwards through old_row_ptrs[] to find an index smaller than new_row_ptrs[x] */
8      while ( new_row_ptrs[x] < old_row_ptrs[j] ) do
9          j ← j − 1;
10     end
11     /* corr[x] is the index, 'j' in old_row_ptrs[] of the first element smaller than new_row_ptrs[x] */
12     corresp_table[x] ← j;
13 end
```

5.4 Building the Corresponding Row Table

The corresponding row table can be built in $\Theta(n)$ time. The procedure is shown in Algorithm 5.2. The algorithm requires 4 arguments; The two pointers arrays *old_row_ptrs*[ ] and *new_row_ptrs*[ ] and the respective sizes of the two arrays *old_nrows* and *new_nrows*. The algorithm allocates a new *corresp_table*[ ] of size (*new_nrows* + 1). It is assumed that this array will be deallocated by the caller later.

The corresponding row table stores the *old_row_index* of the first entry in each of the corresponding new rows. The technique to build the table in $\Theta(n)$ time is to scan backwards through both the *old_row_ptrs*[ ] and *new_row_ptrs*[ ] arrays at the same time. For every entry in *new_row_ptrs*[ ] (line 6) we scan backwards through *old_row_ptrs*[ ] (line 8) until we find an entry that is less than or equal to the *new_row_ptr*. The index, 'j' into the *old_row_ptrs*[ ] array that gives us this value is added to the *corresp_table*[ ] array at position 'x', as this is the *old_row_index* corresponding to new row 'x'. The scan is repeated for each of the new rows.

Due to the nested loops the algorithm may appear to be $\Theta(n^2)$ however this is not the case as when we repeat the outer loop, the inner loop

continues scanning the *old_row_ptrs*[ ] array from the same j position it was at the last time. The algorithm only makes a single complete traversal of each of the arrays. Thus the complexity of the build corresponding row algorithm is $\Theta(n)$. Empty rows and rectangular matrices are automatically handled by the algorithm.

5.5 Corresponding Row Cycle Chasing Algorithm

The Corresponding Row Cycle Chasing algorithm is outlined in Algorithm 5.3. Like the algorithms in the previous chapter it is based on our Generic In-Place algorithm described in Algorithm 4.1.

At the start of the Corresponding row algorithm, after building the *new_row_ptrs*[ ] and *row_offsets*[ ] arrays the algorithm calls the `build_corresp_table()` routine from Algorithm 5.2 in order to build the corresponding row table. In order to build the table the routine needs the *old_row_prts*[ ] and *new_row_ptrs*[ ] arrays which are passed as arguments to the routine along with their respective sizes *old_nrows* and *new_nrows*.

The algorithm starts the cycle chasing algorithm processing each element in the matrix as outlined in Section 4. When it needs to find the old row index of an element it calls the `corresp_srch_upd()` routine outlined in Algorithm 5.1 on lines 23 and 31 to search the *corresp_table*[ ] and *old_row_ptrs*[ ] arrays for the old row index of the element, updating the entry if necessary.

The search takes different arguments at the two different times it is called. Both calls take the same first three arguments: *corresp_table*[ ], *old_row_ptrs*[ ] and *old_nrows*. The first call on line 23 also passes the current new *row* that we are processing and the index x of the element in that row that we are starting a chase from. The second call on line 31 also passes as arguments *dst_row*, the destination new row that we want to move the element in '*src*' to and *dst_x*, the index into the *non_zeros*[ ] and *col_indexes*[ ] arrays of the first free slot in that new row.

Sections 5.7 and 5.8 show the memory and execution time of the algorithm under experimental evaluation. Later sections and chapters analyse

ALGORITHM 5.3: Corresponding Row Cycle-Chasing Sparse Transpose

Input: Matrix M as in Data Structure 3.1

Output: Matrix M containing M^T in CSR, with: *new_row_ptrs*[ ] - new row pointers [cols+1]

```
1   /* Allocate Arrays */
2   Allocate: new_row_ptrs[new_nrows + 1];          Allocate: row_offsets[new_nrows];
3   /* Count number of elements in each new column - offset by 1 */
4   for ( 0 ≤ index < nnz ) do
5   |   col ← col_indexes[index];
6   |   if ( col ≤ (nrows − 1) ) then
7   |   |   new_row_ptrs[col + 1] ← new_row_ptrs[col + 1] + 1;
8   |   end
9   end
10  /* Cumulative sum to get new_row_ptrs[] */
11  for ( 0 ≤ row ≤ nrows ) do
12  |   new_row_ptrs[row] ← new_row_ptrs[row] + new_row_ptrs[row − 1];
13  |   row_offsets[row] ← new_row_ptrs[row] ;          /* Copy to row_offsets[] */
14  end
15  /* Build the Corresponding Row Lookup Table */
16  bu|  new_row_ptrs[row] ← new_row_ptrs[rou
      |  row_offsets[row] ← new_row_ptrs[row] ;ws, old_row_ptrs[], new_row_ptrs[])
17  /* Loop through each 'new row' */
18  for ( 0 ≤ row < new_nrows ) do
19  |   for ( row_offsets[row] ≤ x < new_row_ptrs[row + 1] ) do
20  |   |   /* Take out element */
21  |   |   src_nz ← non_zeros[x];
22  |   |   src_col ← col_indexes[x];
23  |   |   src_row ←
    |   |   corresp_srch_upd( corresp_table[], old_row_ptrs[], old_nrows, row, x );
24  |   |   while ( src_col ≠ row ) do
25  |   |   |   /* While element in 'src' does not belong in original 'row' — Cycle Chase element in
    |   |   |   'src' */
26  |   |   |   dst_row ← src_col;
27  |   |   |   dst_x ← row_offsets[dst_row];          /* 'src' should be at position 'dst_x' in
    |   |   |   'dst_row' */
28  |   |   |   /* Take out the element at 'dst_x' */
29  |   |   |   dst_nz ← non_zeros[dst_x];
30  |   |   |   dst_col ← col_indexes[dst_x];
31  |   |   |   dst_row ←
    |   |   |   corresp_srch_upd( corresp_table[], old_row_ptrs[], old_nrows, dst_row, dst_x );

32  |   |   |   /* Put the element we are chasing 'src' into destination slot 'dst_x') */
33  |   |   |   non_zeros[dst_x] ← src_nz;
34  |   |   |   col_indexes[dst_x] ← src_row;
35  |   |   |   /* Put the element in 'dst' in 'src' so we can chase it next */
36  |   |   |   src_nz ← dst_nz;
37  |   |   |   src_col ← dst_col;
38  |   |   |   src_row ← dst_row;
39  |   |   |   /* Increment row_offsets] */
40  |   |   |   row_offsets[dst_row] ← row_offsets[dst_row] + 1;
41  |   |   end
42  |   |   /* Put 'src' into original position 'x' in the 'row' we started with */
43  |   |   col_indexes[x] ← src_row;
44  |   |   non_zeros[x] ← src_nz;
45  |   end
46  |   Free: M→row_ptrs;      M→row_ptrs ← new_row_ptrs;
47  |   Free: row_offsets;     Free: corresp_table;
48  end
```

the performance of the algorithm in greater detail.

5.6 Cache-Friendly Corresponding Row Algorithm

A major concept of this Thesis is the impact of cache performance on the execution time of the algorithms and conversely, modifying algorithms and data-structures such that they make more efficient use of caches. The Corresponding Row algorithm (Algorithm 5.3 outlined in the previous section) allocates two additional work arrays of size $\Theta(n)$ which are completely independent of the arrays which store the matrix in the CSR format and which only exist for the duration of the algorithm. As such we can implement these arrays however we see fit. These two arrays are the $row_offsets[\,]$ and $corresp_table[\,]$ arrays.

Examining the corresponding row cycle-chasing algorithm outlined in Algorithm 5.3 along with the Corresponding Row Search and Update Algorithm 5.1 we see that every time we access these arrays we *always* access the same index in both of the arrays at the same time. We use the current row to index $row_offsets[\,]$ on line 19 and we again use row to index into the $corresp_table[\,]$ array on line 23 via the call to the `corresp_srch_upd()` routine. Later we use dst_row to index the arrays again, $row_offsets[\,]$ on lines 27 and 40 and $corresp_table[\,]$ on line 31 again via the function call.

This information gives us the opportunity to improve the cache locality of the algorithm implementation with a *Cache Friendly* Corresponding Row algorithm. By *interleaving* the two arrays we can improve the temporal and spatial locality of the access to those two arrays. When we access, say, $row_offsets[342]$ then when the memory subsystem fetches that array location from memory, it will also draw the data from the memory locations following that index into the caches. If the arrays are interleaved, that would mean that $corresp_table[342]$ would also be brought into the cache.

The Cache Friendly implementation as shown in Listing 5.1 defines a "*C*" data-structure containing two integers[a], the first ($corr$) will contain

[a] Or whatever variable size is required to represent an index

```
typedef struct _corfoff_s
{
    int    corr;
    int    off;
} corroff_s;

row_index = corroff[row].corr;
// Access Corresponding Row
offset = corroff[row].off;        // Access Row Offset
```

Listing 5.1: Cache Friendly Implementation

the *corresp_table*[ ] entry for the row and the second (off) will contain the *row_offset*[ ]. An array of these *corroff_s* structures is created instead of the two separate arrays. The values of the individual structure members can be accessed using the standard "C" dot (.) notation as shown in Listing 5.1 on lines 7 and 8.

We created two implementations of the corresponding row algorithm for experimental analysis. A "*Normal*" version which has the two separate arrays and a "*Cache Friendly*" version which combines the two arrays.

We can see from the performance evaluation in Section 5.8 and the detailed performance analysis in Section 5.9 that the two algorithms have a better execution time than Saad and that due to the improved cache locality, the Cache Friendly version also performs slightly faster than the normal version.

5.7 Corresponding Row Memory Usage

In Figure 5.1 we see the memory overhead of the corresponding row algorithm which is shown in relative comparison to the memory overhead of the Saad-IP algorithm. Both the Normal Corresponding Row algorithm described in Section 5.5) and the Cache Friendly Corresponding Row Implementation outlined in Section 5.6 use the exact same amount of memory overhead. The Cache Friendly version has a single array which is the same

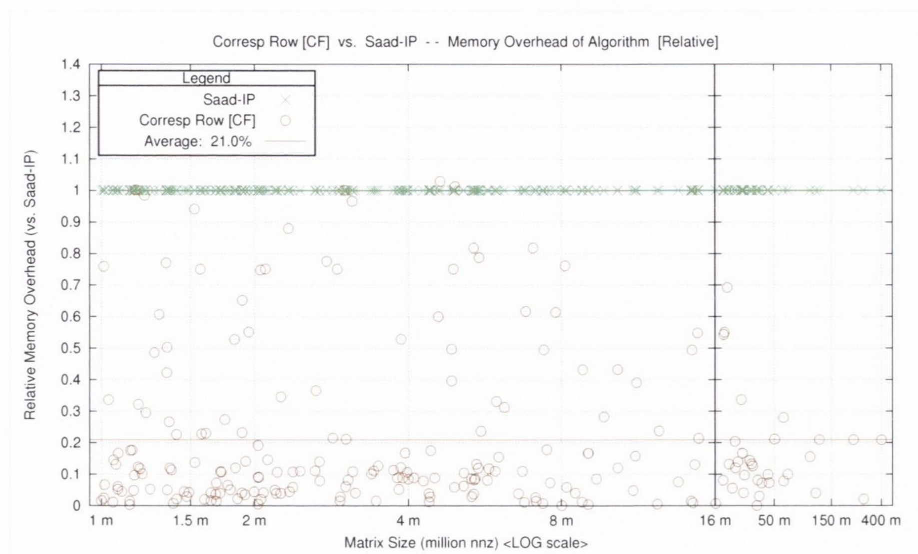


Figure 5.1: Memory overhead of the Corresponding Row algorithm compared to that of Saad. Both Norman and Cache Friendly implementations have the same memory overhead. Corresponding Row uses much less memory ($\leq$ 20% for the majority of inputs) than Saad with just a handful of inputs requiring a memory overhead close to that of Saad. This gives an average of 21% overall.

size as the two separate arrays in the Normal version added together.

The Corresponding Row Algorithm, as outlined in Algorithm (5.3) along with the Build Corresponding Row Table Algorithm (5.2) and the Search/Update Corresponding Row Algorithm (5.1), require the additional storage for the $corresp_table[\,]$ array of size n, along with the two size $\Theta(n)$ arrays $new_row_ptrs[\,]$ and $row_offsets[\,]$ required by the Generic In-Place transpose. This gives a total memory overhead of $\sim(3n)$ for the corresponding row algorithm. This is slightly more than Radix Table Search at $\sim(2n + {}^{n}/_{2})$ and Binary at $\sim(2n)$, however the asymptotic space complexity remains $\Theta(n)$. Figure 5.1 shows that even at $\sim(3n)$ the corresponding row algorithm for the majority of inputs uses considerably less memory than the Saad algorithm, which is $\Theta(nnz)$. In most cases the number of rows is much less than the number of non-zeros ($n \ll nnz$).

The majority of matrices require less than 20% however there are a handful of matrices which are very sparse and hence have a very large num-

ber of rows (*nrows*) compared to the number of non-zero (*nnz*) elements. This causes the corresponding row algorithm to use a higher proportion of memory for these matrices which pushes the memory usage of Corresponding Row much closer to Saad for these handful of matrices, which in turn distorts the average. Thus, on average the corresponding row requires 21% of the memory overhead of Saad.

For very large matrices this reduction represents a significant saving. For the largest matrix, *nlpkkt240*, Saad requires an overhead of 1,531 MiB whereas our Corresponding Row algorithms require just 320 MiB, this constitutes a significant saving of 1,211 MiB. Exact details of memory usages of the different transpose algorithms can be found for a number of sample matrices in Table A.3.

5.8 Corresponding Row Algorithm Execution Time

Figure 5.2 shows the algorithm execution time of Normal Corresponding Row Algorithm relative to Saad and Figure 5.3 shows the algorithm execution time of the Cache Friendly implementation of the Corresponding Row Algorithm relative to Saad. Both algorithms have an asymptotic time complexity of $\Theta(nnz + n)$. The Normal Corresponding Row algorithm is a little faster than Saad for the majority of inputs. The algorithm is slightly slower than Saad for just a few inputs. Overall the Normal Corresponding Row algorithm transposes the matrices in just 92% of the execution time of Saad. The Cache Friendly implementation (Figure 5.3) performs slightly better, improving by 2% to run at 90% the execution time of Saad on average.

This is a very good result for the Corresponding Row algorithm given the memory savings shown in Figure 5.1. This result is somewhat unintuitive given that Corresponding Row performs the same movements of data in the cycle chasing method as Saad yet actually executes more instructions at each step of the algorithm. The improved performance is because Corresponding

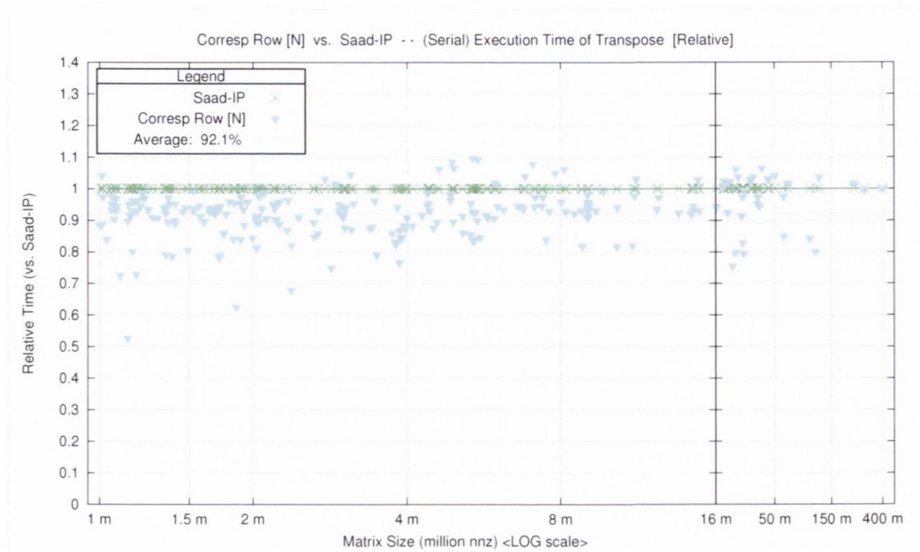


Figure 5.2: Algorithm execution time of the Normal Corresponding Row Transpose compared to Saad. Corresponding Row runs faster in nearly all cases with just a few cases where it is slightly slower. On average the Normal Corresponding Row performs the in-place transpose in 92% of the time of Saad with an average of 21% of the memory overhead.

Row is making better use of the computational resources (Cache/TLB) available. We discuss this further in Section 5.9.

The asymptotic time complexity for corresponding row is $\Theta(nnz + n)$ which is equivalent to the existing In-Place and Out-of-Place algorithms. In the algorithm, nnz elements are moved. Each element requires a look up and update of the $corresp_table[\,]$, which takes amortized $\mathcal{O}(1)$ time. There are some $\Theta(n)$ operations, e.g., computing the initial $new_row_ptrs[\,]$, $row_offsets[\,]$ and $corresp_table[\,]$ arrays, hence the overall complexity of $\Theta(nnz + n)$.

As with all the previous In-Place algorithms, the corresponding row algorithm does not preserve the column order within new rows. Hence a post sorting pass is required to ensure in-row ordering. The QuickSort/InsertionSort based technique described in Section 4.5 was used for the post sort pass. All the timing results for the algorithms thus far have included the time to perform the post sorting step using QuickSort/InsertionSort.

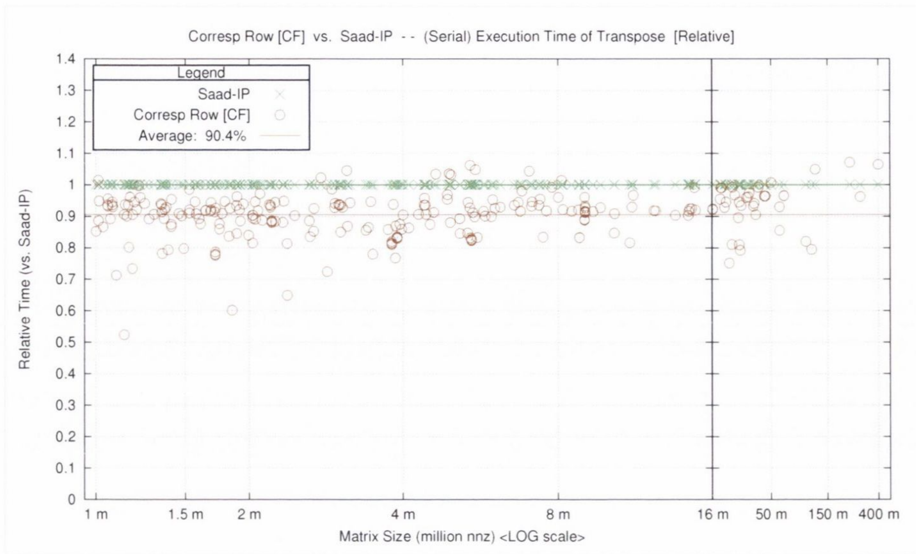


Figure 5.3: Algorithm execution time of the Cache Friendly Corresponding Row Transpose compared to Saad. CF Corresponding Row runs faster in nearly all cases with just a few cases where it is slightly slower. Comparing to Figure 5.2 shows that the Cache Friendly version is slightly faster than the normal, transposing the 259 matrices in just 90% of the time of Saad, again at 21% of the memory overhead.

The sorting step has little effect on the results as for every algorithm the sort performs the exact same operations on the exact same data taking almost the exact same time in every case. The time required to perform the sorting is included in the execution time experiments displayed in Figures 5.2 and 5.3.

These figures show that despite the considerable memory reductions, requiring just 21% on average, and increased complexity of the algorithm, the corresponding row algorithm is similar to the existing Saad algorithm and is faster in the majority of cases, taking just 90% of the time on average.

5.9 Corresponding Row Performance Evaluation

The previous sections presented experimental results of the memory and execution time of the two new corresponding row in-place transpose implementations compared to the existing Saad in-place algorithm. We found that both the corresponding row algorithms required much less memory than Saad, about 21% on average. Both new algorithms also had a slightly faster execution time than Saad transposing the matrices on average in 92% and 90% of the execution time of Saad respectively.

In this section we investigate why the corresponding row algorithms perform better than Saad. Corresponding Row and Saad have the same time complexity of $\Theta(nnz + n)$, indeed the new algorithms perform more operations at each step in the cycle chasing algorithm, yet they have a better execution time in practice. The new algorithms also perform the exact same cycle-chasing operations, moving the same elements to the same locations in the matrix as Saad. So how can the two variants of the corresponding row algorithm achieve the performance improvements shown in Figure 5.2 and Figure 5.3? The reason for the improved performance is the reduced memory overhead from $\Theta(nnz)$ to $\Theta(n)$, which can be a significant difference for some matrices. This reduced memory leads to more efficient use of the caches and TLB and other computational resources available.

5.9.1 Hardware Counters

In order to gain a better understanding of how the algorithms operate in practice, on real machines, the algorithm testing framework was instrumented with code using PAPI [Browne 00] and PerfCtr [Pettersson 05]. This allowed us to access the in-processor hardware counters in order to measure how the algorithms were performing in terms of cache and TLB misses and other metrics. Table 5.1 lists the PAPI events that were monitored and their descriptions. As it is not possible to measure all events at

the same time, events were monitored individually (or in small groups) for multiple runs and the median values taken. Details of the machine, Stoker, that the experiments were run on are given in Section 2.4.

Event	Description
BR_TKN	Conditional branch instructions Taken
BR_MSP	Conditional branch instructions mispredicted
L1_TCM	Level 1 cache misses
L2_TCM	Level 2 cache misses
L3_TCM	Level 3 cache misses
L1_LDM	Level 1 load misses
L2_LDM	Level 2 load misses
LST_INS	Load/store instructions completed
RES_STL	Cycles stalled on any resource
TLB_TL	Total translation lookaside buffer misses
TOT_CYC	Total cycles
TOT_INS	Instructions completed

Table 5.1: Monitored PAPI Events

5.9.2 Branch Misses of CF Corresponding Row

We can see a comparison of the complexity of the Corresponding Row algorithm compared to Saad in Figure 5.4 which shows the relative number of branch misses of the two algorithms. Corresponding Row has many more branch mispredictions for the majority of inputs. Over twice as many in some cases. The reason for the increased number of branch misses is due to the increased number of conditionals that the corresponding row algorithm executes during each step of the cycle chasing, accessing the additional *row_offsets*[ ] and *corresp_table*[ ] arrays and checking if the corresponding row table needs to be updated.

The Corresponding Row algorithm accesses more arrays than Saad. At each step of the cycle chasing, Saad (Algorithm 3.3) will access 4 different arrays: the *non_zeros*[ ], *col_indexes*[ ], *tmp_row_indexes*[ ] and *new_row_ptrs*[ ] arrays. The corresponding row algorithm (Algorithm 5.3)

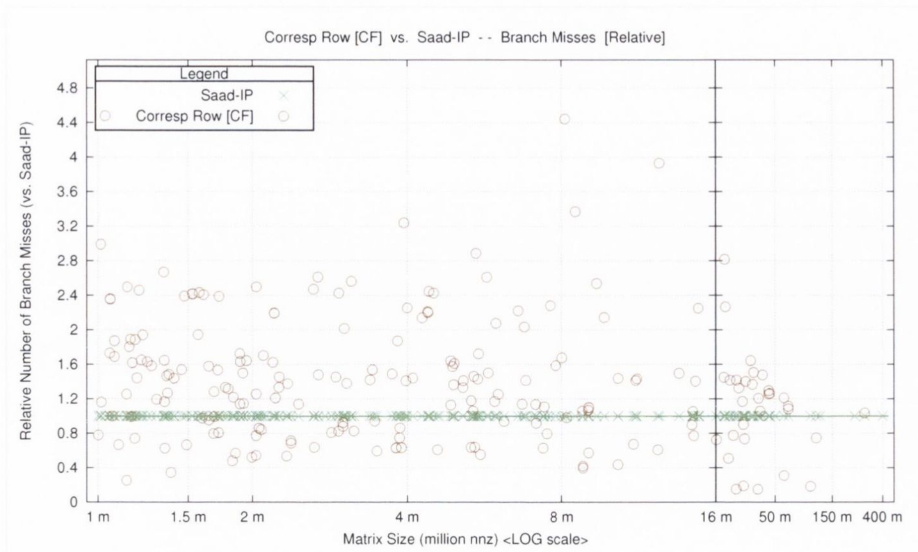


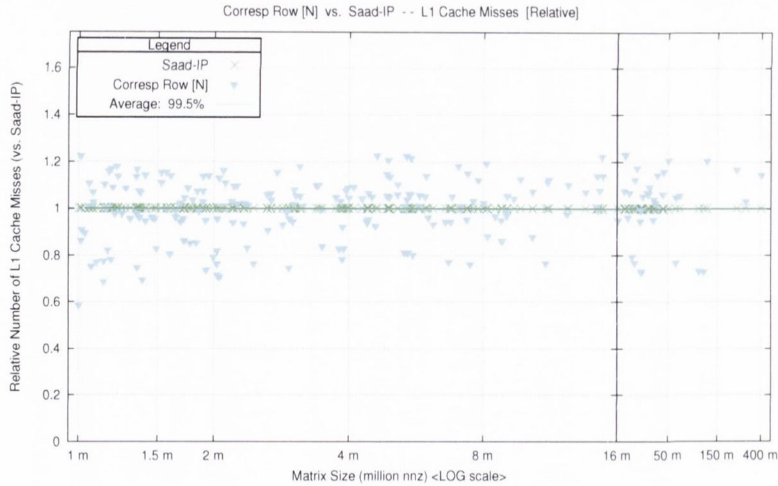
Figure 5.4: Number of Branch Mispredictions of the Cache Friendly Corresponding Row Algorithm when performing the cycle chasing transpose compared to that of Saad. The corresponding Row algorithm has a much higher proportion of branch misspredations than Saad because it performs more operations during each step of the cycle chasing with more conditional control flow.

will access 6 arrays at each step: $non_zeros[\,]$, $col_indexes[\,]$, $new_row_ptrs[\,]$, $row_offsets[\,]$, $corresp_table[\,]$ and $old_row_ptrs[\,]$. These extra accesses will cause more lookups, more cache misses and more branch mis-predictions. Despite this increased complexity, the reduced memory of the corresponding row algorithm leads to improved performance for the majority of inputs.

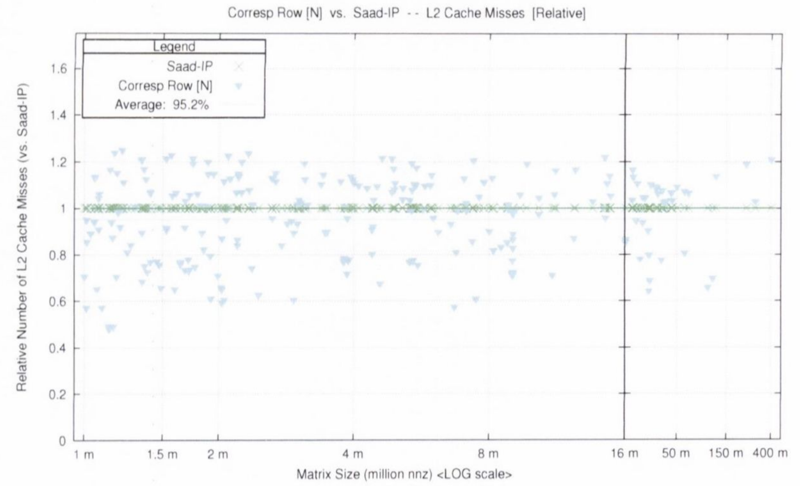
In the following sections we will analyse how the reduced memory overhead of the corresponding row implementations means that they make more efficient use of the caches and TLB (Translation Lookaside Buffer). Which translates into improved execution time.

5.9.3 Normal Corresponding Row Performance Evaluation

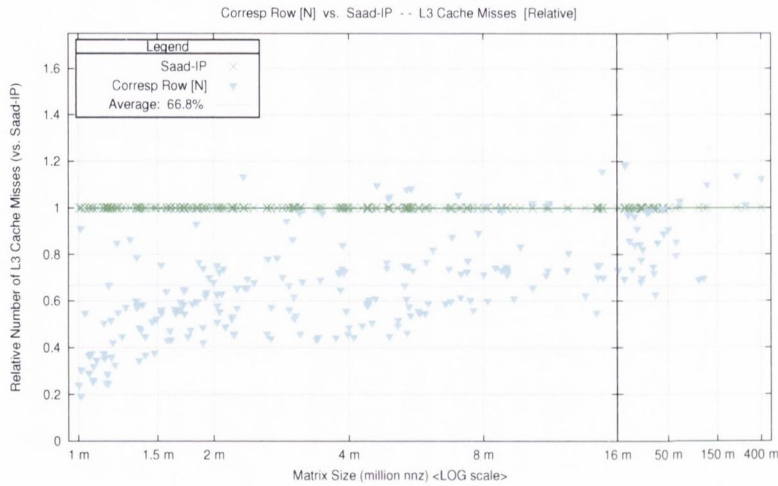
The four graphs in Figure 5.5 show the performance of the three L1, L2 and L3 caches and the TLB for the Normal Corresponding Row Algorithm



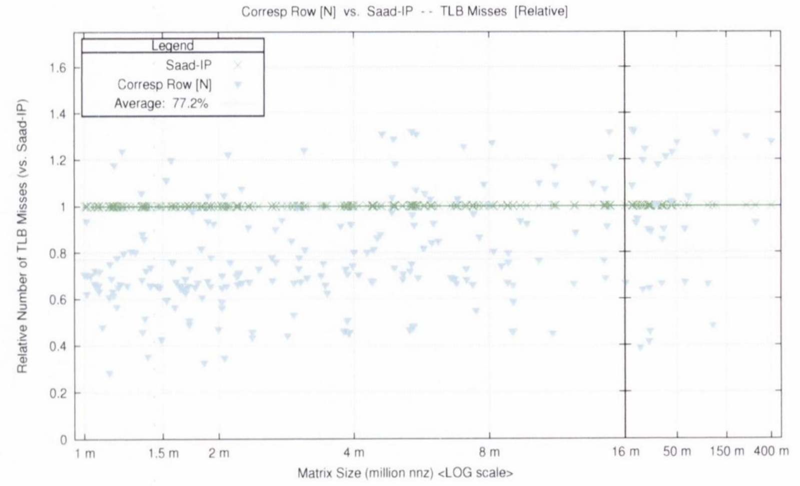
(a) L1 Cache Misses - 100%



(b) L2 Cache Misses - 95%



(c) L3 Cache Misses - 67%



(d) TLB Misses - 77%

Figure 5.5: Normal Corresponding Row Cache Performance

compared to the performance of the Saad algorithm. Figure 5.5 (a) shows the number of L1 Cache Misses compared to those of Saad while performing the full transpose algorithm. Figure 5.5 (b) shows the relative number of L2 Cache Misses. Figure 5.5 (c) shows the relative number of L3 Cache Misses. And Figure 5.5 (d) shows the relative number of TLB Misses encountered by the algorithm.

From these graphs we see that the L1 and L2 cache performance of the Normal Corresponding Row algorithm is broadly similar to that of Saad, however the relative measurements are quite scattered. On average the two algorithms actually have the same number of L1 cache misses as each other with the Normal Corresponding row having an average of 99.5% of Saad. The Corresponding Row algorithm has marginally better L2 cache performance with just 95% of the L2 cache misses as Saad. This is to be expected. As discussed above, although the Corresponding Row algorithm has a lower memory overhead than Saad, which should lead to a reduction in cache misses, the algorithm is accessing 6 different arrays at each step of the cycle chasing compared to the 4 accessed by Saad.

The L3 and TLB graphs in Figure 5.5 (c) and (d) tell a different story. There is a very clear distinction in the L3 cache performance of the Normal Corresponding Row compared to Saad. The Corresponding Row has 30%-60% less L3 cache misses than Saad for the majority of inputs. On average the Normal Corresponding Row algorithm incurs just 67% of the L3 cache misses of Saad. As discussed below, the reason for the reduced L3 and TLB misses is due to the reduced memory overhead of the corresponding row algorithm (21% of Saad).

There is also an indication from Figure 5.5 (c) that the Normal Corresponding Row has an even greater reduction in L3 misses for the smaller matrices, those matrices with 2 million non-zeros or less. This is to be expected as the L3 cache is very large and these smaller matrices are of comparable size. The machine these experiments were performed on has an L3 cache of 18 MiB. A matrix with 2,000,000 non-zero values will require approx 23 MiB to store in the CSR sparse format. Assuming $n \ll nnz$ then the corresponding row algorithm will only add perhaps a few hundred

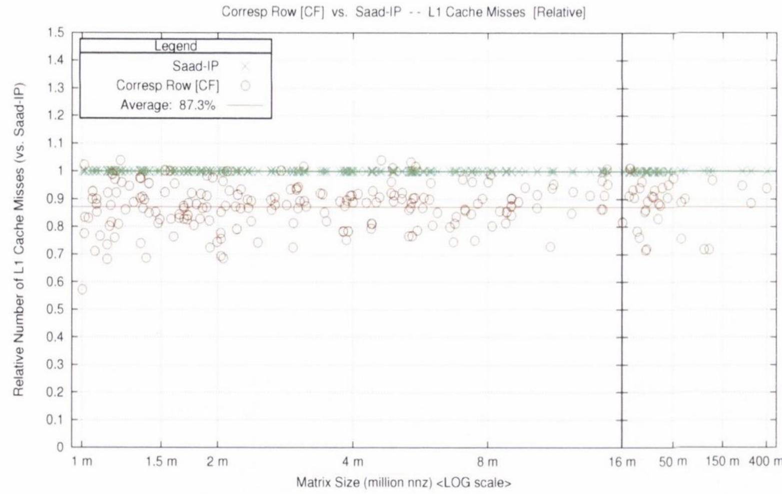
KiB on top of that. Thus for matrices with less than approx 2 million or 4 million non-zero values, a large proportion of the matrix will remain in the L3 cache — depending on replacement policy.

The measurements for the number of TLB misses shown in Figure 5.5 (d) are somewhat more scattered than those of L3, however there is still a very clear trend towards Corresponding Row with a large proportion of inputs having 20% - 40% fewer TLB misses. On average the Normal Corresponding Row incurs 77% of the TLB misses of Saad.

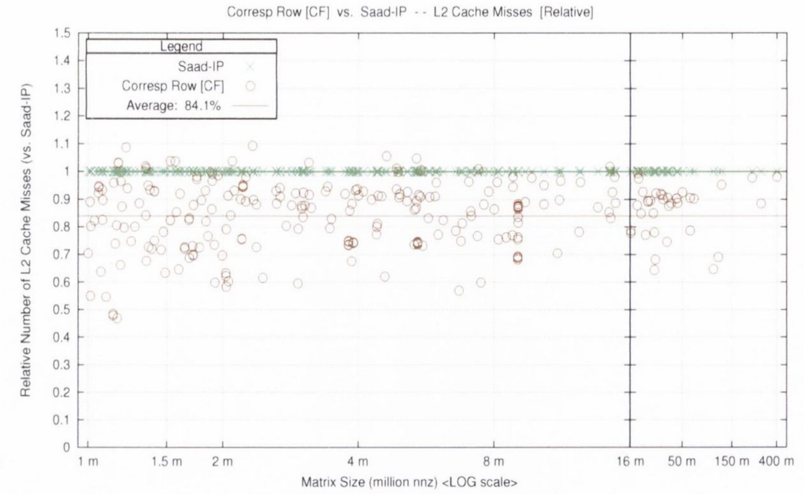
The graphs in Figure 5.5 show that although the Normal Corresponding Row algorithm does not improve on L1 and L2 cache efficiency compared to Saad, it does however improve on the L3 cache and TLB efficiency over Saad which results in the improved performance as shown in Figure 5.2. The reason for the improved L3 cache and TLB efficiency is due to the reduced memory overhead of the algorithm. Corresponding row uses three arrays of size $\Theta(n)$ compared to the $\Theta(nnz)$ array required by Saad.

Key to this is the size of the total working set. That is, the total amount of memory in use by the algorithm while performing the transpose. This includes the memory overhead of the algorithm and the memory required to store the matrix in memory. For the 259 matrices in our test suite, corresponding row has a working set size which varies between 75%-100% of that of Saad. The "average of the percentages" shows that the average working set of Corresponding Row is 80% of that of Saad. This is still a very large working set, however a reduction of up to 25% of memory usage is significant. Particularly for the larger matrices it could represent hundreds of megabytes of memory.

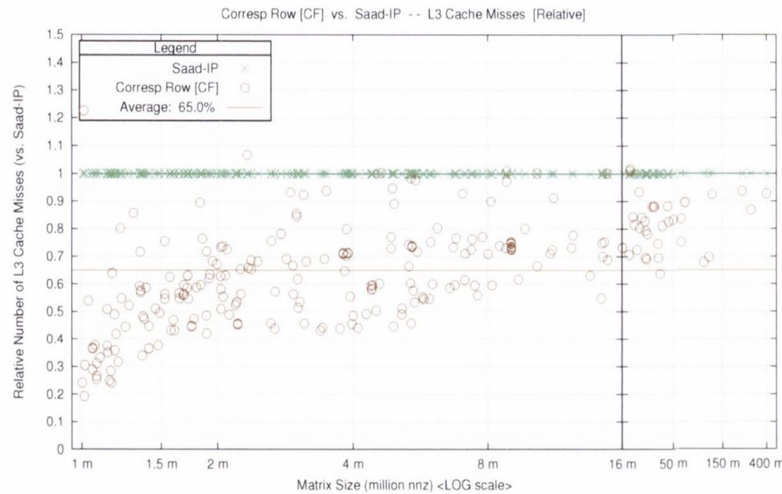
This reduction in working set size is the reason for the improved L3 cache and TLB performance and thus for the improved performance seen in Figure 5.2.



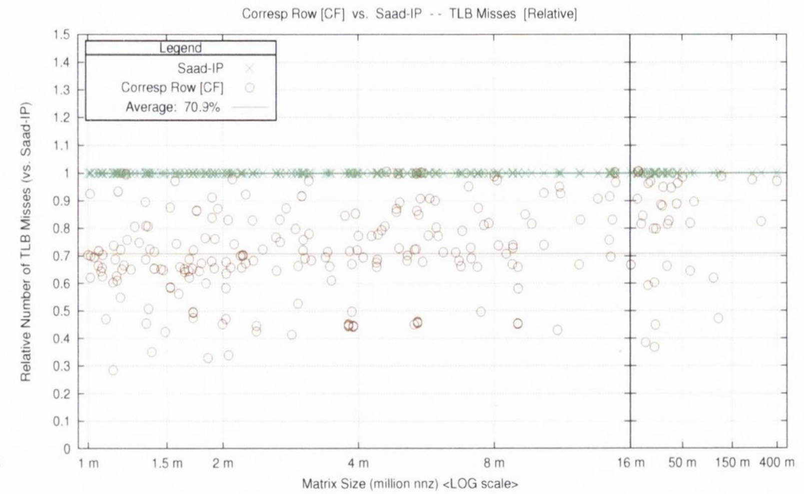
(a) L1 Cache Misses - 87%



(b) L2 Cache Misses - 84%



(c) L3 Cache Misses - 65%



(d) TLB Misses - 71%

Figure 5.6: `CF`: Corresponding Row: Cache Performance

5.9.4 Performance Evaluation of Cache-Friendly Algorithm

Cache and TLB performance of the Cache Friendly Corresponding Row algorithm as outlined in Section 5.6 is shown in comparison to the Saad algorithm in Figure 5.6. Again 5.6(a) shows the relative number of L1 misses incurred by the two algorithms, (b) shows the number of L2 misses, (c) the number of L3 misses, and (d) shows the relative number of TLB misses of the two algorithms incurred while transposing the 259 matrices in our test suite.

It is very informative to compare these four graphs to the four graphs in Figure 5.5 which show the cache and TLB performance of the Normal Corresponding Row algorithm implementation which uses two separate *row_offsets*[ ] and *corresp_table*[ ] arrays. Comparing Figure 5.6(a) with Figure 5.5(a) and Figure 5.6(b) with Figure 5.5(b) we can see the benefit of combining the two arrays in terms of reduced L1 and L2 cache misses. The Cache friendly Corresponding Row algorithm produces, on average 87% of the L1 cache misses and 84% of the L2 of the Saad algorithm. A reduction of 16% and 11% respectively compared to the Normal Corresponding Row.

The Cache Friendly algorithm has less of an impact on the L3 cache and TLB misses. There is still a slight decrease of 2% to 65% of the L3 cache misses and a reduction of 6% to 71% on the number of TLB misses compared to Saad. Once again we can see from Figure 5.6(c) that there is an extra decrease in L3 cache misses for matrices smaller than 2 million non-zero values. As discussed above, these smaller matrices are of a size which require a similar amount of memory to that which is available in the L3 cache. As such, as we will see below, there are very few L3 misses when transposing these matrices so even small reductions in memory size can make a big impact here.

It is clear from a cursory glance at the graphs in Figure 5.6 that the Cache Friendly Corresponding Row algorithm is generally much more efficient in terms of the L1, L2 and L3 caches and TLB usage. This shows why the algorithm performs well in Figure 5.3 where the algorithm performs

the transpose of the 259 matrices in, on average, 90% of the time of Saad.

Although the Cache Friendly algorithm has better L1 and L2 reuse, this (along with the small improvement to L3 and TLB) only results in an average reduction of 2% in execution time from 92% to 90% compared to Saad. It would appear that the L3 and TLB performance (along with some other factors) have a larger impact on performance than the L1 and L2 caches do. As discussed in Section 5.9.3, the reduction in overall memory usage (working set) of the Corresponding Row algorithm results in better L3 and TLB reuse compared to Saad, thus reducing its execution time.

5.9.5 Summary of Normal and Cache-Friendly Evaluation

As we have seen from the last two sections, both implementations of the Corresponding Row algorithm show a marked improvement in execution time compared to the Saad algorithm while only using, on average, 21% of the memory of Saad. The array interleaving of the Cache Friendly implementation does give it a slight performance advantage, as such, we will just be using the Cache Friendly implementation of the algorithm for the remaining experiments and discussions in this document.

5.10 Factors Influencing Cache Performance

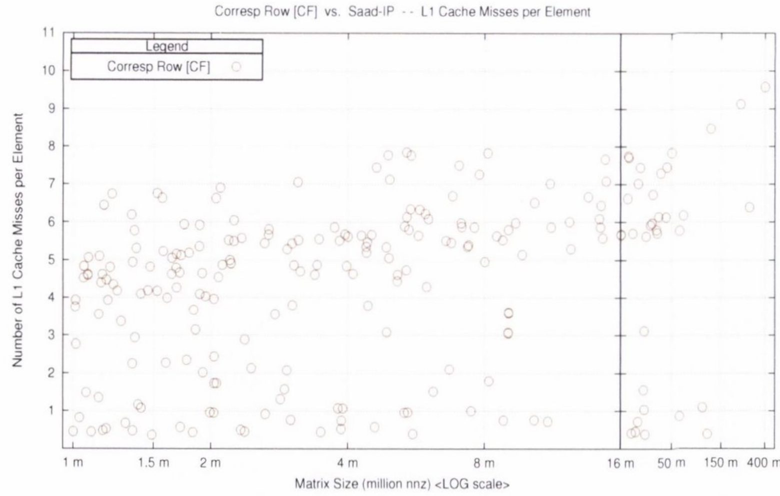
The graphs in Figure 5.6 in the previous section show the cache and TLB performance of the Cache Friendly Corresponding Row algorithm compared to Saad. These graphs are good for comparisons, however they do not give a true representation of the actual numbers of hits and misses of the algorithm. Showing the actual numbers of cache hits/misses is difficult due to the enormous range of matrix sizes and hence measurements. Instead we can show, for example, the number of L1 cache misses the algorithm encounters per non-zero element in the matrix. We do this by taking the total number of L1 misses the algorithm incurs during the transpose and divide that by the number of non-zeros (nnz) in the matrix. Thus

normalizing the number of L1 misses compared to the size of the matrix.

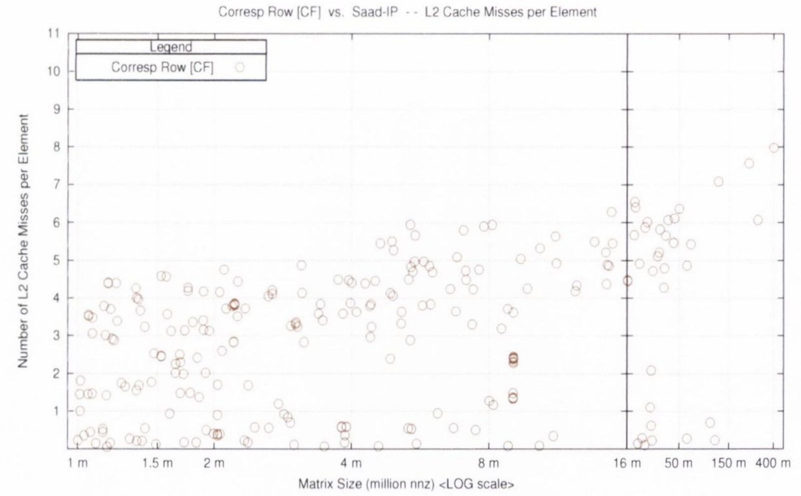
Figure 5.7 thus shows the number of L1, L2, L3 and TLB misses per non-zero element in each matrix incurred by the Cache Friendly Corresponding Row Algorithm while transposing that matrix. The cache and TLB performance numbers shown in all these graphs are the (normalized per element) number of misses incurred during the entire transpose operation, including building the work and lookup arrays and the cycle chasing transpose.

Figure 5.7(a) shows that there is a large number of L1 cache misses for most of the matrices with the majority of inputs causing between 4 and 8 L1 cache misses per non-zero element in the matrix. This is a very large number of cache misses. All of the in-place cycle-chasing transpose algorithms suffer from poor cache and TLB performance, this is because numerous arrays are accessed sequentially and randomly with very little reuse. For example, the corresponding row algorithm scans the $\Theta(nnz)$ size $col_indexes[\,]$ array to build the $new_row_ptrs[\,]$ and $row_offsets[\,]$ arrays, then scans both the $new_row_ptrs[\,]$ and $old_row_ptrs[\,]$ arrays to build the $corresp_table[\,]$. (The $corresp_table[\,]$ and $row_offsets[\,]$ are of course combined in the cache friendly implementation). The algorithm then randomly jumps around the $col_indexes[\,]$ and $non_zeros[\,]$ arrays while chasing cycles and also semi-randomly accesses the $new_row_ptrs[\,]$ $old_row_ptrs[\,]$, $row_offsets[\,]$ and $corresp_table[\,]$ arrays at each jump. This all results in a large number of cache misses.

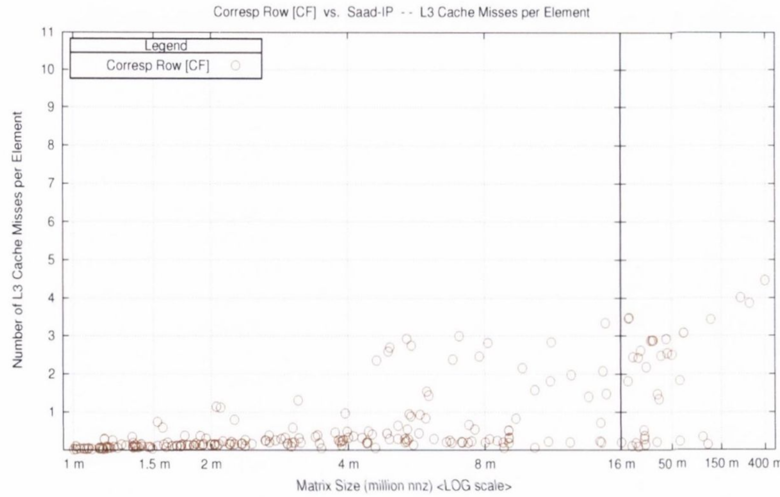
There is a very wide scattering of measurements of L1 cache misses for the Cache-Friendly Corresponding Row algorithm in Figure 5.7(a) with no clear trend apparent. There is a slight suggestion of an increase in L1 misses as matrix size increases from left to right, however there is no obvious correspondence. There are also a number of matrices (of a wide range of sizes) which have inexplicably low numbers of L1 cache misses with many having less than 1 cache miss per non-zero matrix element. A surprising result given the complexity of the algorithm described in the above paragraph. We will investigate these matrices further, examining the performance of the algorithms while transposing them in the following



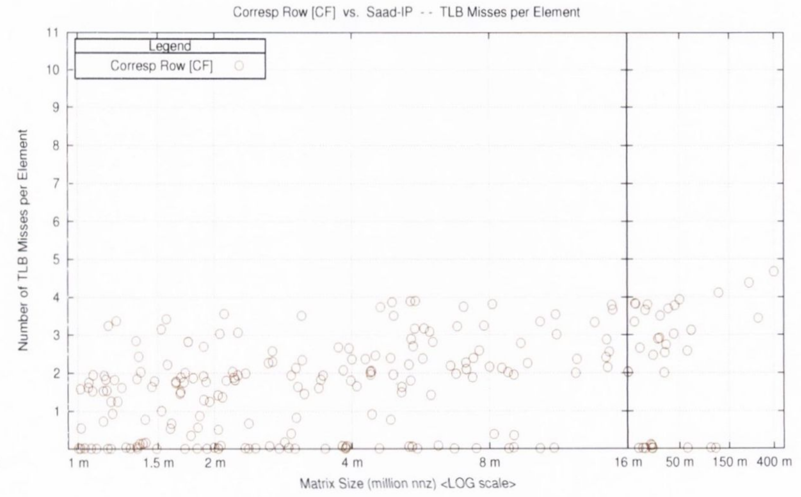
(a) L1 Cache Misses per Element



(b) L2 Cache Misses per Element



(c) L3 Cache Misses per Element



(d) TLB Misses per Element

Figure 5.7: `CF`: Corresponding Row: Cache Performance per Element

section and in Section 7.4 in Chapter 7.

The L2 Cache Misses of the Cache Friendly Corresponding Row are shown in Figure 5.7(b). Again there is a very wide scattering of measurements with most inputs incurring between 3 and 6 L2 cache misses per non-zero element in the matrix. There is a slightly stronger suggestion here of an increase in L2 misses as matrix size increases from left to right but still not quite a trend. Again there are a large number of matrices (of a wide range of sizes) which have very low numbers of L2 cache misses, below 1 or even $1/2$ of a cache miss per non-zero element which is very low compared to other matrices of comparable size.

The L3 cache misses of the Cache Friendly Corresponding Row algorithm are shown in Figure 5.7(c). Here we see that there is almost a trend in L3 misses increasing in proportion to increases in matrix size. Certainly for the smaller matrices there are very few (or even zero) L3 misses per non-zero element. This is not surprising given the size of the L3 cache as discussed above.

The number of TLB misses per non-zero element in the matrix are shown in Figure 5.7(d). There is a very wide scattering of TLB misses showing that the in-place cycle-chasing transpose is not very TLB efficient. There is no significant trend in the graph apart from the slight increase in TLB misses as matrix size increases from left to right. Most matrix inputs incur between 1 and 4 TLB misses per non-zero element. Again there are a certain number of matrices which have apparently inexplicably good TLB performance having zero, or close to zero TLB misses per non-zero element in the matrix.

For the graphs in Figure 5.7 we divided the total number of Cache/TLB misses by the total number of non-zeros (nnz) in the matrix. This removed the dominating influence of the huge matrix size from the measurements. We can still however see that there is a strong correlation between matrix size and cache performance, particularly for the L3 cache and to a lesses extent the L2 cache.

There are however other factors which influence the performance of the transpose algorithms.

5.11 Cycle Length and Cache Performance

In the last section (5.10) we examined the cache and TLB performance per non-zero element in the matrix of the Cache Friendly Corresponding Row algorithm. When we accounted for the major influence of the matrix size by looking at cache performance per element we found that there was still a certain amount of influence of the matrix size on cache performance, particularly for the L3 cache. However there must be other factors which influence cache and indeed execution time.

Deeper analysis of the operation of the Cache Friendly Corresponding Row algorithm, examining metrics and counters, shows that there is something else which has a greater relative influence on the cache/TLB performance of the Cache Friendly Corresponding Row transpose algorithm. Intuitively, the length of the chains we process during the cycle chasing transpose will have an influence on the cache performance, and hence execution time, of the cycle-chasing algorithm. If we are chasing a short chain, then after just a few jumps we will quickly return to the starting row, which will hopefully still be in the cache. Subsequent chains may then jump to rows which were visited by the previous chain and which have a better chance of still being in the cache. With longer chains, every time we jump we are jumping to a new row which has less and less chance of being in the cache the more times we jump. What's more, when we finally finish this long chain and jump back to the starting row, it is likely to have been flushed from the cache at this point.

Cycle chains can be extremely long, they are not just limited by the number of rows or columns in the matrix, they can be almost nnz elements long, transposing nearly the whole matrix in one chain. The longest cycle encountered in our test suite was while transposing the *nlpkkt200* matrix. A 16,240,000 $\times$ 16,240,000 square (triangular) matrix with 232,232,816 non-zero elements. There were just 22 cycles chased while transposing this matrix, the longest was 149,827,091 elements long, transposing 64.5% of the matrix in just one cycle. The longest cycle by relative length was when transposing the *af_shell10* matrix, a 1,508,065 $\times$ 1,508,065 matrix

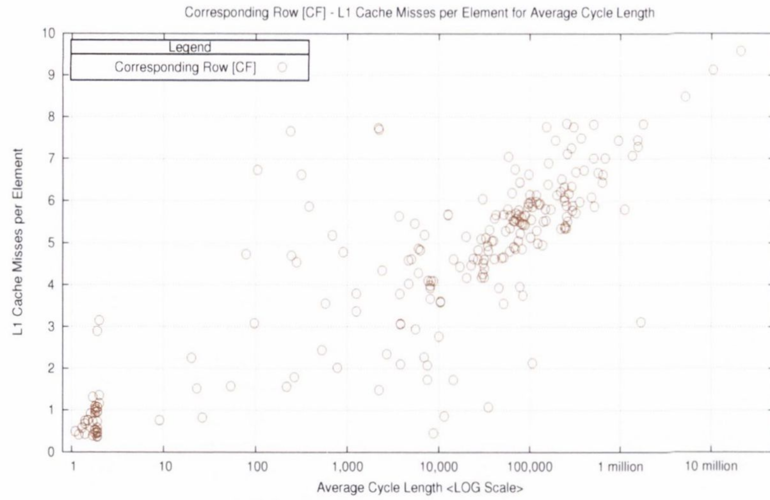
with 27,090,195 non-zero elements. A 27,047,251 element chain transposed 99.8% of this matrix in just one cycle.

Figure 5.8 shows the L1, L2, L3 and TLB misses per matrix non-zero element of the Cache Friendly Corresponding Row algorithm in terms of the *average cycle length*. This is the same data from Figure 5.7 just with the data ordered along the *x-axis* by the average cycle length rather than the matrix size (nnz).

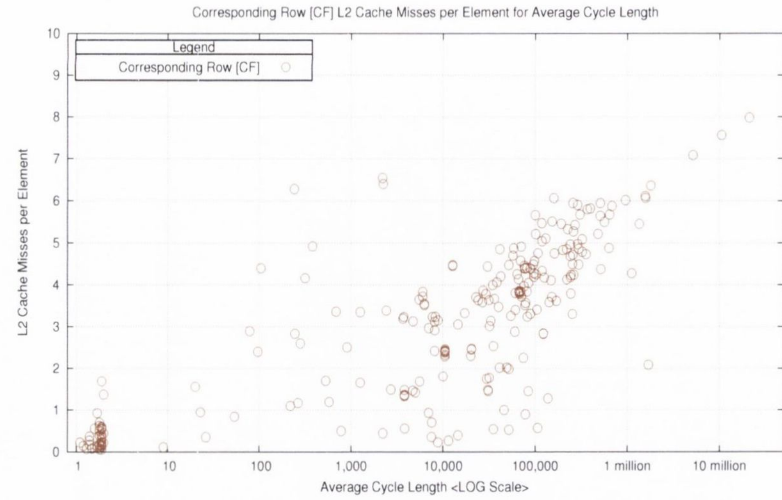
The graphs in Figure 5.8 show a clear trend in cache and TLB misses increasing as the average cycle length increases. The trend is certainly very evident in (a) L1 Cache misses, (b) L2 Cache Misses and (d) TLB Misses. In each of these three graphs there is somewhat of a scattering of measurement between the average cycle lengths of 100 and 10,000. This is possibly because at these levels, such average cycle lengths could be a very large proportion, or indeed very small proportion of the number of elements in the matrix, hence the variability. As such, some of these averages may not be representative of the cycles that are chased when transposing those particular matrices. Smaller matrices are also likely to have smaller average cycle lengths and conversely larger average lengths for larger matrices. Further analysis of standard deviations or perhaps a geometric mean might shed some more light, however, there is still a very clear trend and correlation between the cycle length and cache performance.

Again, the number of L3 cache misses shows that for smaller matrices, there are naturally a very small number of L3 cache misses. However above an average cycle length of about 50,000 elements, the number of L3 cache misses starts to increase as the average cycle length increases.

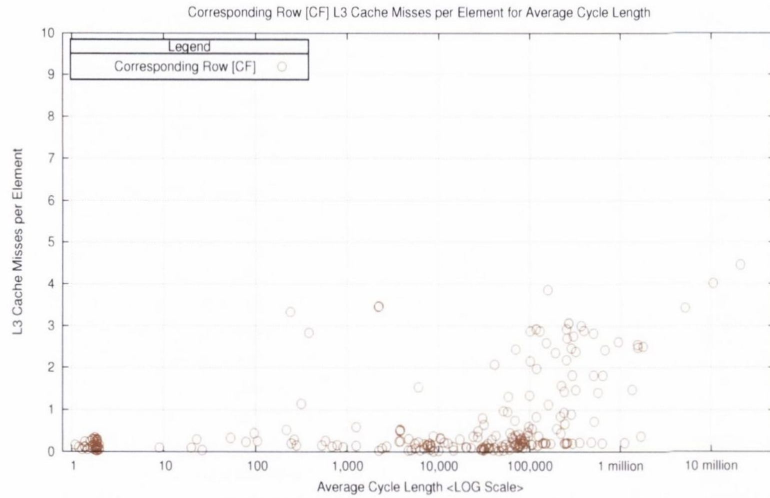
Finally, in each of these graphs there is an anomaly which we have omitted to mention until now. To the left of each of the 5 graphs there is a cluster of matrices which all have a very small average cycle chain length between just 1 and 2 elements long and which also have very low numbers of cache and TLB misses. The majority of the matrices which we identified in the previous section as having unusually low number of Cache and TLB misses are now clustered in the lower left corner of each of the graphs (see also Figure 7.12.). The reason why these matrices have such short average



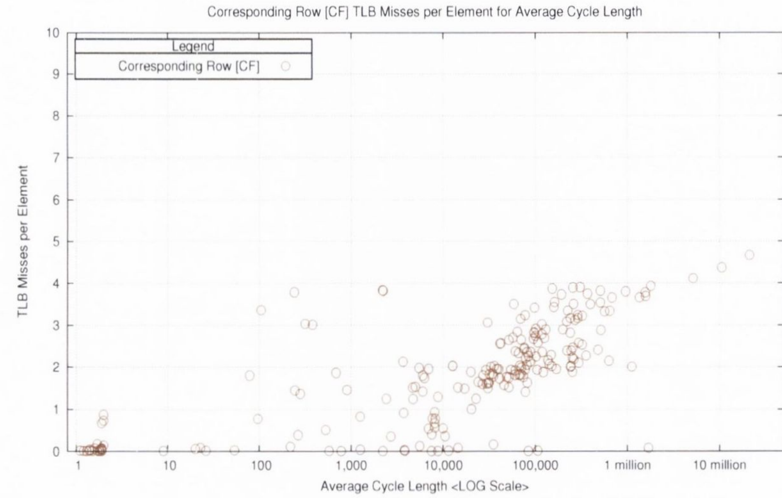
(a) L1 Cache Misses per Element



(b) L2 Cache Misses per Element



(c) L3 Cache Misses per Element



(d) TLB Misses per Element

Figure 5.8: `CF`: Corresponding Row: Cache Misses vs. Avg. Cycle Length

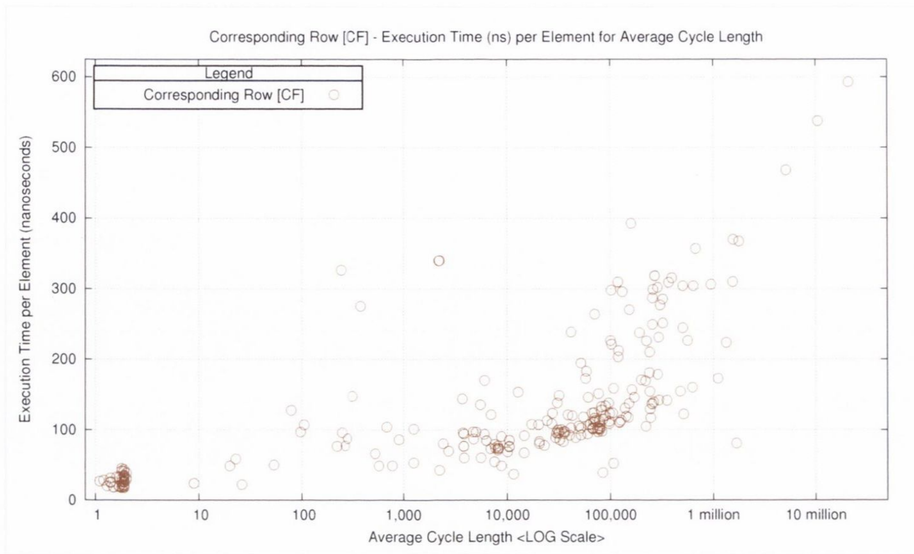


Figure 5.9: Execution Time of the *Cache-Friendly* Corresponding Row algorithm in nanoseconds per non-zero matrix element compared to the average cycle length when transposing the matrix. There is a clear correlation between average cycle length and execution time. Note that a nanosecond is the length of time of two processor cycles in the experimental machine (Clock speed 2GHz).

cycle lengths is because they are "*Structurally Symmetric*". They are not actually symmetric, but they have the layout of a symmetric matrix. When an element is moved, the element at the destination position always just needs to be moved back to the starting row, hence the longest cycle is 2 elements. This is the reason that these matrices have such good cache performance and can be transposed far more efficiently than matrices of comparable size. These structurally symmetric matrices will be discussed further in Section 7.4.

5.12 Summary

In this Chapter we introduced our Corresponding Row In-Place Sparse Matrix Transpose algorithm which transposes a sparse matrix in-place with only $\Theta(n)$ memory overhead while maintaining the $\Theta(nnz + n)$ time complexity of the existing sparse transpose algorithms. This is done by

using a Corresponding Row lookup table to find the *old_row_index* of an element as we move it during cycle-chasing. We show how we can search and update this table in amortized $\mathcal{O}(1)$ time.

The Corresponding Row algorithm reduces the memory overhead of the transpose to 21% of the Saad-IP algorithm. The Normal Corresponding Row algorithm transposes the 259 matrices in the test suite in 92% of the time of Saad on average and the Cache Friendly version transposes the matrices in 90% of the time of Saad.

We performed extensive experimental analysis of the performance of the algorithm using hardware counters to monitor cache and TLB misses. We can see how the cache performance in relation to the average cycle length as shown in Figure 5.8 influences the execution time in relation the average cycle length. Figure 5.9 shows the execution time of the Cache Friendly Corresponding Row algorithm in nanoseconds per non-zero element in the matrix as a function of the average cycle length of the chains chased while transposing the matrix. This graph shows that there is a clear correlation between the length of the cycles chased and the execution time of the transpose.

We have learned from the in-depth analysis and testing with hardware counters and the results demonstrated in Figure 5.8 and Figure 5.9 that short chains in the cycle chasing transpose lead to better cache performance and indeed reduced execution time. Chapter 6 addresses the question of how to use this knowledge to design a more efficient matrix transpose algorithm.

Chapter 6

HyperPartition Sparse Matrix Transpose

In Chapters 4 and 5 we introduced a number of algorithms based on our Generic In-Place Algorithm (4.1) which reduced the memory overhead of the in-place sparse matrix transpose from a space complexity of $\Theta(nnz)$ for the Saad algorithm and $\Theta(nnz + n)$ for Out-of-Place, to just $\Theta(n)$. For the largest matrices this reduced the extra workspace memory required during the transpose by over a Gigabyte (See Table A.3). These culminated in the corresponding row cycle chasing transpose which not only has a much lower memory overhead of 21% on average (see Figure 5.1) than Saad, the existing in-place algorithm, it also performs the transpose operation more efficiently taking just 90% of the execution time of Saad (Figure 5.3) for the sample input matrices.

As discussed previously, analysis with hardware counters demonstrated that the reason for the improved performance, despite being more complicated, is due to improved locality from smaller work arrays and hence reduction in cache and TLB misses. In Sections 5.10 and 5.11 we investigated the influence of the cycle length on the cache and execution time of the in-place cycle-chasing transpose with corresponding row lookup table. We found that there was a correlation between the average lengths of the cycles and the cache performance of the algorithm, and hence with the performance of the algorithm in terms of execution time.

Analysis shows that we could improve the performance of the in-place transpose algorithm if we could reduce the number of cache and TLB misses occurring. Reducing the length of the cycles would improve locality and improve cache and TLB performance.

This Chapter introduces our new Sparse matrix storage format: The HyperPartition, (or HypCSR) structure. The HyperPartition structure allows us to shorten the length of the cycles while performing the in-place

cycle-chasing transpose. This gives improved cache reuse and reduced execution time while also drastically reducing the memory overhead of the algorithm. We discuss how to perform the in-place sparse transpose with reduced memory overhead and improved execution time by converting to the HyperPartition format, performing the HyperPartition transpose in place, and then converting the matrix back to the standard CSR format.

6.1 The HyperPartition Sparse Matrix Format

One way of reducing the cycle length would be to group rows together into blocks of rows, or "*HyperPartitions*" of rows. We can then split the cycle chasing algorithm into two parts. The first part moves each element (if required) between HyperPartitions [blocks of rows] to its correct HyperPartition, rather than moving elements between individual rows. In the second part of the algorithm we take each individual HyperPartition in turn and move elements within that HyperPartition to their correct row in the HyperPartition and correct position within that row. This second part of the HyperPartition transpose can be combined with the Phase-II sorting phase of the in-place transpose algorithm for efficiency.

Performing the transpose in this manner could increase the number of times an element is moved. Elements are already moved multiple times during the corresponding row in-place transpose: Once during the cycle-chasing and possibly a number of times again during the sorting phase. Improving cache performance at the cost of increased copies/moves of elements is a common trade-off in cache efficient algorithm design. If the HyperPartitions are big enough this would greatly reduce the number of cache misses in four ways:

1. Large HyperPartitions mean that there is a greater chance that an element does not have to be moved.

2. Large HyperPartitions mean fewer HyperPartitions, which means

fewer destinations for an element to move to. Therefore it is more likely that the destination HyperPartition (or part of it) is in the cache.

3. These both lead to shorter cycles which means we come to the end of a cycle quicker and then return to the next element in the original row that needs to be moved, which is more likely to still be in the cache.

4. The second phase of the HyperPartition algorithm operates on only a single HyperPartition, and only reads/writes/moves elements within that HyperPartition. This means that we are more likely to get cache hits and it also opens up the possibility of performing this step in parallel.

6.1.1 Grouping Rows

The question is how to represent the HyperPartition. We could require the user to convert the CSR matrix to a hierarchical type structure which would allow us to group rows into blocks or HyperPartitions. However, as outlined in our aims we want our transpose algorithms to take the standard CSR/CSC format matrices without requiring the user to modify the structure of their matrix. The cost of conversion would not be worth it for an $\mathcal{O}(nnz)$ operation such as transpose. Indeed, copying the matrix to another internal structure would be inefficient and use additional memory which we are trying to avoid.

We could use an additional array of order $\mathcal{O}(^n/_k)$ to group rows together, however after moving an element to a new HyperPartition we somehow need to record the row within that HyperPartition that the element belongs to. In order to store this data for every element we would need an extra array of order $\mathcal{O}(nnz)$. We would prefer not to allocate any more additional memory than is required and we certainly do not want to push our space complexity back to $\mathcal{O}(nnz)$.

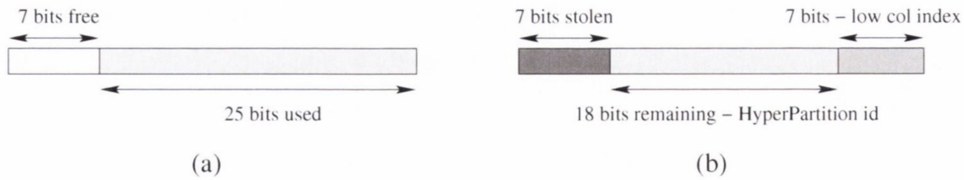


Figure 6.1: Stealing bits from the 32 *bit* integer 27,993,600

6.1.2 Unused Data in CSR Sparse Matrix Format

There is another option. There is in fact some additional unused data within the standard CSR and CSC structures which we may be able to exploit. We know that all the bits in the $non_zeros[\,]$ array are unavailable as they are all being used to store a "double" (or similar) variable for the value of each element. However, there is also the $col_indexes[\,]$ array which is typically an array of integers. In most current implementations an array of 32-bit integers is used. The largest (by n) matrix in our set of sample matrices from the Florida Sparse Matrix Collection (see Section 3.6.2) is the *Schenk/nlpkkt240* matrix. *nlpkkt240* has 27,993,600 rows, 27,993,600 columns and 401,232,976 Non-zeros. As such, the largest value stored in the column indexes array will be 27,993,600 and given that $\lceil log_e(27{,}993{,}600) \rceil = 25$ and that $2^{25} = 33{,}554{,}432$, it therefore only takes 25 bits to store all the possible row indexes. This means that there are *at least* $32 - 25 = 7$ bits (assuming unsigned integer) which are left unused in every one of the nnz indexes in the column indexes array.

We can see this in Figure 6.1(a). The column and row indexes require 25 bits to represent all their possible values, leaving seven bits unused in the 32-bit integer. In Figure 6.1(b) we could choose to steal all of these TOP seven bits to store the row id of the element within the HyperPartition. After an element is moved during transpose the bottom seven bits will become the column index (or new row index) of the element within the partition. This leaves the 18 bits in the middle which are the HyperPartition id of the element. (The HyperPartition id is accessed by masking the top and bottom seven bits and right shifting - This is described in detail later).

With 18 bits for the HyperPartition id, we therefore have $2^{18} = 262{,}144$ HyperPartitions.

By exploiting these unused bits in each of the entries in the *col_indexes*[ ] array we can record which row within each HyperPartition each element is in. For the largest matrix *nlpkkt240*, we can steal seven bits, therefore for this matrix we can have $x = 2^7 = 128$ rows per HyperPartition. Thus reducing the number of locations an element can be moved to from 27,000,000 to just 211,000, a significant reduction in the number of places that a non-zero value can be moved to. Other matrices have an even larger reduction and for matrices with fewer rows we could also have even larger HyperPartitions. The second largest matrix in the test suite is *Fluorem/HV15R* which is a $2{,}017{,}169 \times 2{,}017{,}169$ matrix with 283,073,458 non-zeros. In this matrix it is possible to steal 11 bits meaning we can put 2,048 rows in each HyperPartition and reduce the number of locations an element can be moved to from ~2 million to just ~985 and also the possibility that any jump would end a cycle from ~283 million to ~138 thousand. Stealing 11 bits from *HV15R* reduces the average cycle length from 159,567.3 to 1,177.5 and the longest cycle from 20,982,531 to 7,627,015 which led to a reduction in total execution time of the full transpose (cycle chasing and QuickSort) from 115.6 seconds to 31 seconds (just 26.8% of the original execution time). The number of moves during the cycle chasing was reduced by only 0.05% from 283,073,458 to 282,908,847, a reduction of just 164,611, clearly showing that the main benefit is from improved locality.

6.1.3 The HyperPartition Structure

Figure 6.1 shows our matrix M in the CSR format again. Figure 6.2 shows the same matrix converted into the HyperPartition CSR format.

The *non_zeros*[ ] array remains unchanged as the elements remain in the same order within the matrix. The *row_ptrs*[ ] pointers array has been replaced with a *hyp_ptrs*[ ] array. In this small example we are stealing *one* ($b = 1$) bit giving two rows per HyperPartition. Therefore, in this representation there are three HyperPartitions which start at array locations

0, 5 and 10, respectively. The *col_indexes*[ ] array has been updated to also include the "*HyperPartition row number*" of each element within each HyperPartition.

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
old_row_ptrs	=	0_0		2_1			5_2		7_3			10_4		12_5			15
non_zeros	=	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
col_indexes	=	0	4	0	1	5	1	2	0	3	4	4	5	1	4	5	

Example 6.1: Matrix M in CSR representation

index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
old_hyp_ptrs	=	0_0					5_1					10_2					15
non_zeros	=	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
col_indexes	=	$_0$:0	$_0$:4	$_1$:0	$_1$:1	$_1$:5	$_0$:1	$_0$:2	$_1$:0	$_1$:3	$_1$:4	$_0$:4	$_0$:5	$_1$:1	$_1$:4	$_1$:5	

Example 6.2: Matrix M in HyperPartition CSR representation

The HyperPartition that an element is in is found in this case by taking the row index of the element and right shifting by $(b = 1)$ bit. The *HyperPartition row number* is found by taking (masking) the least significant bit of the element's row index. The HyperPartition row number for each element can be seen as the subscript before the colon (:) in the *col_indexes*[ ] entry for each element in Example 6.2. As only one bit is being stolen, the HyperPartition row number can only be 0 or 1. 0 for the first row in each HyperPartition and 1 for the second. In Example 6.2 the element at position 4 with value 'e' was in row 1 but is now in HyperPartition 0, so its column index has been updated to contain ($_1$:5) to indicate that it is in row 1 within HyperPartition 0 and has column index 5.

Figure 6.2 shows an example of how an element is converted to HyperPartition format. In this case using *two* places of decimal (base 10) for simplicity. The element has row index 28,362 and column index 19,079. The low row digits (62) are masked off from the row index and added to

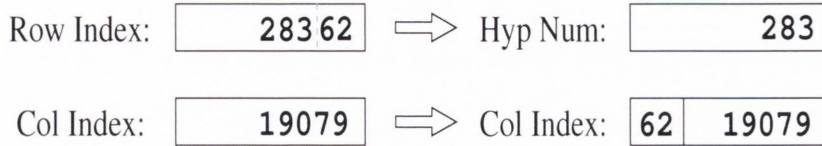


Figure 6.2: Converting an element to HyperPartition using 2 decimal places

the column index entry for the element in the high order digits. The HyperPartition number is found by right shifting the row index by two decimal places to give 283. Thus this element will be in row 62 in HyperPartition number 283.

An additional benefit of the HyperPartition sparse matrix format, as can be seen from Figure 6.6, is that there is a huge reduction in the memory usage of in-place transpose algorithm when using the HyperPartition format. This is because the $hyp_ptrs[\,]$ arrays are much smaller than the $row_ptrs[\,]$ arrays, proportional to the number of bits that we steal. Four arrays are needed for the HyperPartition transpose, thus the actual memory usage of the HyperPartition transpose is $\sim(\frac{4n}{2^{steal_bits}})$. This could be $\sim(\frac{4n}{256})$, $\sim(\frac{4n}{1024})$ or $\sim(\frac{4n}{4096})$ depending on the bits available in the matrix. This memory overhead compares very favourably to the $\sim(3n)$ usage of the corresponding row, the $\sim(nnz)$ usage of Saad and the $\sim(3nnz + n)$ usage of the Out-of-Place algorithm. Actual memory usages for the algorithms for a selection of matrices are shown in Table A.3.

6.1.4 Using the HyperPartition Format

The procedure for performing the sparse matrix transpose using the HyperPartition format is as follows:

a) Convert the matrix from CSR format to Hyperpartition (HypCSR) format.

b) Perform the HyperPartition cycle chasing algorithm to move elements to their correct HyperPartition.

c) Sort each of the HyperPartitions so the elements are in their correct row and correct row order.

d) Convert the matrix back to standard CSR format.

6.2 Converting to HyperPartition Format

The procedure for converting from CSR format to the HyperPartition CSR format [HypCSR] is shown in Algorithm 6.1. The algorithm sets up the matrix and a number of variables and arrays so the HyperPartition cycle chasing algorithm (6.2) can be used to transpose the matrix. The algorithm takes an extra input of the *steal_bits* parameter which is the number of bits we are stealing from the *col_indexes*[ ] array to use as the "*HyperPartition row number*". Based on this input of the *steal_bits* parameter, the algorithm first (Lines 2-9) calculates a number of integer variables, binary masks and shift offsets which will be used later to perform the bit manipulations necessary to massage the row and column indices into the HyperPartition format.

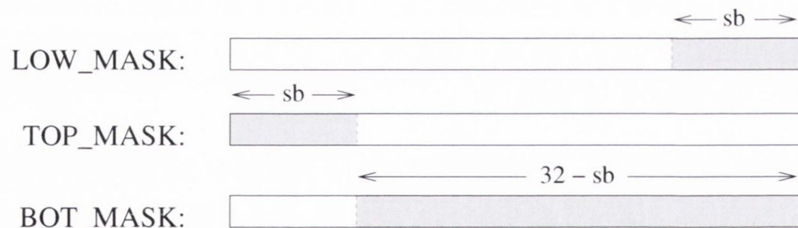


Figure 6.3: Bit Masks for converting to HyperPartition. $sb = steal_bits$

Figure 6.3 show which parts of the entry in *col_indexes*[ ] is isolated by each of the three bit masks LOW_MASK, TOP_MASK, BOT_MASK and shows the length $(32 - sb)$ of the *remain_bits* offset.

The variables, bit masks and offsets are as follows:

remain_bits – The number of bits remaining which are required to store the column index of each matrix entry. Note, for rectangular matrices we need to leave enough bits to store both the maximum row and column index, thus: $remain_bits \geq max(nrow_bits, ncol_bits)$.

LOW_MASK – The bit mask required to get the low order bits from the row index — these low bits become the row number of the element within the HyperPartition which we will place in the top part of the element's *col_index*[ ] entry.

BOT_MASK – The bit mask used to get the bottom part of the integer which contains the column index of the matrix entry.

TOP_MASK – The bit mask used to get the top part of the integer which contains the element's row number within the HyperPartition — the low order bits of the row index which have been right shifted to the top of the integer.

rows_per_hyp – The number of rows per HyperPartition = 2^{steal_bits}.

n_old_hyp – The number of HyperPartitions in the original input matrix.

n_new_hyp – The number of HyperPartitions in the transposed matrix.

Figure 6.4 shows where the different information is stored in the entry in the *col_indexes*[ ] array. The top '*sb*' bits (TOP_MASK) contain the old HyperPartition row (which row within the HyperPartition the element is in). The bottom '$32 - sb$' bits (BOT_MASK) contain the old column index. The lower '*sb*' bits (LOW_MASK) contain the new HyperPartition row number after the transpose. Finally the middle '$32 - sb - sb$' bits (BOT_MASK $\gg$ *remain_bits*) contain the new HyperPartition id the element is in.

Algorithm 6.1 then allocates the *old_hyp_ptrs*[ ] and *new_hyp_ptrs*[ ] arrays (Lines 10-11). The *old_hyp_ptrs*[ ] array contains pointers to the start of each of the current HyperPartitions. It is built on lines 13-15 by

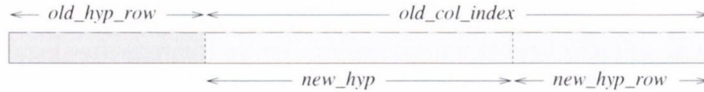


Figure 6.4: Where information is stored in the HyperPartition Format

taking every $n_old_hyp^{th}$ pointer in the $old_row_ptrs[\,]$ array. The lines 18-25 are the core of the algorithm which convert the matrix to HypCSR. The algorithm loops through each of the old rows in the CSR matrix. For each row it gets the low order bits by masking the row number. These bits will become the row number of the element within each HyperPartition so they are left-shifted to the top of the integer. The algorithm then loops through each of the elements in the old row. The new_hyp that this entry *should* be in is calculated by taking the column index of the entry and right shifting it by $steal_bits$. The new_hyp is used to count the number of elements in each of the new HyperPartitions by accumulating the count in $new_hyp_ptrs[\,]$. The row number is then added by adding the low order bits we isolated earlier to the top of the column index and storing the combined number back in the $col_indexes[\,]$ array. Finally, the cumulative sum of the $new_hyp_ptrs[\,]$ array is calculated on lines 27-29 in order to convert from counts to new HyperPartition array pointers.

The algorithm for converting from CSR to HypCSR format is a $\Theta(nnz + n)$ operation. This is the same as the initialization of all the other in-place algorithms. The only difference is that during the HyperPartition initialization, the $col_indexes[\,]$ array is updated to contain the low order row bits in the top of each entry.

6.3 HyperPartition Cycle-Chasing Transpose

The next step is performing the cycle chasing algorithm on the matrix stored in HyperPartition format — moving elements to their correct HyperPartition. The procedure is just like the normal cycle chasing algorithm except that we need to use a number of bit manipulations to get the column index and HyperPartition row number of the element from the $col_indexes[\,]$

ALGORITHM 6.1: Convert from CSR format to HyperPartition format

Input: Matrix M as in Data Structure 3.1, *steal_bits*
Output: Matrix in HyperPartition CSR format

```
1  /* Calculate masks and offsets */
2  remain_bits ← (32 − steal_bits);                    /* Bits remaining: 32 − 6 = 24 */
3  LOW_MASK ← (0xFFFFFFFF >> remain_bits);             /* Mask Low order bits:
   0x0000003f */
4  TOP_MASK ← (LOW_MASK << remain_bits);               /* Mask High Order bits:
   0xfc000000 */
5  BOT_MASK ← (0xFFFFFFFF ⊕ TOP_MASK);                 /* Mask Bottom of index:
   0x03ffffff */
6  /* Allocate HyperPartition Pointers */
7  rows_per_hyp ← pow(2, steal_bits);                  /* Num rows per HyperPartition
   = 2^stolen_bits */
8  n_old_hyp ← old_nrows % rows_per_hyp;               /* Num HyperPartitions in Original
   Matrix, M */
9  n_new_hyp ← new_nrows % rows_per_hyp;               /* Num HyperPartitions in Transpose
   M^T */
10 Allocate: old_hyp_ptrs[n_old_hyp + 1];
11 Allocate: new_hyp_ptrs[n_new_hyp + 1];
12 /* Build Old HyperPartition Pointers using every rows_per_hyp'th index in old_row_ptrs[] */
13 for ( 0 ≤ i < n_old_hyp ) do
14 |   old_hyp_ptrs[i] ← old_row_ptrs[i * rows_per_hyp];
15 end
16 old_hyp_ptrs[n_old_hyp] ← old_row_ptrs[old_nrows];
17 /* Put the low 'k' bits of the row index in the top 'k' bits of col_indexes[] array */
18 /* Also count the number of elements per HyperPartition to build new_hyp_ptrs[] */
19 for ( 0 ≤ row < old_nrows ) do
20 |   low_row_bits ← ((row & LOW_MASK) << remain_bits);
21 |   for ( p ← old_row_ptrs[row] ; p < old_row_ptrs[row + 1] ; p ← p + 1 ) do
22 |   |   new_hyp ← ((col_indexes[p] >> steal_bits) + 1);
23 |   |   col_indexes[p] ← (col_indexes[p] || low_row_bits);
24 |   |   new_hyp_ptrs[new_hyp] ← new_hyp_ptrs[new_hyp] + 1;
25 |   end
26 end
27 /* Cumulative sum of hyp counts to create new_hyp_ptrs[] - Also set hyp_offsets[] */
28 for ( 1 ≤ hyp < n_new_hyp ) do
29 |   new_hyp_ptrs[hyp] ← new_hyp_ptrs[hyp] + new_hyp_ptrs[hyp − 1];
30 |   hyp_offsets[hyp] ← new_hyp_ptrs[hyp];
31 end
32 /* Build the Corresponding Row Lookup Table */
33 build_corresp_table(old_nrows, new_nrows, old_row_ptrs[], new_row_ptrs[]);
```

array. We also need to do some housekeeping when we move an element. We must swap the row and column indexes and the '*row number bits*' at

the top of the entry in the *col_indexes*[ ] array. Note: it is also necessary to update the *col_indexes*[ ] entry even if the element remains in the same HyperPartition.

The algorithm for performing the In-Place HyperPartition Cycle-Chasing Sparse Matrix Transpose is shown in Algorithm 6.2. The CSR to HypCSR conversion Algorithm (6.1) must be run first, then this cycle chasing algorithm continues. The cycle-chasing algorithm (6.2) re-uses the binary masks, offsets and HyperPartition arrays initialized during the conversion.

The HyperPartition cycle chasing starts (line 2) by looping through each of the new HyperPartitions just like the Corresponding Row in-place Algorithm (5.3) loops through rows. The algorithm loops through each of the elements in the new HyperPartition and copies the element into the "src" temporary location (Lines 6-11). The old column index is found by masking the entry for the element in the *col_indexes*[ ] array with BOT_MASK. As in the generic cycle chasing algorithm we look up the old HyperPartition that the current element resides in by searching the *corresponding HyperPartition* table using the current *new_hyp* as the key. We obtain *low_rows*, the old row's least significant low order bits by masking them from the top of the integer in *col_indexes*[ ] with TOP_MASK and right shifting them down by *remain_bits* to the bottom of the integer. *low_rows* is used as an intermediary step for clarity and to prevent line wrapping. The current old row of the element is calculated by right shifting the element's old HyperPartition number to the top of the integer and then adding *low_rows*. *target_hyp*, the new HyperPartition that this element *should* be in is calculated by taking *src_col* (which is the *new_row* the element should be in) and right shifting it by *steal_bits*.

We now have our triplet of information about the element; *src_val*, *src_row* and *src_col* along with *target_hyp*, the HyperPartition that the element should be moved to. The algorithm then starts chasing this element in 'src' as normal on line 12 until it finds an element that should be in *cur_hyp*, the new HyperPartition the algorithm started chasing from. Line 14 finds the destination location *dst_x* that it needs to move the element in 'src' to. On lines 15-19 the old element at that location (*dst_x*)

is copied into the 'dst' temporary location using the same bit manipulations as lines 6-11 to get the elements *value*, *row index* and *column index* triplet of information. On lines 21-23 the element in 'src' is copied to the arrays at position dst_x, and $row_offsets[\,]$ is updated to mark that this element has already been moved. Line 22 is important. When updating the $col_indeses[\,]$ array we need to swap the old column index with the old row index (aka: new col index) as always. However we also need to swap the low order bits from the old row index at the TOP of the integer with the low order bits from the old column index (aka: new row index) in order to know later which new row within the new HyperPartition the element needs to be in.

The algorithm then copies the *row*, *col* and *val* triplet from 'dst' into 'src' and calculates a new $target_hyp$ for this new element in src. The algorithm then loops back to line 12 chasing this new element in 'src' until it finds an element which should be in the cur_hyp we started from, at which point (lines 31-32) it copies that element in '*src*' back to position 'x' in the original cur_hyp where we started chasing the chain. $row_offsets[\,]$ is finally updated on line 33 to indicate that that element has been moved. The algorithm continues looping through each HyperPartition and each (unmoved) element in each HyperPartition until all elements have been moved to their correct HyperPartition.

6.4 Sorting HyperPartition after Cycle-Chasing

At this point the algorithm has performed the cycle chasing such that all the elements are in their correct 'new' HyperPartition as can be seen in Example 6.3. Elements are however not necessarily in their correct new rows within the HyperPartition, nor are they in the correct column order within these rows. In Figure 6.3, the elements in the first Hyperpartition (hyp 0) are actually all in their correct rows within the HyperPartition as can be seen by the subscripts in their $col_indexes[\,]$ entries, but they are not in the correct order within each of the rows. Elements in row 0 are (a,h,c) where they should be in the order (a,c,h). In row 1 elements (d,m,f) should be

in the order (d,f,m). In HyperPartition 2, elements are in the wrong rows. Element 'e' in row 0 should be in the second row in HyperPartition 2, and element 'n' in row 1 should be in the first row in the HyperPartition. The ordered elements in HyperPartition 2 should be (b,j,k,n) and (e,l,o).

The procedure from here is to take each individual HyperPartition and sort the elements of that HyperPartition into their correct rows and into the correct column index order within the rows. Due to the way we added the low order bits of the row number to the TOP of the integer in the *col_indexes*[ ] array we can actually just re-use the QuickSort/InsertionSort algorithm outlined in Section 4.5 to sort the HyperPartitions by simply passing the HyperPartition segment of the two arrays to the algorithm where previously we passed just the segments of each of the rows. Note: We are now using all the high order bits of the integers in *col_indexes*[ ] so it is important that the sorting algorithms treat the entries in the *col_indexes*[ ] array as unsigned integers. We will investigate optimizations to this sorting step further in Section 7.1 in the next Chapter.

Using both the row number and column index means that when we sort the two arrays based on the full integer values in the *col_indexes*[ ] array, then the sorting will arrange the elements into the correct rows in the HyperPartition and also into the correct column order within each row, at the same time.

The sorted HyperPartition matrix can be seen in Figure 6.4. All the Elements are now in their correct row and column index ordering within each HyperPartition.

6.5 Converting from HypCSR back to CSR

After sorting the HyperPartitions, we then have to convert the matrix from HypCSR, the HyperPartition CSR format, back to the standard CSR format. The procedure is shown in Algorithm 6.3, it involves going through each element in each new HyperPartition and removing the low order new row index bits from the element's *col_indexes*[ ] entry. This is done on line 15 by AND'ing the entry with BOT_MASK. We also need to count the

number of entries in each of the new rows in the transposed matrix in order to rebuild the $new_row_ptrs[\,]$ array for the transposed matrix. We do this by taking the low order row index bits we just removed (line 13) from the $col_indexes[\,]$ entry, right shifted to the bottom of the integer and adding it to the current HyperPartition number left shifted by $steal_bits$ to get the high order bits of the row index (lines 10, 13 and 15). The algorithm finally does a cumulative sum to turn the new row counts into $new_row_ptrs[\,]$.

For efficiency we combine the sorting and HypCSR to CSR conversion into the one procedure. After sorting, many of the elements in this HyperPartition are still likely to be in caches, so it is appropriate to then convert this HyperPartition in HypCSR format back into the standard CSR format before sorting another HyperPartition which would likely overwrite the cached data. As can be seen on line 8 of Algorithm 6.3, the algorithm goes through each HyperPartition and first calls the `HyperPartition_Sort()` routine on the appropriate segments of the $non_zeros[\,]$ and $col_indexes[\,]$ arrays. For now we are using the two-array QuickSort/InsertionSort outlined in Section 4.5, we revisit the sorting phase again in Section 7.1 in the next Chapter.

6.6 Heuristic: Choosing Number of Bits to Steal

The number of bits we steal in the HyperPartition cycle chasing algorithm influences the performance of the algorithm. In many cases, particularly for large matrices, stealing all the available bits seems like the best option. However for smaller matrices this could end up leaving us with a very small number of very large HyperPartitions. In the worst case we may end up with a single gigantic HyperPartition containing all the rows in the matrix. In this case the entire transpose operation would be done by just the sorting phase – sorting the entire matrix at once.

The only actual restriction on the number of bits we can steal is $need_bits$. Where $need_bits$ is the number of bits required to store $max(nrows, ncols)$.

i.e.

$$need_bits = \lceil log_2(max(nrows, ncols)) \rceil$$

There must be at least enough bits remaining ($remain_bits \geq need_bits$) in order to represent both the largest possible row index and the largest possible column index. Aside from this, we can steal anywhere between ($0 \leq steal_bits < (32 - need_bits)$). There is also little point in stealing any more than $need_bits$, if $need_bits = 12$ therefore there are $(32 - 12 = 20)$ bits available to steal however stealing this many would leave us with a single HyperPartition containing all the rows. The limits of the number of bits that are available to steal are straightforward:

$$\begin{aligned} 0 \leq\ & steal_bits \leq (32 - need_bits) \\ 0 \leq\ & steal_bits \leq need_bits \\ & need_bits = \lceil log_2(max(nrows, ncols)) \rceil \end{aligned}$$

However, actually knowing the best number to steal is more difficult. Experimental analysis showed that the number of stolen bits which produced the best performance differed from matrix to matrix with no clear consensus.

The number of bits we steal, in combination with the relative dimensions of the matrix ($nrows$,$ncols$,nnz) influence the size and number of Hyper-Partitions which in turn influences the relative performance of both the cycle chasing HyperPartition transpose (Phase-I) and the HyperPartition sorting (Phase-II). Pushing too much work into one phase or the other will impact on performance. There are a number of naïve approaches we could take to choosing the number of bits to steal. Some of the heuristics include:

Naïve Heuristics:

- Always steal the maximum number of bits
- Always steal the minimum number of bits
- Always steal (at least) 'x' bits
- Always steal (at most) 'x' bits

As discussed above, stealing all available bits is good for some (large) matrices but detrimental to others. Stealing the minimum (zero) number of bits is pointless. The other strategies of always stealing at least 'x' or at most 'x' run into the same problem that it would be good for some matrices but detrimental to others.

6.6.1 Remaining Bits Heuristic

The number of bits we steal influences the structure of the matrix in three ways;

1) The number of bits we steal determines the number of rows in each HyperPartition. The size of the HyperPartitions then depends on the average number of elements in each row which depends on the sparsity of the matrix.

2) The number of bits remaining ($remain_bits$) controls the number of HyperPartitions created as we put 2^{steal_bits} number of rows into each HyperPartition.

3) Therefore the number HyperPartitions created is $\sim 2^{remain_bits}$.

Balancing these two variables is important for good performance of the HyperPartition transpose. We can balance the number of and size of HyperPartitions with the following relation which we call the *Remaining Bits Heuristic*:

The Remaining Bits Heuristic:

$$remain_bits = (need_bits - steal_bits) \geq k$$

In other words:

Steal as many bits as possible but always leave at least 'k' need_bits remaining.

The advantage of this heuristic is that it is agnostic of matrix dimensions. It will keep a balance between the size of the HyperPartitions and number of HyperPartitions regardless of matrix dimensions.

We can see this in Figure 6.5 which shows that in in the $32-bit$ integer number 7,694, there are 13 bits used and 19 bits that are available to be stolen.

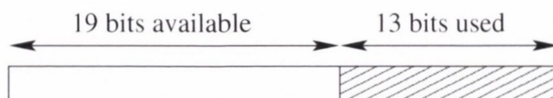


Figure 6.5: Integer Bits Available in the number 7,694

In this case, although there are 19 bits available, for a 'k' value of the heuristic of $k = 4$, we would only steal nine bits, thus ensuring that there are still four bits left in the row index. This would leave us with $\sim 2^4 = 16$ HyperPartitions with $\sim 2^9 = 512$ rows per HyperPartition.

The Remaining Bits Heuristic is the only heuristic presented in the HyperPartition experiments below.

We can vary the 'k' parameter to balance the work of the transpose between the cycle chasing phase and the sorting phase. Using this heuristic we have found from experimentation that the following values for the k parameter of the remaining bits heuristic give good performance.

- $k = 9$ or $k = 10$ for Serial HyperPartition
- $k = 5$ or $k = 6$ for Parallel HyperPartition

As discussed above, $steal_bits$ also influences memory usage as it determines the number of HyperPartitions and hence the size of the four hyp_* arrays the algorithm needs. In all cases we steal enough bits and the HyperPartitions are large enough that memory usage is negligible.

6.7 HyperPartition Memory Usage

As discussed in Section 6.1.3, the memory overhead of the HyperPartition algorithm is $\Theta(n)$, however in practice it is just a tiny fraction of n depending on the number of bits stolen, so could be of the order of $\sim(\frac{n}{2^7})$ or less. This is evident from Figure 6.6. Even showing relative memory usage of

the HyperPartition algorithm compared to Saad, the memory usage of HyperPartition is just a flat line at zero at the bottom of the graph.

In order to see the memory of the HyperPartition algorithm for different matrices, Figure 6.7 shows a zoomed close-up of the bottom of the memory usage graph from Figure 6.6. Given the memory usage for Saad is so much larger and does not fit on the graph, we show two dotted lines representing 1% and 0.25% of the relative memory usage of Saad. Aside from three outliers, the memory overhead of the HyperPartition algorithm for the majority of inputs matrices from our test suite is well below the 0.25% line. The overall average memory usage of the HyperPartition algorithm for these matrices is 0.07% of that of Saad.

The HyperPartition algorithm performs the In-Place Sparse Matrix Transpose with negligible memory overhead particularly compared to Saad and OOP which for the largest matrix *nlpkkt240* require up to 1,531 MiB and 4,699 MiB respectively. HyperPartition uses a maximum of just 3.3 MiB for this largest matrix.

Figures 6.6 and 6.7 show the HyperPartition algorithm where we are using the heuristic described in Section 6.6 where we always attempt to *leave behind* at least k bits in order for the HyperPartition Cycle Chasing to be efficient. In this case we are using a value of $k = 9$, which we have found experimentally to give good efficiency overall for the serial HyperPartition algorithm across the sample matrices used.

Table A.3 on page 226 shows the number of bits which have been *stolen*, *left behind* and *remain available* for a set of sample matrices from the test suite when using the HyperPartition algorithm with the $k = 9$ heuristic. Table A.3 also shows the memory overhead of the various transpose algorithms for the set of sample matrices.

6.8 HyperPartition Transpose Execution Time

Figure 6.8 shows the execution time of the HyperPartition Algorithm relative to the Saad-IP algorithm. The HyperPartition algorithm is being run serially with a heuristic setting of $k = 9$ and using the simple *Two-Array*

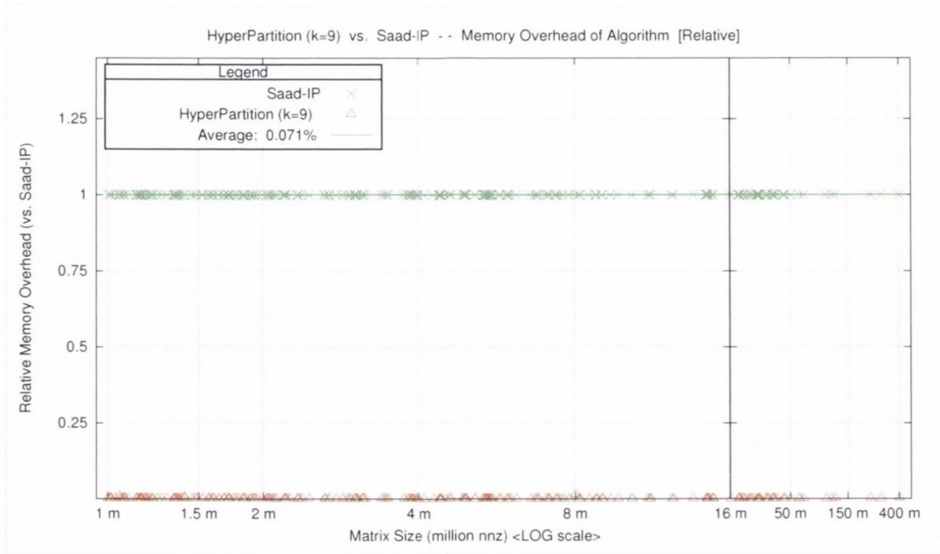


Figure 6.6: The relative memory usage of the HyperPartition Transpose ($k = 9$) algorithm is negligible compared to Saad, only showing as a line along the bottom at zero.

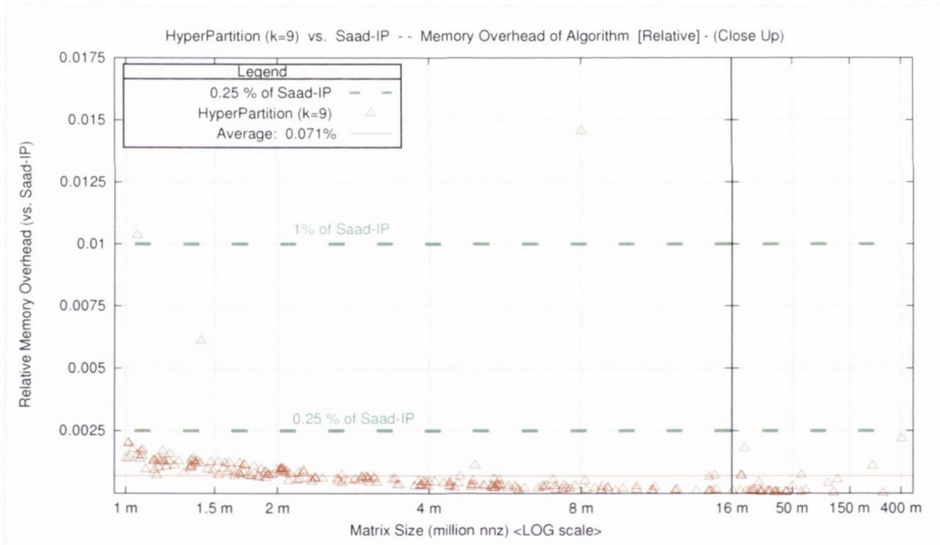


Figure 6.7: Close-up Relative memory usage of the HyperPartition Transpose ($k = 9$). This graph shows two dotted lines to represent 1% and 0.25% of the memory usage Saad. Most inputs use less than 0.25% with an average of 0.07%.

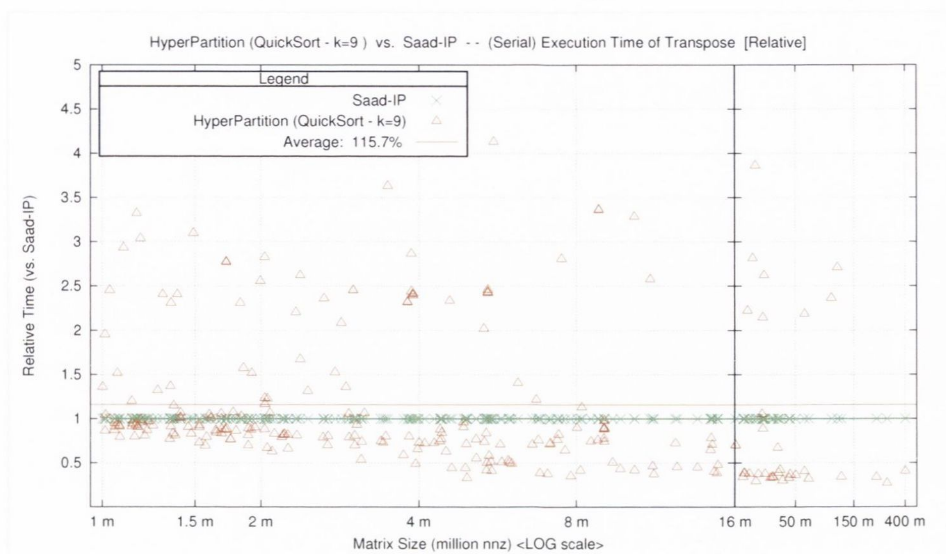


Figure 6.8: Serial execution time of HyperPartition Transpose ($k = 9$) with QuickSort compared to Saad. The time complexity of HyperPartition is $\Theta(nnz + n)$ however the performance is quite variable due to cache performance. HyperPartition is quite a bit slower in some cases, however for a large number of matrices the HyperPartition performs the transpose much faster than Saad showing the benefit of improved caching.

QuickSort from Section 4.5 for the *Phase-II* sorting part of the algorithm. Note that these graphs show the full transpose time which includes both the time for the cycle chasing transpose phase and the sorting phase for both algorithms.

In Figure 6.8 the graph shows the execution time of the algorithm using the parameter $k = 9$ for our *Remaining Bits* heuristic (Section 6.6.1). As we will see in Section 6.8.2 this gives the best overall performance for the serial HyperPartition algorithm.

It is clear from the graph that for a number of matrices the HyperPartition transpose is considerably slower than Saad. The majority of these *slow* matrices are the "*Structurally Symmetric*" matrices which we identified in Sections 5.10 and 5.11. These structurally symmetric matrices already have very short cycle lengths and have very good performance for the in-place Saad and Corresponding Row algorithms compared to other matrices of comparable size. These matrices will be examined in depth in Section 7.4

where we show that the reason for HyperPartition being slower for these matrices is that switching to the HyperPartition format means that we no longer get the two-element cycles and thus no longer enjoy the benefit that Saad and Corresponding Row gain from the symmetry.

In Chapter 7 we also propose methods to quickly identify and efficiently handle these structurally symmetric matrices and improve the performance of the algorithm when transposing them.

6.8.1 Excluding Structurally Symmetric Matrices — For HyperPartition Transpose Execution Time

Given that we know these Structurally Symmetric matrices cause a particular problem for the HyperPartition transpose and we show how to simply identify and handle them in Chapter 7 it is informative to see how the HyperPartition algorithm performs on the other, non-symmetric matrices. Figure 6.9 shows the performance of the serial HyperPartition *excluding* those matrices which are structurally symmetric. This figure again shows the performance of the serial HyperPartition transpose with a *remaining bits* heuristic value of $k = 9$. Varying the value of k had very little impact on the overall performance of the serial HyperPartition transpose after the structurally symmetric matrices were excluded. The value did alter the performance of the algorithm for individual matrices, however for some matrices it improved and others it degraded. The result of this is that the average performance of the algorithm to Saad does not vary greatly for different values of the k heuristic.

Figure 6.9 highlights a number of things that were somewhat obscured in Figure 6.8. When we look past the Structurally Symmetric matrices we find that the HyperPartition in-place transpose is very efficient at transposing a large number of these matrices. For quite a few matrices the HyperPartition algorithm transposes the matrix in less than 50% of the execution time of Saad, while using less than 0.25% of the memory. Overall the serial HyperPartition algorithm transposed this subset of matrices in just 80.3% of the time of the Saad algorithm.

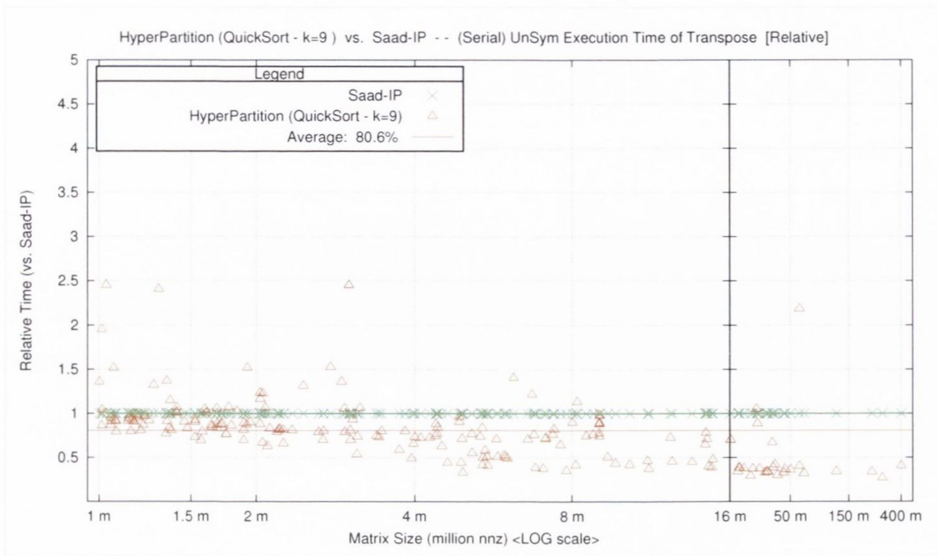


Figure 6.9: Symmetric Matrices Excluded: Serial execution time of HyperPartition Transpose ($k = 9$) with QuickSort compared to Saad for non structurally symmetric matrices. This is the same graph as Figure 6.8 just with results for symmetric matrices omitted. **Note:** This figure and Figures 7.9, B.3 and B.4 are the only figures with matrices excluded. All other graphs include results of all matrices in the test suite.

There also seems to be a strong trend indicating that the HyperPartition Transpose is becoming even more efficient than Saad as matrix size increases. Furthermore, there is actually a greater than 50% reduction for most of the large matrices. Coupled with the huge reduction in memory overhead, this is a very good result. Recall that for the largest matrix, *nlpkkt240*, OOP requires 4,699 MiB in memory overhead, Saad requires 1,531 MiB and HyperPartition requires just 3.3 MiB and can transpose the matrix in 80.7 seconds, just 36% of the execution time of Saad.

These results clearly show that by performing the in-place transpose using the HyperPartition format reduced the average cycle length which led to improved cache reuse and thus much better execution time for a large number of matrices.

There are still a small number of matrices for which the HyperPartition transpose performs poorly which pulls up the average relative execution time. However, do recall from Figure 6.6 (and refer to Table A.3) that the

HyperPartition transpose uses just a tiny fraction of the memory overhead of Saad, 0.07% on average. Further methods to improve the performance of the HyperPartition transpose will be investigated in Chapter 7.

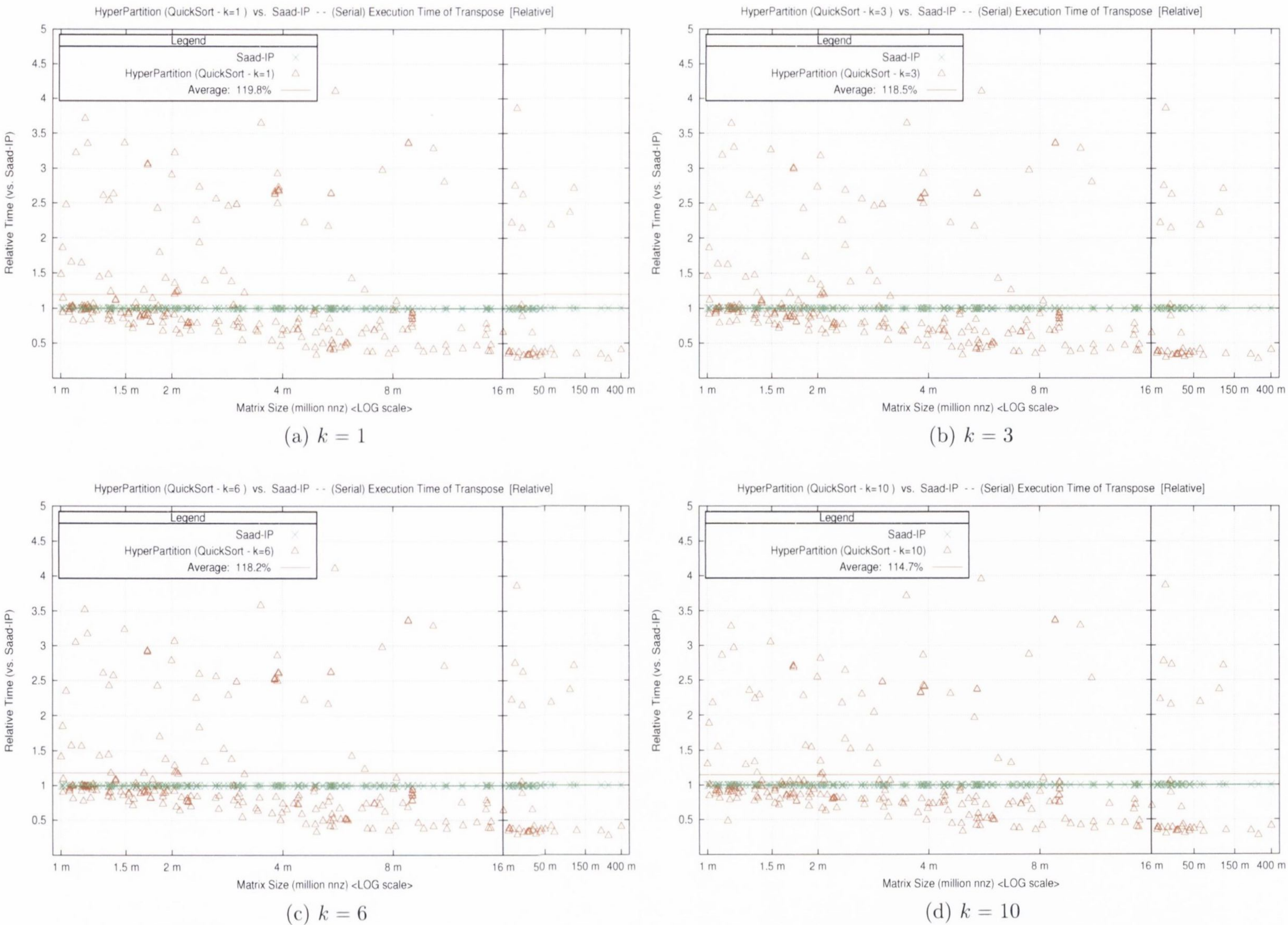
6.8.2 Serial Performance of the *Remaining Bits* Heuristic

Section 6.6 discussed the '*remaining bits*' heuristic which we can use to determine the number of bits that should remain behind after we steal some of the available bits when converting to the HyperPartition format. This parameter directly influences the trade-off between the number of HyperPartitions and the number of rows/elements per HyperPartition. This in turn controls the trade-off in work done during the Phase-I cycle-chasing part of the algorithm and the Phase-II sorting part of the algorithm.

Figure 6.10 shows the influence of selecting different values of 'k' on the execution time of the Serial HyperPartition algorithm with QuickSort. The figure just shows four values of k: $k = 1$, $k = 3$, $k = 6$ and $k = 10$. Figure B.1 in Appendix B shows the performance for all values of k between 1 and 10.

We can see from Figure 6.10 and Figure B.1 that the 'k' value does not have a very big impact on the overall performance of Serial HyperPartition with QuickSort. Overall the performance of the algorithm transposing our 259 matrices runs from an average relative execution time of 120% of Saad with a parameter of $k = 1$ to 115% with a parameter of $k = 10$. The improvement is not dramatic but it is still a reasonable reduction of 5%. Thus for Serial HyperPartition with QuickSort, the recommended value for the 'k' heuristic is $k = 9$ or $k = 10$.

Closer inspection shows that for some individual matrices as the number of remaining bits (as determined by the value of 'k') increases, the execution time performance improves. However, for other matrices, as the value of 'k' increased the execution time of the algorithm for these matrices deteriorates. Thus showing how this trade-off in work differs between matrices. For some matrices the algorithm performs better doing more of the work in the cycle



(a) $k = 1$

(b) $k = 3$

(c) $k = 6$

(d) $k = 10$

Figure 6.10: Serial Hyperpartition with QuickSort with; $k = 1, 3, 6, 10$

chasing phase and for other matrices the algorithm performs better doing more of the work in the sorting phase. This leaves us with an overall performance that only improves slightly with larger 'k' values.

Seeing how this trade-off in work between cycle-chasing and sorting influences performance, it shows that as we give more work to the sorting phase it means that we should look at ways of improving the sorting phase. This can be done in two ways: `a)` Performing the sort in parallel, which we will investigate next and `b)` Improving the sorting operation which we will investigate in Chapter 7.

6.9 Parallel HyperPartition Transpose

A side benefit of performing a larger proportion of the work of the transpose in the Phase-II Sorting step is that the work of sorting each HyperPartition is completely isolated from all the other HyperPartitions. This means that it becomes incredibly easy to parallelize a large proportion of the transpose. It has always been easy to perform the sorting in parallel, however as we saw in Figure 4.9 in Section 4.5, it was only a small proportion of the total work. In which case, performing the QuickSort in parallel would not improve greatly on the overall execution time of the transpose. With HyperPartition sorting there is much more to be gained from doing this in parallel. Figure 7.1 in Chapter 7 shows the proportion of time being spent on each of the cycle-chasing and sorting phases of the serial HyperPartition transpose with QuickSort.

6.9.1 Parallel Sorting Algorithm

Parallelizing the sorting step is very straightforward as shown in Algorithm 6.4 which is a modified parallel version of Algorithm 6.3. The easiest way is simply to compile with OpenMP[Dagum 98] enabled and to add a "`#pragma omp parallel for`" before the `for()` loop on line 3 in Algorithm 6.4. Setting `openmp_num_threads(`x`)` at the start of the application will cause OpenMP to automatically split iterations of the for loop across

'x' processor threads on the machine. This has the added benefit that the HypCSR to CSR conversion routine will also run in parallel. All the operations on the HyperPartitions are independent on the different processors so they should not interfere with each other. However all processes write to the same $new_row_ptrs[\,]$ array when counting the new row indexes. Therefore it is necessary to include a "`#pragma omp atomic`" before this increment on line 15 to ensure synchronisation.

Using OpenMP is a quick and simple method of parallelizing which works quite well in this case as we simply wish to split the HyperPartitions up between processors to be sorted independently. Some more focus on the distribution of work, parallelization and synchronization with OpenMP could lead to further improvements in throughput.

6.10 Parallel HyperPartition Memory Usage

Similar to Section 6.7 which showed the memory for the serial HyperPartition transpose, this section shows the memory usage for the parallel HyperPartition transpose with a heuristic value of ($k = 6$) compared to the parallel Saad algorithm. A value of ($k = 6$) was chosen because values of $k = 5$ and $k = 6$ give the best performance for the parallel HyperPartition transpose. As lower values of k mean that we steal more bits, we therefore have a smaller number of larger HyperPartitions. This reduces the size of the arrays used during cycle chasing which gives a lower overall memory usage when $k = 6$.

Figure 6.11 shows the relative memory usage of the parallel HyperPartition transpose algorithm compared to the parallel Saad algorithm. Again the memory overhead of HyperPartition is such a tiny fraction of Saad that it just shows as a line on the bottom of the graph at zero. We show a close-up of the bottom of the graph in Figure 6.12. Here again the memory usage of Saad is so high that we just show a dotted line for 0.25% of the memory usage of Saad. The memory usage of HyperPartition at $k = 6$ is below this 0.25% line for all input matrices in our test suite and well below this line for the majority. The average memory overhead is 0.016%.

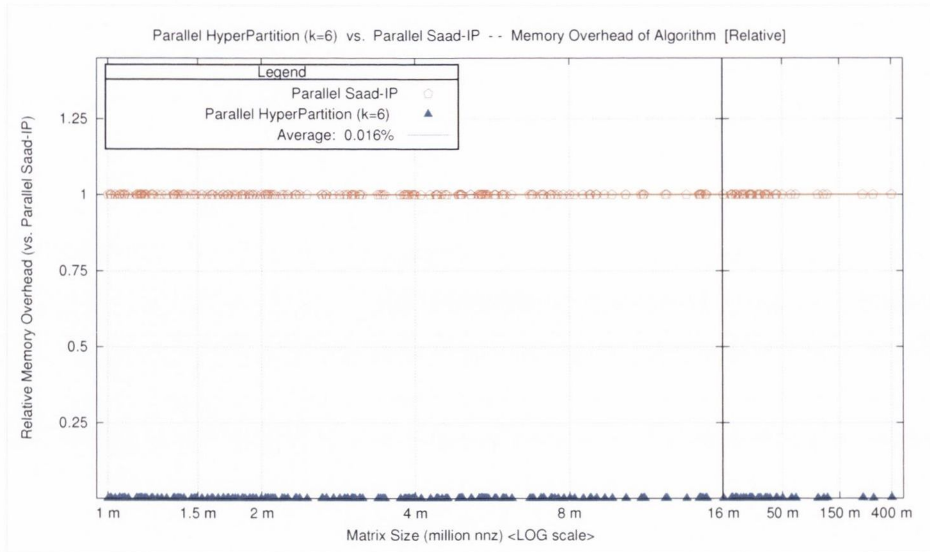


Figure 6.11: The relative memory usage of the Parallel HyperPartition Transpose ($k = 6$) algorithm is negligible compared to Parallel Saad, again, only showing as a line along the bottom at zero.

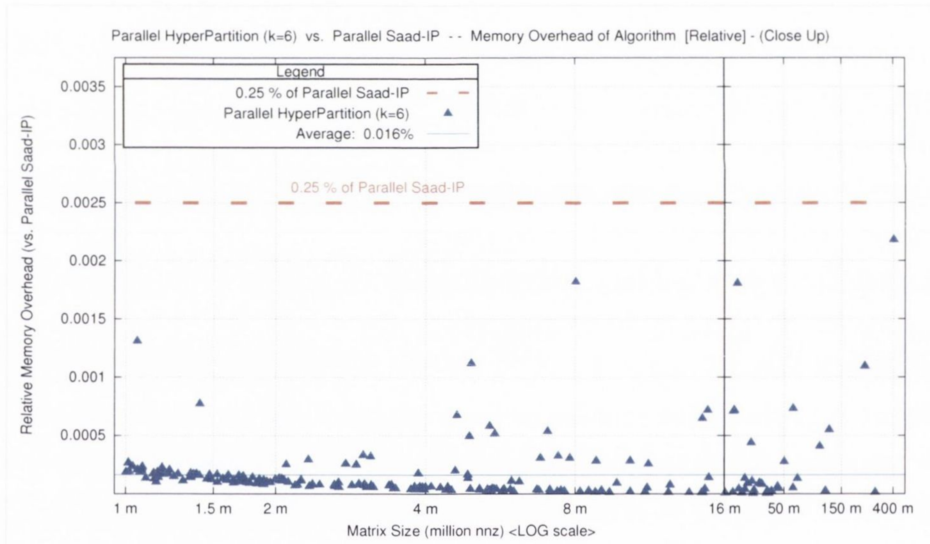


Figure 6.12: Close-up relative memory usage of the Parallel HyperPartition Transpose ($k = 6$). This graph shows a dotted line to represent 0.25% of the memory usage of Saad. All input matrices use less than 0.25% with an average of 0.016%.

The memory usage of the serial and parallel HyperPartition algorithms with QuickSort are identical. However, the value of k for the remaining bits heuristic does influence the memory usage of the algorithm. In this case, because we are using a value of $k = 6$ compared to $k = 9$ for the serial version, the memory usage is lower. The parallel algorithm at $k = 6$ has an average overall usage of 0.016% compared to the 0.071% usage of the serial version at $k = 9$.

Larger values of 'k' result in greater memory usage for the HyperPartition algorithm as we are stealing fewer bits and this causes the HyperPartition structure to have a larger number of HyperPartitions each containing fewer rows.

With all values of k for the *remaining bits* heuristic between $k = 0$ and $k = 10$ the memory usage of the HyperPartition algorithm is much less than Saad.

6.11 Parallel HyperPartition Execution Time

Note: In these graphs of the parallel HyperPartition we also show the parallel execution time of the Saad algorithm. With the Saad algorithm only the sorting phase can be done in parallel so there is some parallel speedup, but it is minimal.

Figure 6.13 shows the execution time of the HyperPartition algorithm with QuickSort running in parallel for the sorting phase across 32 cores on our experimental machine. Figure 6.13 shows the execution time of the algorithm using a *remaining bits* heuristic parameter of $k = 6$ which we found to give the best overall performance. See Section 6.11.2 for further details.

The Parallel HyperPartition with QuickSort is significantly faster than the parallel Saad with QuickSort for the majority of inputs. Taking 42.7% of the execution time of Saad on average, with many matrices requiring even less than this.

Earlier in Section 6.6 we discussed the selection of the number of bits to steal and found when using the heuristic $((need_bits - steal_bits) \geq$

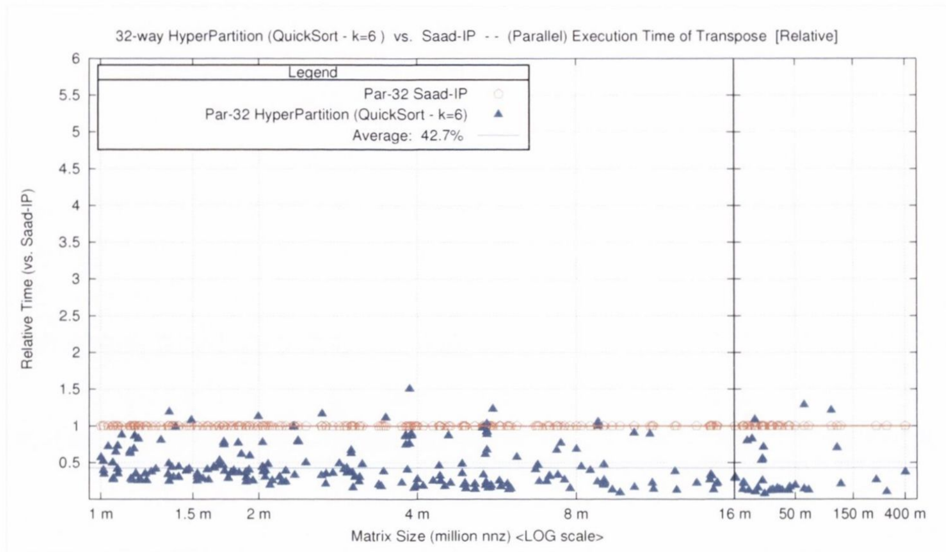


Figure 6.13: Parallel execution time of HyperPartition Transpose with QuickSort (with a heuristic value of ($k = 6$) compared to Saad. The parallel HyperPartition is slightly slower for a small number of inputs, however it is considerably faster than Saad for the majority of inputs taking on average 42.7% of the execution time of Saad overall.

k) that a value of about $k = 9$ or $k = 10$ gave good results for the serial HyperPartition with QuickSort/InsertionSort. When performing the HyperPartition with QuickSort/InsertionSort in parallel across 32 processor threads, we can afford to do more work in the sorting Phase-(II) part of the algorithm. As such, we have found experimentally that a value of $k = 6$ gives a good performance for HyperPartition with QuickSort in parallel on 32 cores based on our sample set of matrices from Florida (see Section 3.6.2).

6.11.1 Parallel HyperPartition vs. Serial Saad

Note that for these parallel results, the sorting phase of the Saad algorithm was also performed in parallel on 32 cores and is displayed as such on the graphs. In order to see how the parallel HyperPartition compares to the original *serial* version of Saad, Figure 6.14 shows the execution time of the parallel version of the HyperPartition Transpose with QuickSort compared

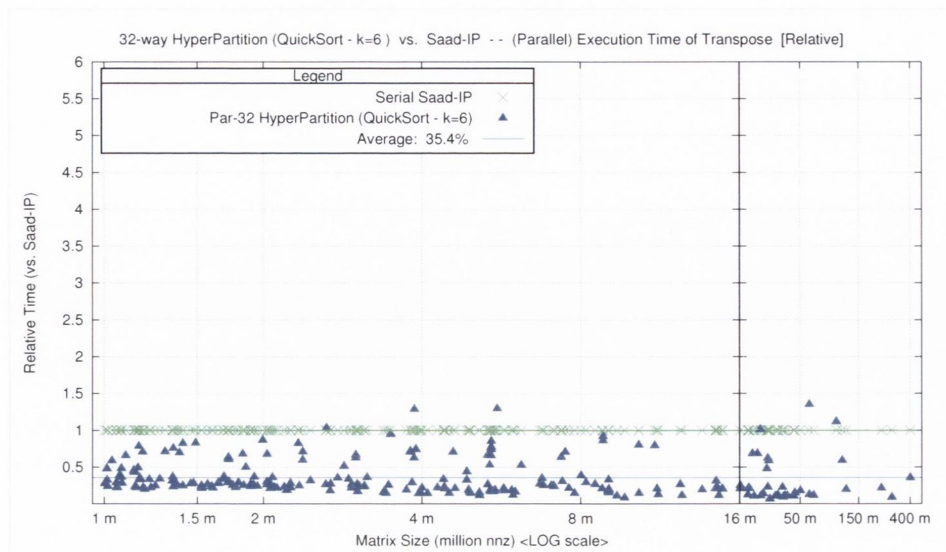


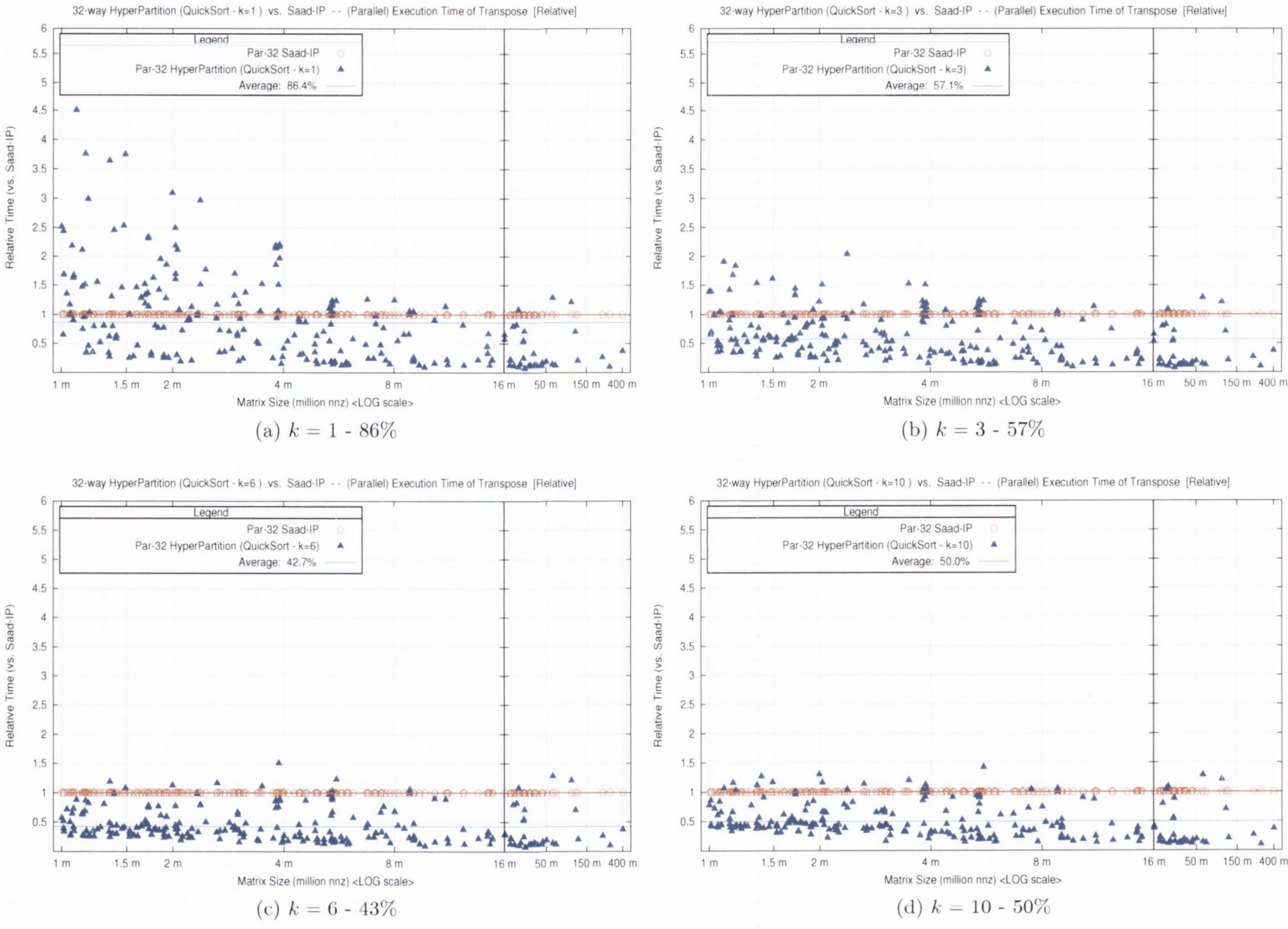
Figure 6.14: Parallel execution time of HyperPartition Transpose with QuickSort compared to the execution time of the Serial version of Saad. The Parallel Hyperpartition performs transpose on average in 35% of the time of Saad.

to the **serial** execution time of Saad. Using less than 1% of the memory overhead of Saad the HyperPartition algorithm performs the transpose on average in 35% of the time of Saad with the majority of input matrices taking even less time than this.

6.11.2 Parallel Performance of *Remaining Bits* Heuristic

Figure 6.15 shows the performance of the Parallel Hyperpartition with QuickSort for the *remaining bits* heuristic for the 'k' parameter values of $k = 1$, $k = 3$, $k = 6$ and $k = 10$. Figure B.2 in Appendix B shows the performance of this algorithm for all 'k' values between 1 and 10.

Unlike the graphs for the serial Hyperpartition in Section 6.8, these graphs show that different values of the 'k' parameter have a very profound influence on the performance of the Algorithm for the matrices overall. The parallel HyperPartition with QuickSort has an overall average execution time of 86% of Saad at $k = 1$ to 42.7% of Saad for $k = 6$. Interestingly,



(a) $k = 1$ - 86%

(b) $k = 3$ - 57%

(c) $k = 6$ - 43%

(d) $k = 10$ - 50%

Figure 6.15: Parallel Hyperpartition with QuickSort with; $k = 1, 3, 6, 10$

$k = 6$ appears to be the sweet spot for this algorithm as the performance then begins to deteriorate again for larger values of k, going up to 50% of Saad at $k = 10$.

It is evident from these graphs that the heuristic and the value of 'k' has a bigger impact on the smaller matrices. This is because for the larger matrices, even if we steal all the available bits, there are still a large number of bits remaining. Thus, changing the value of 'k' will not change the number of bits we steal from these matrices until 'k' becomes very large.

For the smaller matrices, increasing the value of 'k' means that we can do more of the transposing work in parallel during the sorting phase. A value of $k = 6$ gives the best performance in this case, running at an average of 42.7% of the Execution Time of Saad overall with a large proportion of the matrices running much faster than this. This is a very good result for an algorithm that has a memory overhead of less than 1% of Saad (see memory usage in Figure 6.6).

6.12 Reviewing the *Remaining Bits* Heuristic

In Section 6.6 we proposed the *Remaining Bits* heuristic for deciding on the number of bits that should be stolen when converting from the CSR sparse matrix storage format to our new HyperPartition CSR format. The heuristic suggests that we should: *Steal as many bits as possible but always leave at least 'k' bits behind.*

We saw how this heuristic influenced the performance of the Serial HyperPartition with QuickSort in Figure 6.10 in Section 6.8.2. We also saw how it influenced the performance of the Parallel HyperPartition with QuickSort in Figure 6.15 in Section 6.11.2.

We found that the heuristic only has a small effect on the performance of the serial HyperPartition transpose. The execution time did improve from a relative performance of 120% of Saad when stealing all available bits to 114.7% when leaving at least $k = 10$ bits behind. An improvement

of over 5%.

The heuristic had a much larger influence on the performance of the Parallel HyperPartition algorithm as the heuristic allowed more of the work to be done in the parallel sorting phase of the transpose. The relative performance of the algorithm compared to Saad went from 94.6% when stealing all available bits to 42.7% when ensuring that there are at least $k = 6$ bits left. This is a very significant *halving* of the overall performance of the algorithm.

Larger values of 'k' caused the performance of the parallel HyperPartition to increase to 50% of Saad when $k = 10$. This is because this creates a greater number of smaller HyperPartitions which loses the benefit of performing the sorting in parallel. The heuristic had a much greater effect on the performance of the parallel HyperPartition algorithm when transposing the smaller matrices in the test suite where, as we predicted, stealing all available bits would have an averse affect resulting in a very small number of very large HyperPartitions.

An important point to note from investigating the influence of the heuristic on the performance of the algorithm is that while modifying the parameter improved the performance of the algorithm for some matrices, it also degraded the performance of the algorithm for other matrices at the same time. This behaviour should be investigated further. It may be appropriate to use different heuristic parameters for different matrices which have different dimensions and which lead to different HyperPartition sizes.

6.13 Summary

We have seen in this chapter how our new HyperPartition sparse matrix format greatly reduces the memory overhead of the in-place sparse matrix transpose to just a fraction (less than 1%) of the existing algorithms.

We have seen in this chapter how we can use our new HyperPartition sparse matrix format to perform the in-place cycle-chasing transpose in a more cache friendly manner by reducing the length of cycles which

are chased during the transpose. Thus allowing us to transpose the non-symmetric matrices in an average overall execution time of just 80% of Saad with many of the larger matrices having a greater than 50% reduction in execution time.

We also found that switching to our HyperPartition format allowed us to perform a larger amount of the transpose work during the sorting phase of the transpose allowing even further benefits from good cache reuse and also allowing us to perform the sorting phase in parallel. For our 259 matrices, the parallel HyperPartition gave an overall average runtime of 42.7% of the execution time of the parallel version of Saad and 35.4% of the execution time of the serial version, over all input matrices. In the next chapter we present methods to improve the execution time of the HyperPartition sorting phase in order to further improve the throughput of the sorting phase of the transpose.

In this chapter we also presented our *remaining bits* heuristic which allows us to balance the amount of work being done between the cycle-chasing and sorting phases of the transpose. We found that values of about $k = 9$ or $k = 10$ gave good performance for the serial HyperPartition transpose and values of about $k = 5$ or $k = 6$ gave good performance for the parallel HyperPartition transpose.

In this chapter we identified that that the HyperPartition transpose algorithm does not perform as well as Saad or Corresponding Row on matrices that are Structurally Symmetric. This is because the HyperPartition format interferes with their structure thus losing the benefit of the two-element cycle length. In the next chapter we will show how to quickly and easily detect if an input matrix is structurally symmetric and outline a technique for a Hybrid HyperPartition transpose to handle it more efficiently.

ALGORITHM 6.2: Cycle Chasing HyperPartition Transpose

Input: Matrix M in HyperPartition format from Algorithm 6.1

Output: Transposed Matrix M^T in HyperPartition format

```
1  /* Loop through each new Hyperpartition */
2  for (0 ≤ cur_hyp < n_new_hyp) do
3      /* For each of the (unmoved) elements in 'cur_hyp' */
4      for ( hyp_offsets[cur_hyp] ≤ x < new_hyp_ptrs[cur_hyp + 1] ) do
5          /* Take out element at 'x' and put in 'src' */
6          src_val ← non_zeros[x];
7          src_col ← col_indexes[x] & BOT_MASK;
8          src_hyp ←
           srch_upd_corresp(corresp[ ], cur_hyp, x, n_old_hyp, old_hyp_ptrs[ ]);
9          low_row ← ((col_indexes[x] & TOP_MASK) >> remain_bits);
10         src_row ← ((src_hyp << steal_bits) + low_row;
11         target_hyp ← (src_col >> steal_bits); /* The hyp the element in 'src' should be in
           */
12         while ( target_hyp ≠ cur_hyp ) do
13             /* Find destination for element in 'src' and store element at 'dst_x' into 'dst' */
14             dst_x ← hyp_offsets[target_hyp];          /* Destination index */
15             dst_val ← non_zeros[dst_x];
16             dst_col ← col_indexes[dst_x] & BOT_MASK;
17             dst_hyp ←
               srch_upd_corresp(corresp[ ], target_hyp, dst_x, n_old_hyp, old_hyp_ptrs[ ]);

18             low_row ← (col_indexes[dst_x] & TOP_MASK) >> remain_bits;
19             dst_row ← (dst_hyp << steal_bits) + low_row;
20             /* Copy 'src' into new 'dst' position */
21             non_zeros[dst_x] ← src_val;
22             col_indexes[dst_x] ←
               (((src_col & LOW_MASK) << remain_bits) + src_row);
23             hyp_offsets[target_hyp] ← hyp_offsets[target_hyp] + 1;    /* Flag 'dst_x' has
               moved */
24             /* Copy 'dst' into 'src' - So we can continue cycle-chasing the new element in 'src' */
25             src_val ← dst_val;
26             src_row ← dst_row;
27             src_col ← dst_col;
28             target_hyp ← (dst_col >> steal_bits); /* The hyp the element in 'src' should
               be in */
29         end
30         /* Found element that should be in 'cur_hyp' - copy that from 'src' back into 'x' */
31         non_zeros[x] ← src_val;
32         col_indexes[x] ← (((src_col & LOW_MASK) << remain_bits) + src_row);
33         hyp_offsets[cur_hyp] ← hyp_offsets[cur_hyp] + 1;
34     end
35 end
```

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
new_hyp_ptrs =	0_0						6_1		8_2							15
non_zeros =	a	h	c	d	m	f	i	g	b	e	j	k	l	n	o	
col_indexes =	0:0	0:3	0:1	1:1	1:5	1:2	1:3	0:2	0:0	1:1	0:3	0:4	1:4	0:5	1:5	

Example 6.3: Matrix M^T in HypCSR after Hyper Circuit Chasing
Note: Elements are not in correct row or column order.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
new_hyp_ptrs =	0_0						6_1		8_2							15
non_zeros =	a	c	h	d	f	m	g	i	b	j	k	n	e	l	o	
col_indexes =	0:0	0:1	0:3	1:1	1:2	1:5	0:2	1:3	0:0	0:3	0:4	0:5	1:1	1:4	1:5	

Example 6.4: Matrix M^T in HypCSR after Hyper-Sorting
Note: Elements are now sorted into their correct row and column index ordering.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
non_zeros =	a	c	h	d	f	m	g	i	b	j	k	n	e	l	o	
col_indexes =	0	1	3	1	2	5	2	3	0	3	4	5	1	4	5	
new_row_ptrs =	0_0			3_1			6_2	7_3	8_4				12_5			15

Example 6.5: Transposed Matrix M^T in CSR representation after sorting and conversion back.

ALGORITHM 6.3: Convert HyperPartition back to CSR format

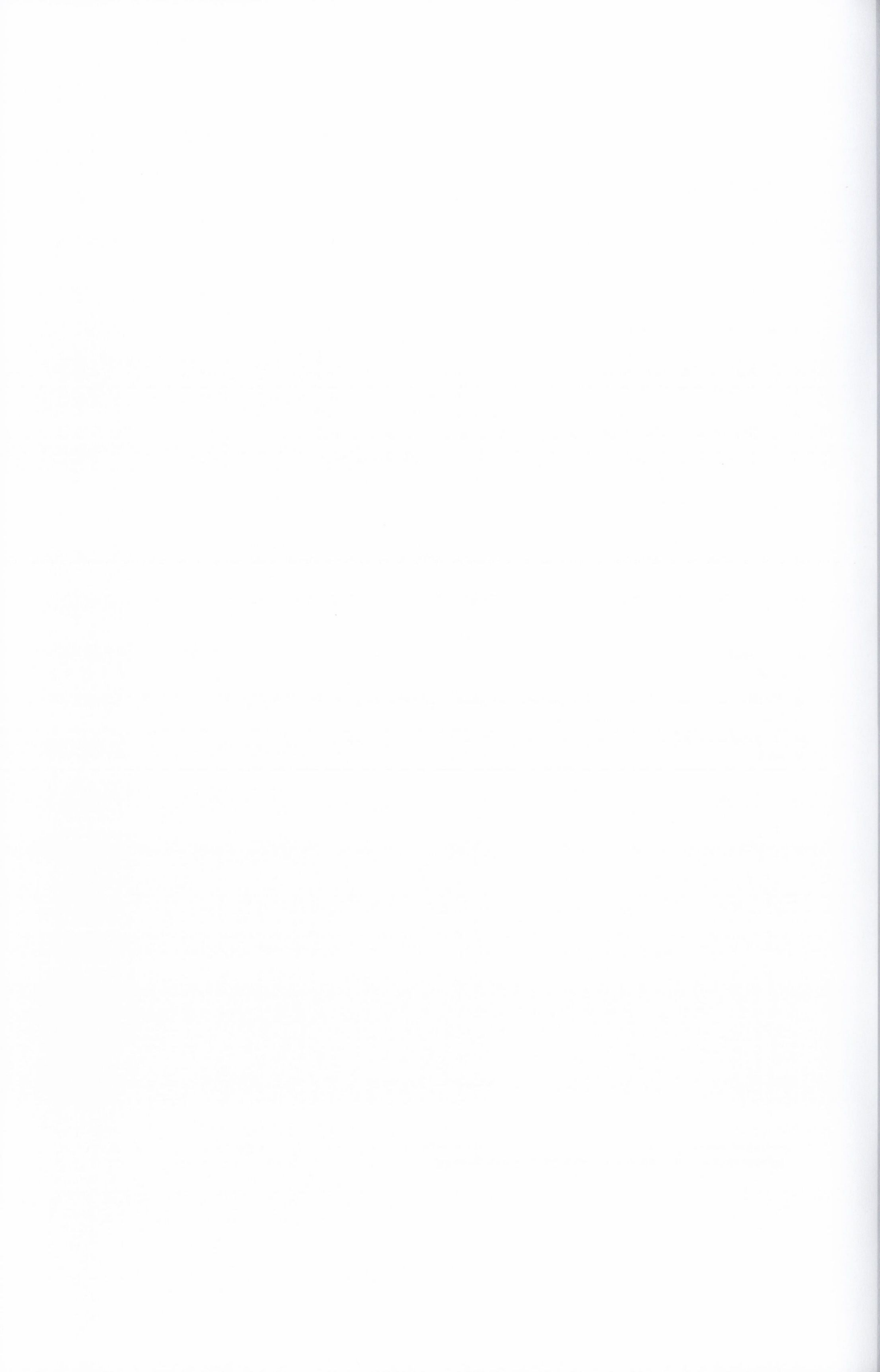
Input: Matrix M^T in HyperPartition format from Algorithm 6.2
Output: Matrix M^T in CSR format

```
1  Allocate: new_row_ptrs[new_nrows + 1];
2  /* Loop through each hyperpartition */
3  for (0 ≤ hyp < n_new_hyp) do
4      /* Beginning and end of hyperpartition 'hyp' */
5      start ← hyp_ptrs[hyp];
6      stop ← hyp_ptrs[hyp + 1] − 1;
7      /* Sort this HyperPartition - See Sections 4.5 and 7.1 */
8      HyperPartition_Sort(col_indexes[ ], non_zeros[ ], start, stop);
9      /* Get the most significant bits of the row index from current 'hyp' */
10     row_msb ← (hyp << steal_bits);
11     /* Count row indexes to build new_row_ptrs[ ], then fix all of the col_indexes[ ] in this
       hyperpartition */
12     for ( start ≤ x ≤ stop ) do
13         row ← (row_msb + ((col_indexes[x] & TOP_MASK) >> remain_bits));
14         new_row_ptrs[row + 1] ← new_row_ptrs[row + 1] + 1; /* Count row indexes -
           Offset by 1 */
15         col_indexes[x] ← (col_indexes[x] & BOT_MASK);
16     end
17 end
18 /* Cumulative sum - Turn index counts into row pointers */
19 for ( 1 ≤ i ≤ new_nrows ) do
20     new_row_ptrs[i] ← new_row_ptrs[i] + new_row_ptrs[i − 1];
21 end
```

ALGORITHM 6.4: Convert HyperPartition back to CSR format in Parallel

Input: Matrix M^T in HyperPartition format from Algorithm 6.2
Output: Matrix M^T in CSR format

```
1  Allocate: new_row_ptrs[new_nrows + 1];
2  /* Loop through each hyperpartition */
3  #pragma omp parallel for
4  for (0 ≤ hyp < n_new_hyp) do
5      /* Beginning and end of hyperpartition 'hyp' */
6      start ← hyp_ptrs[hyp];
7      stop ← hyp_ptrs[hyp + 1] − 1;
8      /* Sort this HyperPartition – See Sections 4.5 and 7.1 */
9      HyperPartition_Sort(col_indexes[ ], non_zeros[ ], start, stop);
10     /* Get the most significant bits of the row index from current 'hyp' */
11     row_msb ← (hyp << steal_bits);
12     /* Count row indexes to build new_row_ptrs[ ], then fix all of the col_indexes[ ] in this
       hyperpartition */
13     for ( start ≤ x ≤ stop ) do
14         row ← (row_msb + ((col_indexes[x] & TOP_MASK) >> remain_bits));
15         #pragma omp atomic
16         new_row_ptrs[row + 1] ← new_row_ptrs[row + 1] + 1; /* Count row indexes -
           Offset by 1 */
17         col_indexes[x] ← (col_indexes[x] & BOT_MASK);
18     end
19 end
20 /* Cumulative sum - Turn index counts into row pointers */
21 for ( 1 ≤ i ≤ new_nrows ) do
22     new_row_ptrs[i] ← new_row_ptrs[i] + new_row_ptrs[i − 1];
23 end
```



Chapter 7

Further Optimisations RadixSort and Hybrid Transpose

In Chapter 5 we introduced the Corresponding Row transpose algorithm which reduced the space complexity of the Sparse In-Place Transpose to $\Theta(n)$ while maintaining the time complexity of $\Theta(nnz + n)$. Then in Chapter 6 we presented our HyperPartition algorithm which drastically reduced the memory overhead required for the sparse in-place transpose to just a fraction of both Saad and Corresponding Row in practice (Figure 6.6), while also improving the execution time for the majority of the matrix inputs (Figure 6.8).

In this Chapter we investigate further ways in which the HyperPartition algorithm can be improved. Section 7.1 presents the MSD RadixSort algorithm which improves the execution time of the sorting phase of the in-place transpose. With the HyperPartition algorithm, a larger proportion of the work of the transpose operation is being done during the sorting phase. Section 7.2 presents the performance results of the RadixSort Algorithm. Section 7.4 discusses structural analysis of sparse matrices in order to detect the structurally symmetric matrices for which the HyperPartition algorithm does not perform as well as Saad-IP. This leads to a quick and simple heuristic algorithm which can predict the likelihood that a particular matrix is structurally symmetric. We use this heuristic in Section 7.5 to develop a Hybrid HyperPartition in-place transpose algorithm which is more efficient at transposing structurally symmetric matrices at the cost of a quick test and a small increase in memory overhead. Section 7.6 presents the performance results of the Hybrid HyperPartition in-place algorithm.

7.1 Most Significant Digit (MSD) RadixSort

As discussed in Section 4.5 the In-Place Sparse Matrix Transpose is performed in two *phases*. The first is the cycle chasing phase which moves elements into their correct *new* row in the transposed matrix. After the cycle-chasing phase there is no guarantee that the elements will be in column order within each row. Therefore the second phase involves *sorting* the two *non_zeros*[ ] and *col_indexes*[ ] arrays based on the indexes in *col_indexes*[ ] to ensure the elements are stored *in-order* within each of the rows.

The existing Saad in-place circuit-chasing algorithm in the Sparskit2 package only moves elements to the correct new row. It does not ensure elements are ordered within the rows. The HSL library provides a routine `MC59()` for re-ordering completely unsorted sparse matrices in COO format. This routine also has an option for ordering the elements within columns of a CSC matrix. In many cases it is more desirable to have in-row elements which are in the correct order by column index, therefore we feel that it is important to also examine and improve the algorithms for this sorting step. In this section we investigate ways of improving the performance of the Phase-II step of ordering the HyperPartition sparse matrix.

When transposing with the Saad-IP, Binary Range Search, Radix Table Lookup, and Corresponding Row in-place algorithms, the sorting step accounts for just a small percentage of the overall execution time of the transpose as can be seen in Figure 4.9 in Chapter 4. As discussed in Section 4.5, we have been using a simple *Two-Array QuickSort* which drops to *InsertionSort* when ($length \leq$ LIMIT). However, as we improved the performance of the cycle chasing algorithm with the HyperPartition transpose, the relative cost of the sorting phase has become a much higher proportion of the overall execution time. Figure 7.1 shows the proportion of execution time of the serial HyperPartition algorithm with the *remaining_bits* heuristic parameter of $k = 6$ as shown in Figure 6.8. There is less work being done in the cycle chasing phase and more work being done in the sorting phase. We are also performing fewer sort operations, each on a much larger amount of data — so improvements to the sorting algorithm will have an

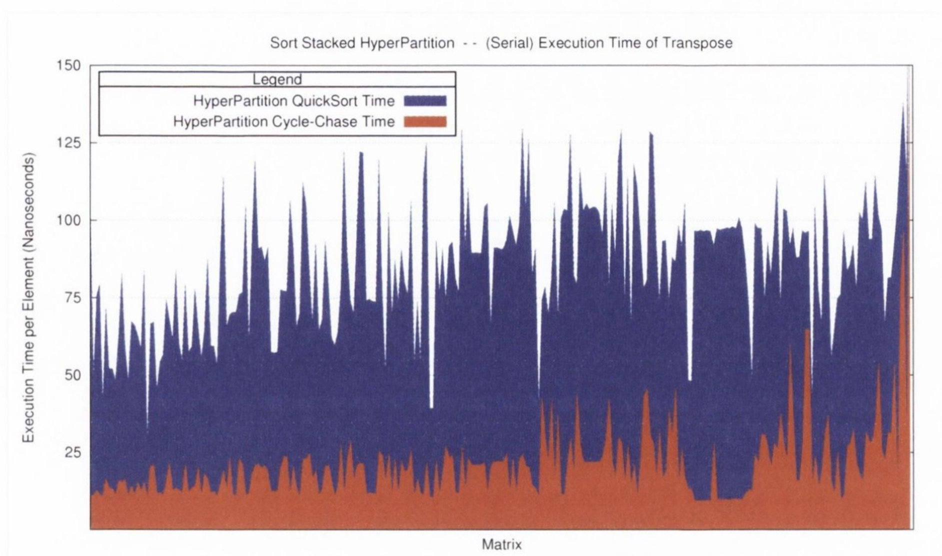


Figure 7.1: HyperPartition Sort Time stacked on top of Algorithm Time

ever bigger impact on the total execution time. With the HyperPartition algorithm, in some cases over 95% of the work of the transpose is being done in the sorting phase. Therefore it is worth investigating improvements to the sorting algorithm.

Using our knowledge of the type and distribution of the keys in the *col_indexes*[ ] array and the way the HyperPartitions need to be sorted in-place we believe that a Two-Array (see Section 4.5) version of a Most Significant Digit (MSD) In-Place RadixSort would be the best choice of algorithm in order to give good cache and execution time performance for sorting the matrix entries within the HyperPartitions. We need a Two-Array version, because like our QuickSort/InsertionSort algorithm, our new RadixSort needs to sort corresponding sections of the two *non_zeros*[ ] and *col_indexes*[ ] arrays based on the contents of the *col_indexes*[ ] array. Note also that when sorting the HyperPartitions we need to treat the integer values in the *col_indexes*[ ] array as unsigned integers in order to sort correctly.

In some ways, the generic cycle chasing method (Algorithm 4.1) we have been using is conceptually similar to a BucketSort algorithm. Except that

cycle chasing uses a very large number of very small buckets which makes the algorithm inefficient due to poor cache re-use. The HyperPartition algorithm groups rows into HyperPartitions which reduces the number of buckets and hence increases the average size of each bucket.

The RadixSort involves reading and copying/moving many elements multiple times. It seems unintuitive that moving elements three or more times gives better execution time performance than the generic cycle-chasing algorithm which only moves each element a single time. As we showed previously in Section 5.9, the cache performance of these algorithms when dealing with extremely large matrices has a huge impact on the execution time of the algorithms. The first (and second) pass of the RadixSort algorithm moves elements to buckets much closer to where they actually belong. In subsequent passes the algorithm will only need to process a smaller number of entries which are all contiguous in memory which would give much better temporal and spatial locality and therefore improved performance. The RadixSort is more cache friendly than using cycle chasing for the whole matrix and indeed it is more cache friendly than the QuickSort/InsertionSort algorithm we have been using.

985,042,230	=	00111010	10110110	10001101	00110110
pass	=	1st	2nd	3rd	4th

Example 7.1: Radix bit Passes of BucketSort

For our RadixSort we use a '*Radix*' on the bits of the integers in the *col_indexes*[ ] array to sort the matrix partition entries into buckets. For example: if we are using a RadixSort with 256 buckets, then we would use sets of eight bits ($256 = 2^8$) of the integers in *col_indexes*[ ] to sort the elements into buckets. Example 7.1 shows the bits in a 32-bit integer used on each of the four passes. In the first sorting pass we would use the first eight most significant bits (bits 31-24) to decide which bucket to place each element in. If an element had the first 8 bits $00111010 = 58$, then it would be moved to *bucket*[58]. On the second pass the algorithm goes through

each of the original 256 buckets, which now all have the same first eight bits, and further sorts them into sub-buckets based on the next eight most significant bits (bits: 23-16 = 10110110).

Using 256 buckets and a radix length of eight the RadixSort would sort all the elements in the HyperPartition after four levels of passes. A radix length of four *bits* (= 16 buckets) would take eight levels of passes to sort the arrays. As the RadixSort is not a comparison sort it will always sort the array in a constant k number of passes based on a given radix length and size of integer used for the indices. As such, no matter the size of the matrix and no matter the size of the HyperPartition, for any given radix length/number of buckets the RadixSort will always sort the partition in a constant number of passes. Thus the complexity of the RadixSort is $\mathcal{O}(nnz \, . \, k)$. Given k is constant for a particular radix length and index size, and does not vary with the size of input nnz, the complexity of the sort is essentially proportional to $\mathcal{O}(nnz)$.

7.1.1 MSD RadixSort Algorithm

Algorithm 7.1 shows our Two-Array variation of the Most Significant Digit In-Place RadixSort algorithm for sorting HyperPartitions using a radix length of eight *bits* which gives 256 buckets. On the first pass the algorithm sorts elements into buckets based on the most significant eight *bits*. The algorithm calls itself recursively, such that on subsequent passes the algorithm then sorts each sub-bucket based on the next eight *bits*. As with the QuickSort algorithm (Section 4.5), for efficiency we drop to InsertionSort when the array length is below a certain limit.

The algorithm needs two integer arrays of size $numBuckets = 256$. The memory overhead of the In-Place RadixSort is therefore very low. With 256 buckets the overhead is just 2 KiB. Two arrays are needed for every level of the RadixSort, using eight *bits* or 256 buckets would give four levels which would still require only 8 KiB. Even running this RadixSort in parallel over 32 cores would require at most 256 KiB of memory overhead. Given that the memory overhead of the Saad-IP algorithm for the smallest matrix in

ALGORITHM 7.1: Radix BucketSort Algorithm - 256 Buckets

Input: Matrix M^T in HyperPartition format from Algorithm 6.2

Output: Matrix M^T in HyperPartition format with ordered HyperPartitions

```
1  /* 8bit radix = 256 buckets */
2  radixBits ← 8;                      numBins ← 256;
3  Allocate: startBin[numBinss];       Allocate: endBin[numBinss];
4  /* Scan the array and count the number of items that will be in each bin */
5  for (0 ≤ cur ≤ last) do
6      digit ← ((indexes[cur] & bitMask) >> shiftAmt);
7      if (digit + 1 ≤ numBins) then
8          startBin[digit + 1] ← startBin[digit + 1] + 1;
9      end
10 end
11 /* Calculate the start and end of each bin from the count */
12 for (1 ≤ i < numBins) do
13     startBin[i] ← endBin[i] ← startBin[i] + startBin[i − 1];
14 end
15 /* Go through each "cur" element in the array and move it to the correct bin if necessary */
16 for (0 ≤ cur ≤ last) do
17     while (true) do
18         /* extract the digit we are sorting based on */
19         digit = ((indexes[cur] & bitMask) >> shiftAmt);
20         if (endBin[digit] == cur) then
21             break;
22         end
23         SWAP(indexes, values, cur, endBin[digit]);
24         endBin[digit] ← endBin[digit] + 1;
25     end
26     endBin[digit + 1] ← endBin[digit + 1] + 1; /* leave the element at its location and grow
       the bin */
27     cur ← cur + 1; /* advance the current pointer to the next element */
28     while (cur ≥ startBin[nextBin] && nextBin < numBins) do
29         nextBin ← nextBin + 1;
30     end
31     while (endBin[nextBin − 1] == startBin[nextBin] && nextBin < numBins)
       do
32         nextBin ← nextBin + 1;
33     end
34     if (cur < endBin[nextBin − 1]) then
35         cur = endBin[nextBin − 1];
36     end
37 end
38 bitMask ← bitMask ≫ radixBits;
39 if (bitMask ≠ 0) then
40     /* end recursion when all the bits have been processes */
41     shiftAmt ← shiftAmt − radixBits;
42     for (0 ≤ i < numBins) do
43         numEle ← (endBin[i] − startBin[i]);
44         if ((numEle) ≥ ISORT_DROP) then
45             /* endBin actually points to one beyond the bin */
46             RadixSort(indexes, values, startBin[i], (numEle −
               1), bitMask, shiftAmt);
47         end
48         else
49             ISORT(indexes, values, startBin[i], numEle);
50         end
51     end
52     Free: startBin;
53     Free: endBin;
54 end
```

our test suite is 3,910 KiB, this is still a small overhead and is very small compared to the memory required to store these very large matrices. In addition, with careful programming the bucket delimiting arrays can be reused during lower level passes in the sorting.

7.1.2 Choosing Number of Buckets for Radix Sort

There are a number of parameters/factors which influence the performance of the HyperPartition RadixSort. As discussed in Section 6.6, the number of bits we choose to steal when converting to HyperPartition format will influence the proportion of work that is split between the Phase-I cycle-chasing part of the transpose and the Phase-II sorting part of the transpose. The number of bits will also influence the size of the HyperPartition which will also influence the performance of the radix sort as the sort performs better on a smaller number of larger HyperPartitions. Another factor, as discussed in Section 6.9 is parallelism. Clearly the ability to perform more of the work in parallel will influence the performance of the algorithm and thus the scalability of the Radix Sort algorithm in sorting the two arrays (and converting back to CSR) is important.

The RadixSort adds another parameter to this, the number of buckets per level of the sort. The number of buckets per level affects the performance of the sort. A very small radix means there are just a few buckets, then we end up with a small number of very large buckets. This means that there will be a larger number of levels of the sort which will likely result in elements being moved many times and poor cache performance. With a large radix length there will be many buckets each with just a few elements. The sort begins attempting to move the elements to their exact location on the first pass which is not cache efficient as it runs into the same locality problems that we are trying to avoid with the HyperPartition algorithm.

Figure 7.2 shows the performance of the RadixSort algorithm for different numbers of buckets from 2 $\rightarrow$ 16,384 for a particular matrix (*RM07R*) which is a 381,689 $\times$ 381,689 matrix with 37,464,962 non-zero values. This matrix requires 19 bits to represent the row and column index leaving 13 bits

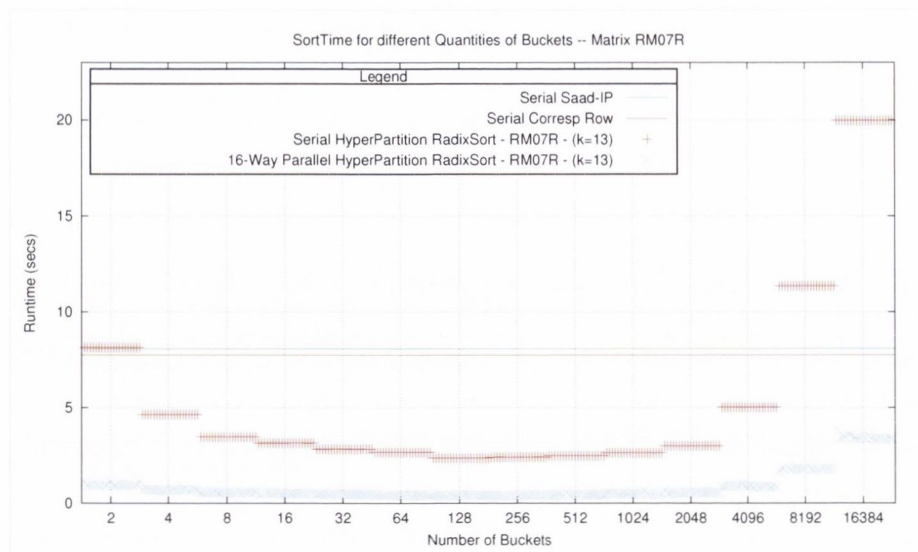


Figure 7.2: Serial and Parallel execution time (seconds) of the RadixSort for the *RM07R* matrix stealing 13 bits for different quantities of buckets. The algorithm was repeated in serial and parallel on 16 cores for 20 iterations at each quantity of buckets between 2 and 16,384. The cycle-chasing phase took roughly 0.441 seconds. For comparison: Saad-IP = 8.07 sec, Corresp Row = 7.73 sec.

available. If we steal all 13 bits leaving $k = 6$ behind then at the first level we have 64 HyperPartitions each with 8,192 rows and an average of 804,093 elements per HyperPartition. The transpose was repeated 20 times for each quantity of buckets for this particular matrix and HyperPartition size. The graph shows just the time for the Phase-II sorting part of the transpose as the time for the cycle-chasing portion is almost identical for all bucket sizes. The cycle chasing phase takes roughly 0.441 seconds in both the serial and parallel transpose. In Serial the division of work (in terms of time) is 16% cycle-chasing and 84% sorting. In Parallel the division of work (in time) is 49% cycle-chasing and 51% sorting.

The performance in Figure 7.2 is very consistent at each number of buckets. The performance of the serial RadixSort has a very distinct profile for this matrix. Starting at about 8.1 seconds at $b = 2$ buckets ($radix = 1$), improving with increasing number of buckets to a sweet spot at about $b = 128$ to $b = 256$ with a execution time of 2.4 seconds. Execution time

then begins to increase rapidly for larger quantities of buckets up to about 20 seconds at $b = 16{,}384$ buckets ($radix = 14$). The profile of the parallel RadixSort in Figure 7.2 is much flatter, starting at about 1 second for $b = 2$ buckets, down to about 0.42 at $b = 128$ buckets and increasing to 3.5 seconds at $b = 16{,}384$ buckets. For comparison the graph also shows the performance of serial Saad-IP = 8.07 seconds and serial Corresponding Row = 7.73 seconds for transposing this matrix.

The RadixSort exhibits a performance profile very similar to this across many of the combinations of input matrix and HyperPartition size. Unfortunately there is a large amount of variability in performance of the RadixSort algorithm between the different input matrices. No single pair of values for the $remaining_bits$ heuristic and $bucket_size$ gives optimal performance on all input matrices. A more complicated heuristic would need to be constructed which took in the many factors of matrix dimensions, bit availability, bits stolen and remaining, number of buckets (hence bucket size) and parallelism. For the results in the following Section we used a fixed quantity of buckets of $b = 256$ as this appeared to give a good performance for most inputs, however specific number of buckets (and hence bucket sizes) for each individual matrix would give better performance.

As discussed in Section 6.6.1 we found that for the 259 matrices in our test suite a value of $k = 9$ for the remaining bits heuristic — the number of needed bits left after stealing — gave a good performance. This value gave a good balance of work between the HyperPartition cycle chasing and sorting with QuickSort/InsertionSort. For serial RadixSort we again found that a value of $k = 9$ was a good choice for the HyperPartition transpose. For parallel RadixSort on 32 cores we also again found that $k = 6$ was a good choice as this value gives a higher proportion of work to the RadixSort portion which can be done in parallel.

7.2 HyperPartition RadixSort Results

Figure 7.3 shows the memory overhead of the Serial HyperPartition in-place transpose with RadixSort. The HyperPartition transpose was performed

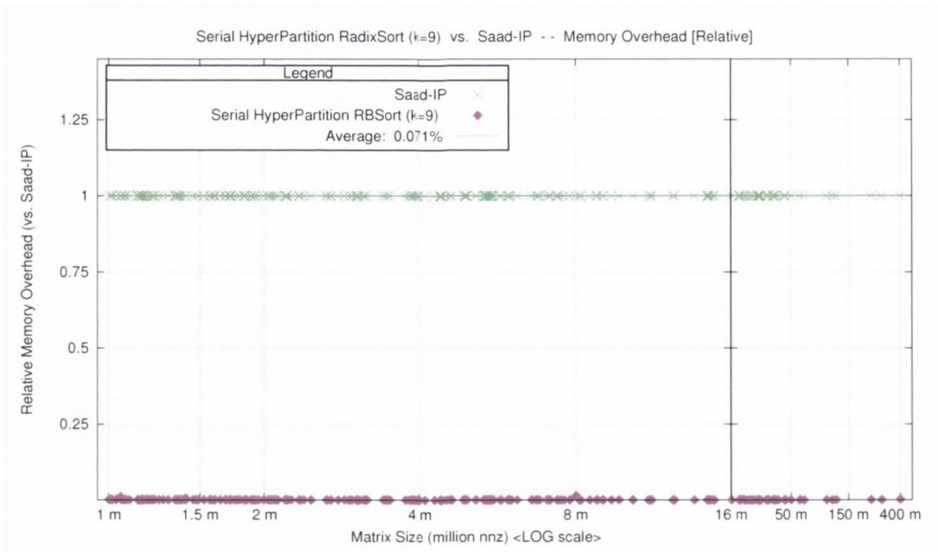


Figure 7.3: Serial Memory overhead of HyperPartition Transpose ($k = 9$) with RadixSort ($B = 256$) compared to Saad. The memory usage is almost identical to HyperPartition with QuickSort. There is just a tiny increase of 0.01% to an average of 0.017%. The memory requirements of Serial HyperPartition with RadixSort is less than 0.25% of that of Saad for all 259 input matrices.

using a remaining bits heuristic value of $k = 9$ and the RadixSort using $b = 256$ buckets ($radix = 8$). This graph is almost identical to the memory overhead of HyperPartition with QuickSort shown in Figure 6.11. The average memory overhead has increased slightly by 0.01% to 0.017%. The overhead remains less than 0.25% of that of Saad for all input matrices. The RadixSort has a memory overhead of 8 KiB which is very small so has little impact. Also, the RadixSort runs after the $old_row_ptrs[\,]$ array has been deallocated after converting the matrix to the HyperPartition format, thus in many cases the extra 8 KiB does not increase the memory overhead at all.

Figure 7.4 shows the serial execution time for the HyperPartition algorithm with RadixSort. The HyperPartition transpose was again performed using a remaining bits heuristic value of $k = 9$ and the RadixSort using $b = 256$ buckets ($radix = 8$). HyperPartition performs much better than Saad for the majority of inputs. As discussed previously, there are a number

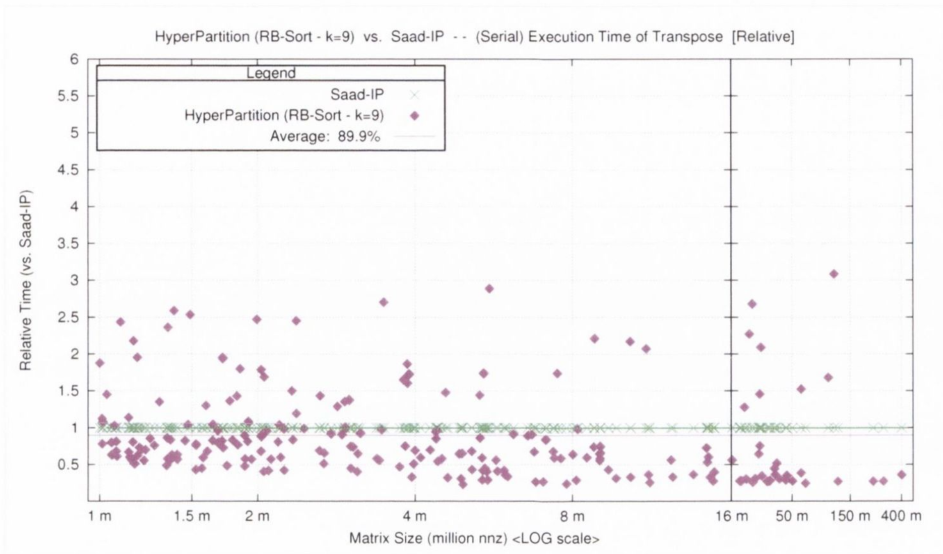


Figure 7.4: Serial execution time of HyperPartition Transpose ($k = 9$) with RadixSort ($B = 256$) compared to Saad. The RadixSort improves the execution time of HyperPartition compared to QuickSort. HyperPartition with RadixSort is significantly faster than Saad for a large number of inputs, however as certain matrices cause problems (see Section 7.4).

of input matrices that are structurally symmetric for which HyperPartition does not perform as well on as the Saad and Corresponding Row algorithms. These structurally symmetric matrices will be examined further in Section 7.4.

Figure 7.4 shows the serial execution time of the HyperPartition algorithm using the RadixSort algorithm from Algorithm 7.1 in Section 7.1 for performing the *Phase-II* sorting step of the Sparse In-Place Transpose. The execution time of the algorithm is still quite variable depending on inputs, however there is a distinct improvement over the HyperPartition with QuickSort with the performance significantly improving for nearly all input matrices. There are still some matrices slower than Saad, but the majority are faster with a large proportion running more than 50% faster.

One point to note for the results in this Section, the sorting phase of the HyperPartition in-place transpose is now different to that of the Saad and Corresponding Row algorithms. HyperPartition is using RadixSort (Algo-

rithm 7.1) and Saad is still using QuickSort (Algorithm 4.7). With Saad-IP and Corresponding Row the sorting phase is only a small proportion of the total execution time (typically about 5%) also, there are a large number of small rows which do not benefit from the RadixSort algorithm. However the graphs still show the total execution time of the full transpose operation both including cycle chase and sorting, so results are still comparable.

7.2.1 Parallel HyperPartition with RadixSort Performance

As we discussed in Section 7.1, a huge benefit of performing more of the transpose in the second sorting phase is that all the sorting operations just process a single row (or HyperPartition). This means we can perform this second phase in parallel.

Figure 7.5 shows the memory usage of the Parallel HyperPartition transpose with RadixSort running on 32 cores. The parallel HyperPartition transpose was again performed using a remaining bits heuristic value of $k = 6$ which proved a good value for Parallel HyperPartition with QuickSort. This value again proved a good value for parallel HyperPartition with RadixSort. The RadixSort again used $b = 256$ buckets ($radix = 8$). In this case the memory overhead of the RadixSort is a lot larger because each of the 32 process threads requires its own set of buckets. Each thread has an overhead of 8 KiB which is 256 KiB overall. The RadixSort runs after the $old_row_ptrs[\,]$ array has been deallocated so for many input matrices which have a large $nrows$ the 256 KiB does not increase the memory overhead.

As can be see from Figure 7.5, the 32-Way Parallel HyperPartition with RadixSort does increase the memory overhead compared to Serial HyperPartition with RadixSort in Figure 7.3 and HyperPartition with QuickSort in Figure 6.6. In Figure 7.6 we can see a close-up of the memory overhead of the algorithm. Here we can see that the overhead of using RadixSort in parallel on 32 cores does slightly increase the memory overhead for some of the smaller matrices. There is a dotted line to indicate 5% of the memory overhead of the Saad algorithm, for some input matrices the

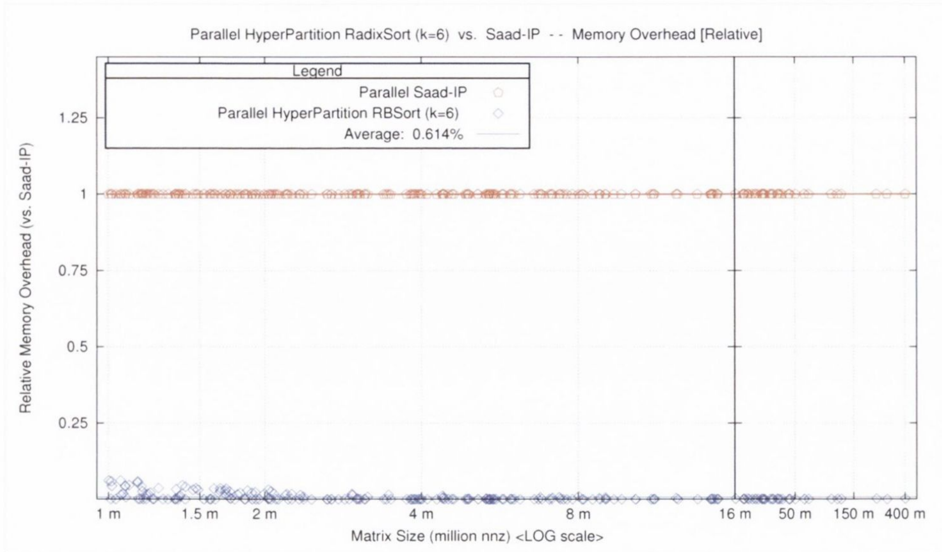


Figure 7.5: Memory usage of 32-Way Parallel HyperPartition with RadixSort. HyperPartition was run with $k = 6$ and the RadixSort was run with $b = 256$ buckets. The memory overhead for the algorithm when transposing the smaller matrices has increased slightly but the overhead is much less than Saad with an average of 0.6%.

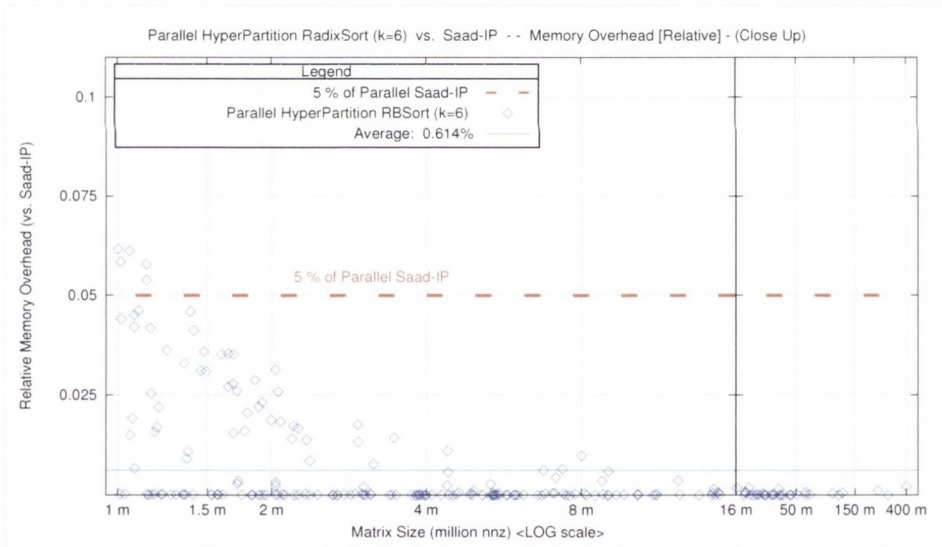


Figure 7.6: Close-Up of Figure 7.5. A dotted line shows 5% of the memory usage of Saad. RadixSort slightly increases relative memory overhead for some of the smaller matrices.

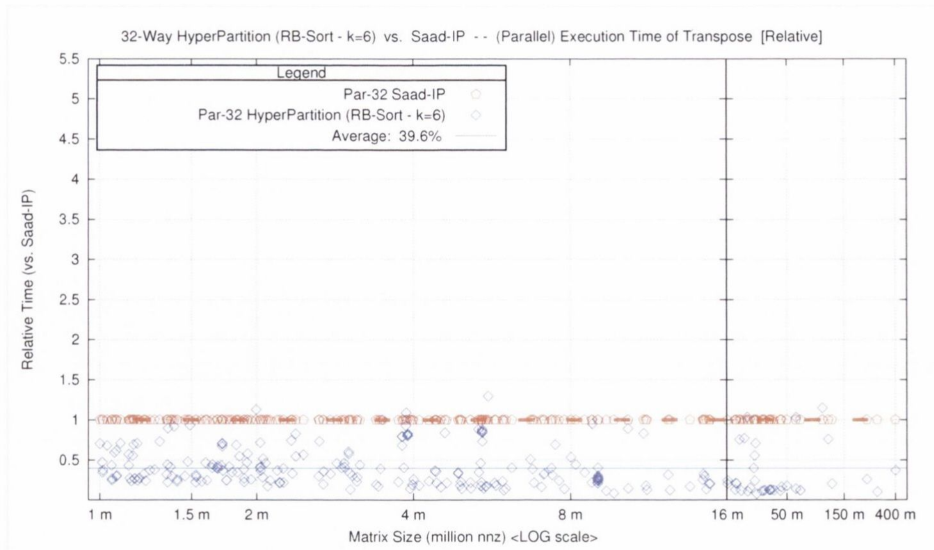


Figure 7.7: 32-Way Parallel execution time of HyperPartition Transpose ($K = 6$) with RadixSort compared to Saad. This shows the parallel performance improvement exploited by HyperPartition with RadixSort over the serial version in Figure 7.4. Performance is considerably improved with only 3 inputs running slightly slower and the majority less than 50% and many less than 25% of Saad with an average of 39.6% of Saad. This compares well with the Out-of-Place algorithm in Figure 3.3. Note: Saad QuickSort also run in 32-way parallel.

HyperPartition with RadixSort approaches and in a few cases surpasses this level. The average memory overhead for the 259 matrices in the collection is still just 0.6% of the memory overhead of Saad.

The RadixSort step is performed in parallel using *OpenMP* using all 32 cores of the 'Stoker' machine described in Sections 2.4 and 3.7.

Figure 7.7 shows the execution time of the Parallel HyperPartition in-place transpose with RadixSort compared to the *Parallel* version of Saad-IP. It is more appropriate to compare Parallel HyperPartition to Parallel Saad-IP. The parallel version of Saad-IP performs slightly faster than the serial version, however the improvement is minor. Saad does not benefit from RadixSort as most of the work of the Saad algorithm is in the original cycle-chasing transpose and the resulting partitions are too small to benefit from RadixSort. The HyperPartition transpose was performed

using a remaining bits heuristic value of $k = 6$ which proved a good value for Parallel HyperPartition with QuickSort. The RadixSort again used $b = 256$ buckets ($radix = 8$).

Figure 7.7 clearly shows the benefit of using the RadixSort in parallel. While using only 0.6% of the memory overhead of Saad-IP, the 32-way parallel HyperPartition with RadixSort performs better than Saad-IP for nearly all input matrices. For the majority of matrices the parallel HyperPartition performs the in-place transpose in less than 50% of the execution time of Saad-IP with an average of 39.6% of the execution time of Saad for the 259 matrices in the test suite.

From this figure we can see that the HyperPartition with Parallel Radix sort performs more than 50% faster than Saad for the majority of the input matrices, and indeed a large proportion run in less than 25% of the execution time of Saad. A very satisfactory result for a complicated three step algorithm which has only a tiny fraction of the memory overhead of the existing Saad algorithm.

7.3 MSD RadixSort Summary

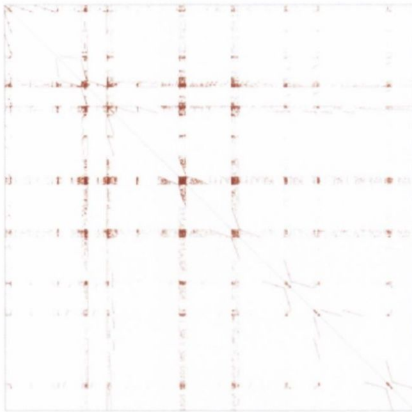
In Section 7.1 we presented our variant of the Most Significant Digit In-Place RadixSort algorithm for sorting HyperPartitions which improves the efficiency of the second sorting phase of the HyperPartition transpose algorithm. Performance analysis in Section 7.2 shows that the MSD RadixSort gives a significant improvement over the QuickSort/InsertionSort algorithm presented earlier in Section 4.5, reducing the execution time of the serial HyperPartition transpose from 115% of Saad to 90% of Saad using less than 1% of the memory overhead. The MSD RadixSort also improved the parallel HyperPartition transpose from 42.7% to 39.6% of the execution time of Saad. Again with less than 1% of the memory overhead.

7.4 Structural Analysis

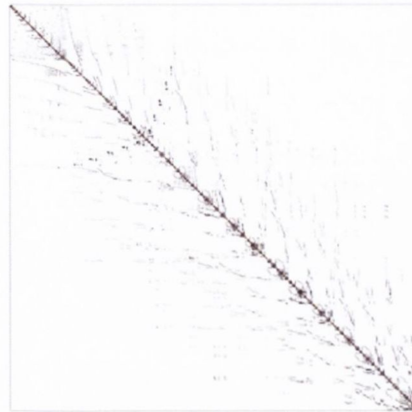
In Section 5.11 we found that certain matrices had cycles which were no longer than two elements long. This resulted in excellent cache usage, and hence performance when transposing these matrices. Analysing the structure of these matrices we find that the reason that these 51 matrices have a maximum cycle length of two and exhibit such good cache performance is that they are structurally symmetric. A symmetric matrix is a matrix that is identical to it's own transpose ($M = M^T$), where all the elements mirrored through the diagonal have the same value. `i.e.` $M_{i,j} = M_{j,i} \mid \forall_{i,j} \in n, m$. Generally when a matrix is truly symmetric, only one triangle of the matrix is stored to save on space. The other triangle can be obtained by accessing the first triangle in reverse. As outlined in Section 3.6.2 we have included a number of symmetric matrices in our test suite to increase the number of matrices, however as transposing a symmetric matrix is pointless we are just using them as triangular matrices. Some of these triangular matrices can be seen in Figure 7.11.

In Section 6.8 we then found that the HyperPartition transpose did not perform as well as the Saad or Corresponding Row algorithms for these structurally symmetric matrices. This can be seen in Figure 6.8 on page 165. It is not that HyperPartition performs poorly for these matrices, it is just that Saad and Corresponding Row perform unexpectedly well due to the short cycle lengths. When converted to the HyperPartition format the matrices are no longer in a structurally symmetric layout, hence the drop in performance. HyperPartition looses the benefit of structural symmetry because it groups rows into HyperPartitions so it is no longer moving elements to the location of their corresponding opposite element.

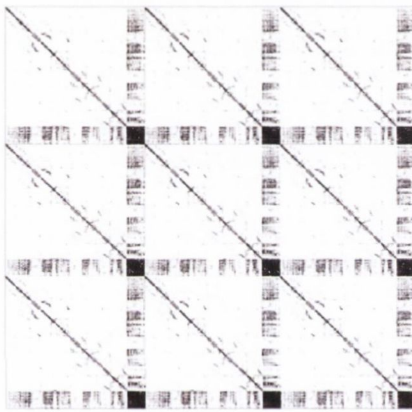
Figure 7.9 shows the execution time of the HyperPartition algorithm with QuickSort with $k = 9$ compared to the execution time of the Saad-IP algorithm. The graph shows the execution time of the algorithms transposing *just* the 51 structurally symmetric matrices in the test suite, the timings for the other 208 matrices are omitted. The HyperPartition transpose clearly does not perform as well as Saad for these types of matrices



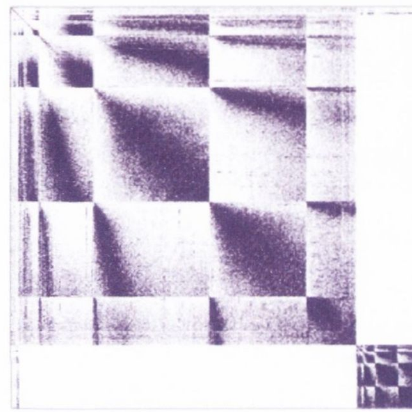
(a) ASIC_680k



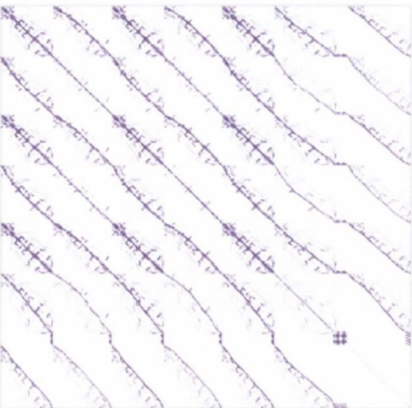
(b) cage14



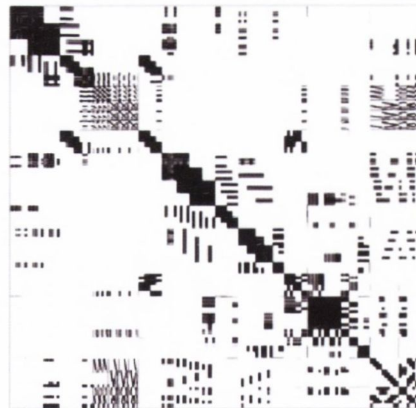
(c) para-10



(d) thermomech_dK



(e) mixtank-new



(f) heart1

Figure 7.8: Some Structurally Symmetric Matrices

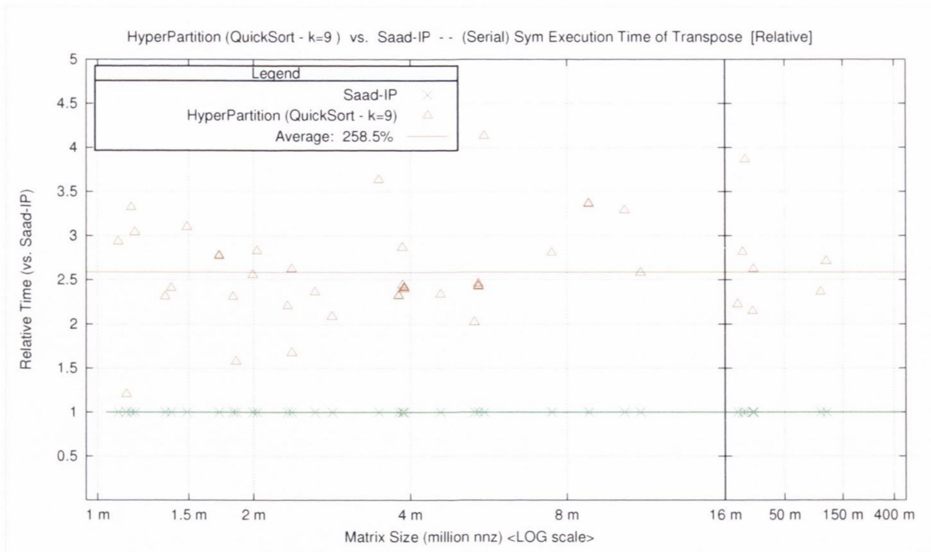
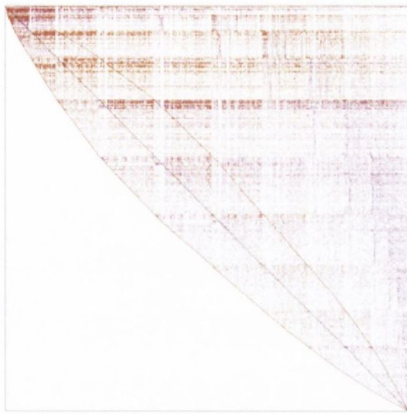


Figure 7.9: Just Structurally Symmetric — The Serial execution time of the Hyper-Partition transpose with QuickSort with $K = 9$. Only the relative execution time of the algorithms for transposing structurally symmetric matrices is shown. HyperPartition performs poorly compared to Saad for these matrices.

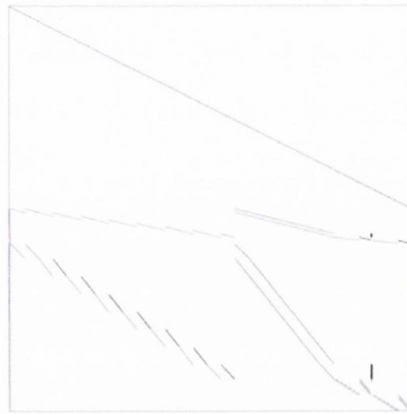
with the average execution time for these 51 matrices being 258% of the execution time of Saad.

The 51 matrices that are causing problem for HyperPartition are not symmetric (or triangular), they are structurally symmetric. They have the same structural layout of the elements in a symmetric matrix with elements in the same locations mirrored through the diagonal, however in these matrices the opposite elements through the diagonal do not have the same value. Therefore they are not symmetric. Some structurally symmetric matrices used in the test suite are show in Figure 7.8 and some general unsymmetric matrices are shown in Figure 7.10. It is the structural symmetry of these matrices which results in cycles which are only two elements long which gives much better temporal and spatial locality to memory lookups, which in turn greatly improves execution time of Saad and Corresponding Row.

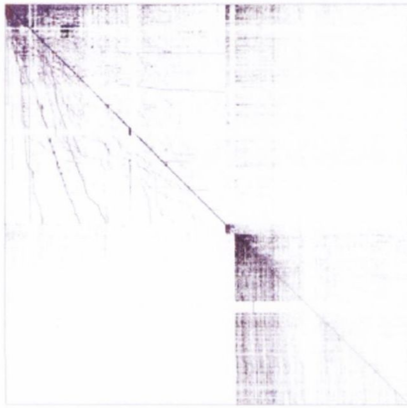
Analysis of the runtime performance of the algorithms using hardware counters in Section 5.9 showed that for these matrices the Saad and



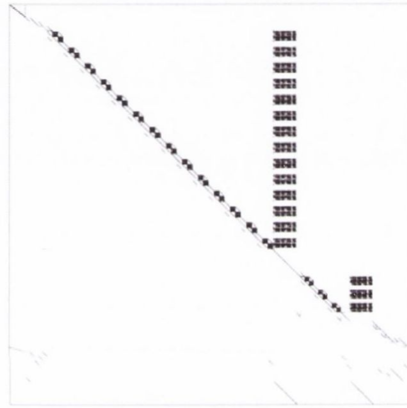
(a) language



(b) Hamrle3

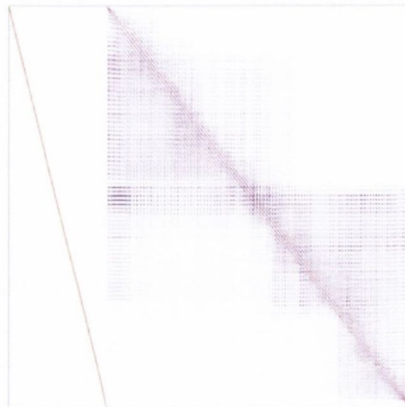


(c) webbase_1M



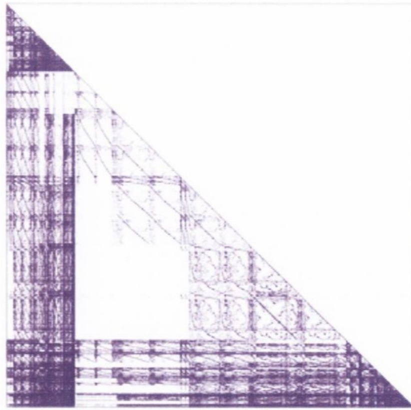
(d) Zd-Jac3

(e) invextr1_new

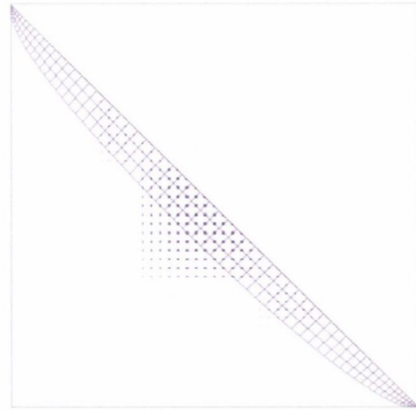


(f) av41092

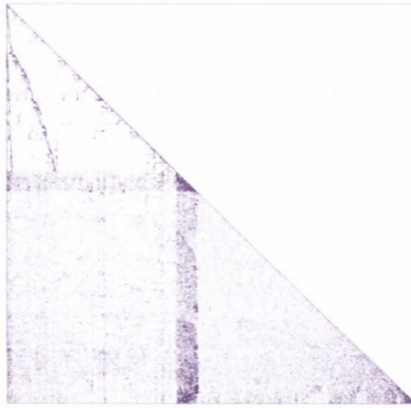
Figure 7.10: Some Unsymmetric Matrices



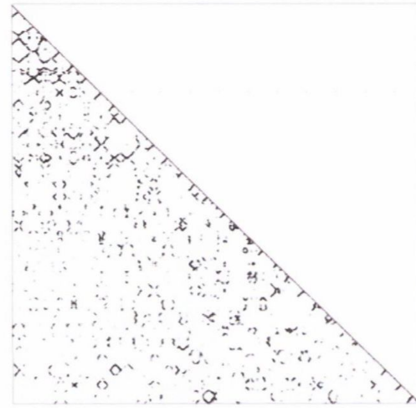
(a) gsm_106857



(b) SiO2

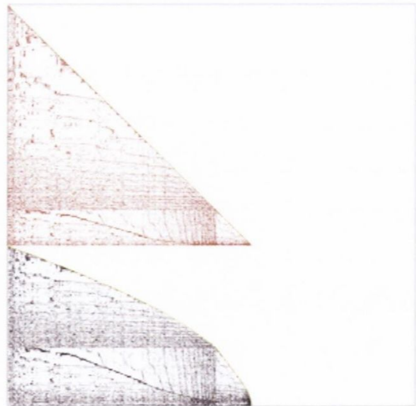


(c) F2



(d) nd3k

(e) filter3D

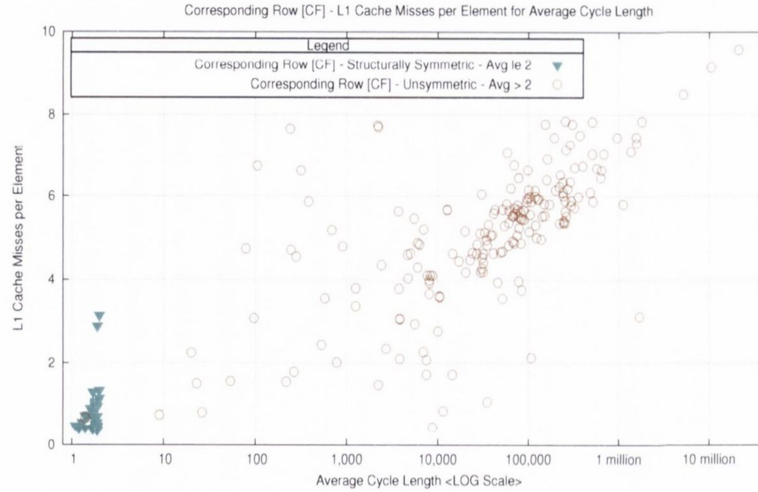


(f) darcy003

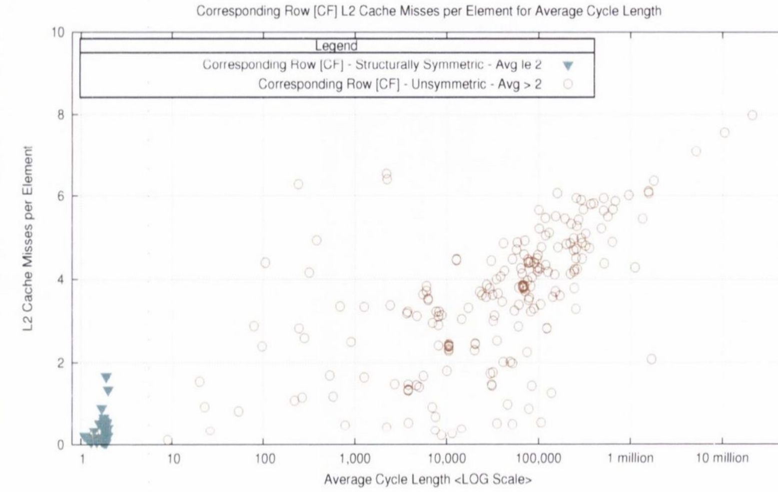
Figure 7.11: Some Triangular Matrices - Lower Triangle of Symmetric

Corresponding row algorithms have surprisingly good cache performance compared to matrices of comparable size. Deeper analysis of the internal operation of the algorithms as they process the matrices showed an interesting feature of 51 of these "slow" matrices. For these 51 matrices the average cycle length of all the cycles when performing the cycle chasing transpose with Saad and Corresponding row is just two elements. Indeed, for these 51 matrices the maximum cycle length of all the chains chased is just two elements. Every time we start a cycle, the first element that we jump to just jumps straight back to the source row that we started in. What's more, the element actually jumps to the exact position that it should be in in the transposed matrix meaning that the Phase-II sorting step is not required for these matrices. These short cycle lengths are the reason for the good cache performance of Saad and Corresponding Row. We know from Section 5.10 that shorter cycles lead to an improvement in cache performance.

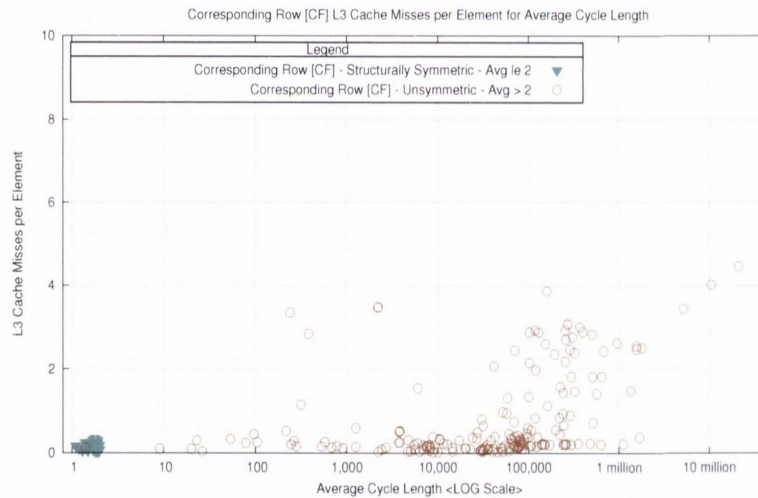
Figure 7.12 shows the L1, L2, L3 cache and TLB performance of the structurally symmetric matrices compared to the unsymmetric matrices while transposing the matrix with the Corresponding Row algorithm. Each of the four graphs shows the relative number of cache/TLB misses per non-zero element in the matrix relative to the average cycle length. The structurally symmetric matrices can be seen in the left of each graph as they have average cycle lengths between one and two elements long. It is clear from these graphs that the structurally symmetric matrices incur far less L1, L2 and TLB misses than the vast majority of the other matrices in the test suite. This means that the Corresponding Row algorithm has much better cache performance when transposing these structurally symmetric matrices. There is less of a difference in terms of L3 misses as for most input matrices the algorithm incurs relatively few L3 misses, however the structurally symmetric matrices once again occupy the lower end of the scale.



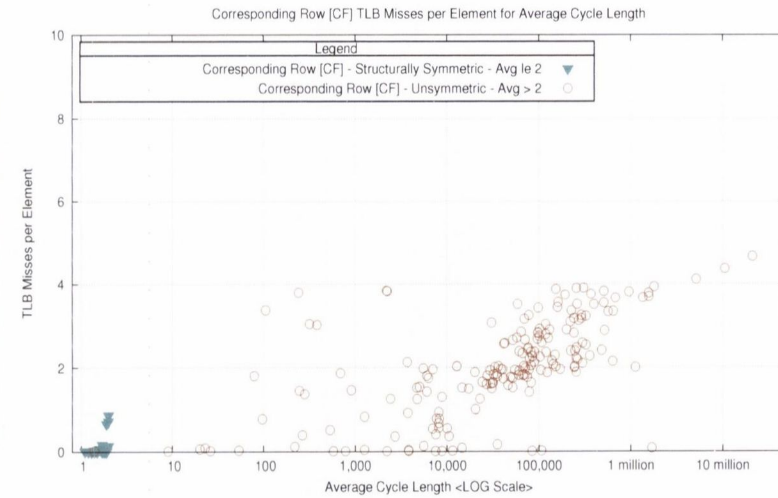
(a) L1 Cache Misses per Non-Zero Element



(b) L2 Cache Misses per Non-Zero Element



(c) L3 Cache Misses per Non-Zero Element



(d) TLB Misses per Non-Zero Element

Figure 7.12: [CF] Corresponding Row : Relative Cache Performance of structurally symmetric matrices compared to unsymmetric matrices in relation to Average Cycle Length

7.4.1 Detecting Structural Symmetry

There is an accurate (and computationally expensive) method to determine if a matrix is structurally symmetric. First, the matrix needs to be square, i.e. $m = n$ or indeed $nrows = ncols$. We then check if all the Corresponding Rows and columns have the same pattern of non-zeros. To do this, we make a copy of the $old_col_indexes[\,]$ array and transpose the matrix. We then compare every index in $old_col_indexes[\,]$ array with every corresponding entry in $new_col_indexes[\,]$ array. If every entry is the same, such that the two arrays are identical, then we know that the matrix has the identical pattern of non-zero elements in every Corresponding Row and column in the matrix. Meaning that it is structurally symmetric.

We analysed all of the matrices in our test suite using this "Expensive Test" method. Out of the 259 matrices we found that 51 of the matrices are structurally symmetric. The very same 51 matrices with the two-element cycle chains that caused problems for HyperPartition. No other matrices were found to be structurally symmetric. Comparing the performance of HyperPartition compared to Saad for these 51 matrices as shown in Figure 7.9 we find that HyperPartition is slower than Saad for every one of the 51 matrices. We also compared the execution time of Corresponding Row compared to Saad as shown in Figure 5.3 for these 51 matrices and found that Corresponding Row performs better in all cases. Also comparing the memory usage of Corresponding Row with Saad as in Figure 5.1 shows that Corresponding Row also uses less memory than Saad for all of the 51 matrices.

This means that if we had a simple method of determining structural symmetry then we could have a hybrid algorithm that could choose between the HyperPartition and Corresponding Row algorithms depending on the structural symmetry of the input matrix in order to give better performance.

7.5 Hybrid HyperPartition Transpose Algorithm

This gives the opportunity for a Hybrid algorithm which will choose either to run the Corresponding Row algorithm if the input matrix is structurally symmetric, otherwise run the HyperPartition algorithm.

The method outlined above for testing if a matrix is structurally symmetric is too expensive to use. There is however a quick test that we can use that is very inexpensive and is actually quite accurate in practice. We check if the matrix is square ($nrows = ncols$) and then simply compare the number of elements in every row with every corresponding column. If every row has the same number of elements as its opposite column then it is possible that the matrix is structurally symmetric. At the start of our generic in-place cycle-chasing transpose we need to construct an array of $new_row_ptrs[\,]$ to point to the start of every new row in the matrix. While calculating the cumulative sum on this array we can compare all the new row pointers in this array with the old row pointers in the $old_row_ptrs[\,]$ array. If every pointer for each new/old row is the same in both arrays then we know that each row in the matrix has the same number of elements as its corresponding column. Where we have such a matrix then there is a strong probability that the matrix is structurally symmetric.

We used this quick test on the 259 matrices in our collection. The test accurately identified all of the 51 structurally symmetric matrices without a single false positive. The test is very quick and inexpensive. It can be performed in $\Theta(n)$ time while building the $new_col_ptrs[\,]$ array which we needs to be done anyway. As such the overhead of the check is negligible. Algorithm 7.2 shows the pseudo code for the simple test algorithm used by the Hybrid HyperPartition Transpose.

ALGORITHM 7.2: Detect Structural Symmetry Heuristic

```
1  /* Count number of elements in each new column - offset by 1 */
2  for ( 0 ≤ index < nnz ) do
3  |   col ← col_indexes[index];
4  |   if ( col ≤ (nrows − 1) ) then
5  |   |   new_row_ptrs[col + 1] ← new_row_ptrs[col + 1] + 1;
6  |   end
7  end
8  /* Cumulative sum to get new_row_ptrs[] */
9  /* Check if Structurally Symmetric */
10 isSymm ← true;
11 for ( 0 ≤ row ≤ nrows ) do
12 |   new_row_ptrs[row] ← new_row_ptrs[row] + new_row_ptrs[row − 1];
13 |   row_offsets[row] ← new_row_ptrs[row] ;        /* Copy to row_offsets[] */
14 |   if (new_row_ptrs[row] ≠ old_row_ptrs[row]) then
15 |   |   isSym ← false;
16 |   end
17 end
```

7.6 Hybrid HyperPartition Transpose Performance

The Hybrid algorithm increases the memory requirement over that of HyperPartition. There is a trade-off between memory and performance. As we see in Figure 6.6 in the previous chapter, the memory overhead of HyperPartition is negligible. As such, HyperPartition is still the best choice of transpose algorithm for very low memory usage and although it is very efficient for the majority of input matrices, the low memory feature does however come at the cost of poor performance of some (structurally symmetric) matrices.

Figure 7.13 shows the memory usage of the Hybrid HyperPartition in-place transpose algorithm. The Hybrid algorithm requires more memory than the HyperPartition algorithm because in order to check for possible structural symmetry the algorithm needs to build the *new_row_ptrs*[ ] array before the *old_row_ptrs*[ ] array has been deallocated. Thus the algorithm needs $\Theta(n)$ memory overhead for the symmetry check. The HyperPartition algorithm avoids this $\Theta(n)$ by deallocating the *old_row_ptrs*[ ] array, per-

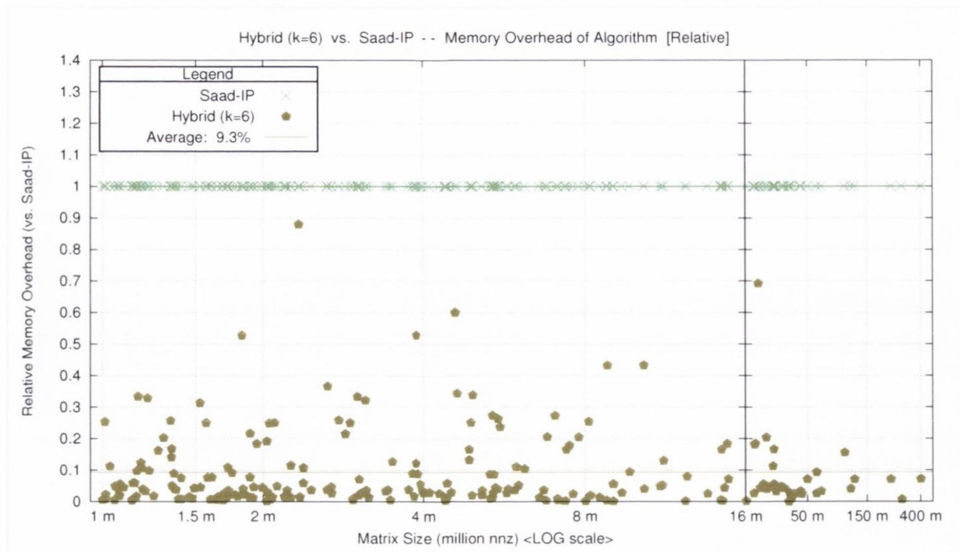


Figure 7.13: Memory overhead of the Hybrid HyperPartition algorithm compared to that of Saad. The Hybrid algorithm uses more memory than HyperPartition however it still requires considerably less memory than Saad with most inputs requiring less than 10-20% than that of Saad with an average of 9.3%.

forming the cycle-chasing and sorting algorithms and only then building the *new_row_ptrs*[ ] array at the end. This is why the memory overhead of HyperPartition is so low.

The Hybrid algorithm uses the (*new_row_ptrs*[ ]) array of size n to check for symmetry. If the matrix is symmetric the Corresponding Row algorithm is used to transpose the matrix, requiring a total of 3 arrays of size n (*row_offsets*[ ] and *corresp_table*[ ] in addition to the *new_row_ptrs*[ ] array). Otherwise the HyperPartition algorithm is used which has minimal additional memory requirements. Thus the Hybrid algorithm requires less memory than the Corresponding Row algorithm and much less than the Saad algorithm, only requiring 9.3% of the memory overhead of Saad on average for the 259 matrices in the test suite.

Figure 7.14 shows the execution time of the Hybrid HyperPartition algorithm with RadixSort with a remaining bits heuristic value of $k = 9$. As can be seen comparing this graph to Figure 7.4, the 51 structurally symmetric matrices for which HyperPartition did not perform so well on

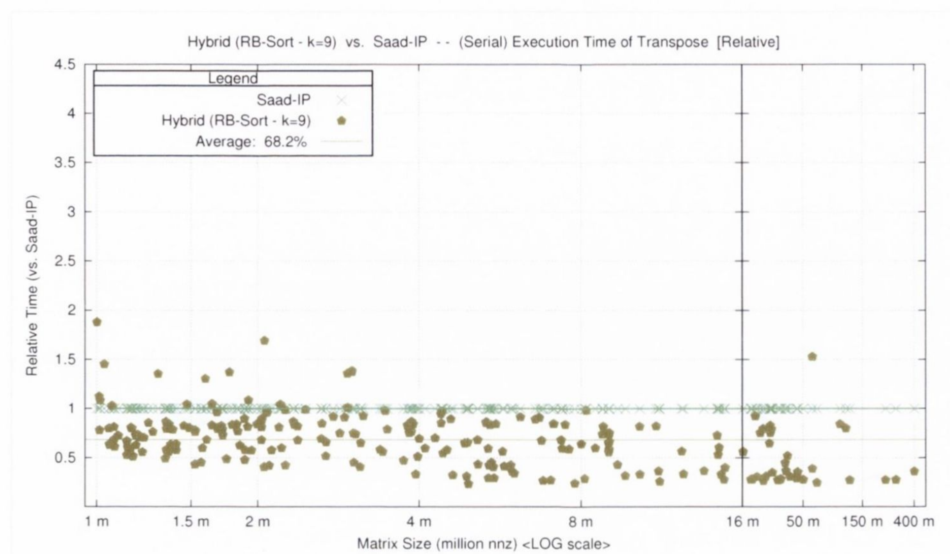


Figure 7.14: Serial execution time of the Hybrid HyperPartition $k = 9$ transpose algorithm relative to Saad-IP. The Hybrid algorithm uses Corresponding Row for structurally symmetric matrices, otherwise HyperPartition. For these matrices, Corresponding Row is faster than both HyperPartition and Saad making this Hybrid algorithm not only competitive with Saad but considerably faster for the vast majority of inputs with much less memory usage. The Serial Hybrid HyperPartition with RadixSort has an execution time of 68% of Saad on average.

are now all below the line. There are a small number of matrices for which the Hybrid algorithm does not perform as well as Saad however for the majority of input matrices the Hybrid algorithm performs better than Saad, and in some cases performs much better with the algorithm requiring less than 50% of the execution time of Saad for a number of matrices. On average the Hybrid HyperPartition algorithm transposes the 259 matrices from the test suite in 68.2% of the execution time of Saad.

7.6.1 Parallel Hybrid HyperPartition Performance

Just as with the HyperPartition algorithm, the Hybrid HyperPartition algorithm performs a large proportion of its work (for non structurally symmetric matrices) during the sorting phase of the transpose. This sorting phase, which we have improved the performance of with the RadixSort in

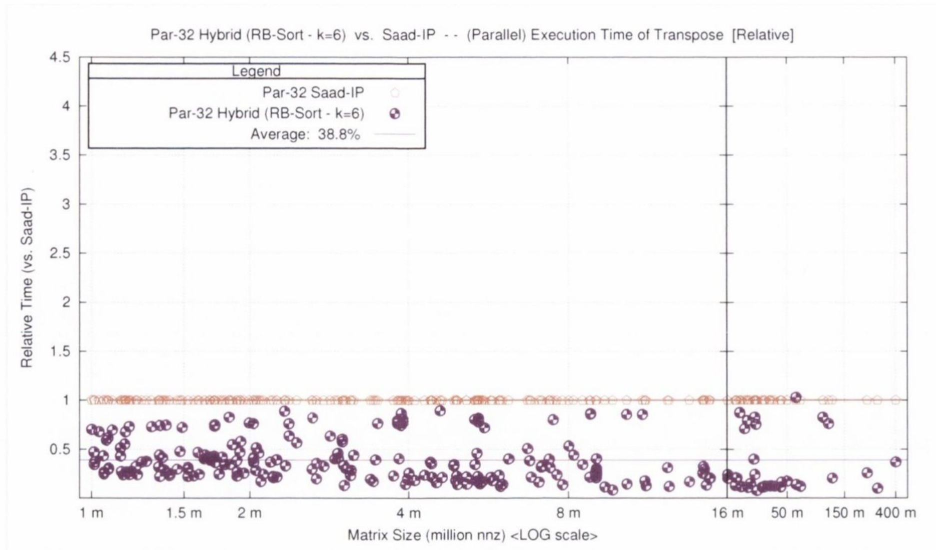


Figure 7.15: 32-Way OpenMP Parallel execution time of Hybrid Transpose (with HyperPartition of $K = 6$ with RadixSort) compared to Saad. This shows the parallel performance improvement exploited by HyperPartition with RadixSort over the serial version in Figure 7.14. Performance is considerably improved with the majority of inputs running less than 50% and many less than 25% of Saad. This compares well with the execution time of the high memory OOP Algorithm in Figure 3.4. Note: Saad QuickSort also run in 32-way parallel.

Section 7.1 can benefit greatly from parallel execution.

The parallel Hybrid HyperPartition in-place transpose with RadixSort is shown in Figure 7.15. Again a remaining bits heuristic value of $k = 6$ is used for parallel execution along with $b = 256$ buckets for the sorting phase which was run in parallel across 32 cores. This graph shows that the parallel Hybrid algorithm performs much better than the Parallel Saad-IP Algorithm. The Hybrid is faster than Saad on all bar one of the input matrices which is just a fraction slower. The parallel Hybrid algorithm transposes the majority of the input matrices in less than 50% of the time of Saad and in less than 25% for many. On average the Parallel Hybrid algorithm transposes the 259 matrices in the test suite in 38.8% of the time of Saad while only requiring 9.3% of the memory.

Figure 7.15 showed the performance of the parallel Hybrid transpose

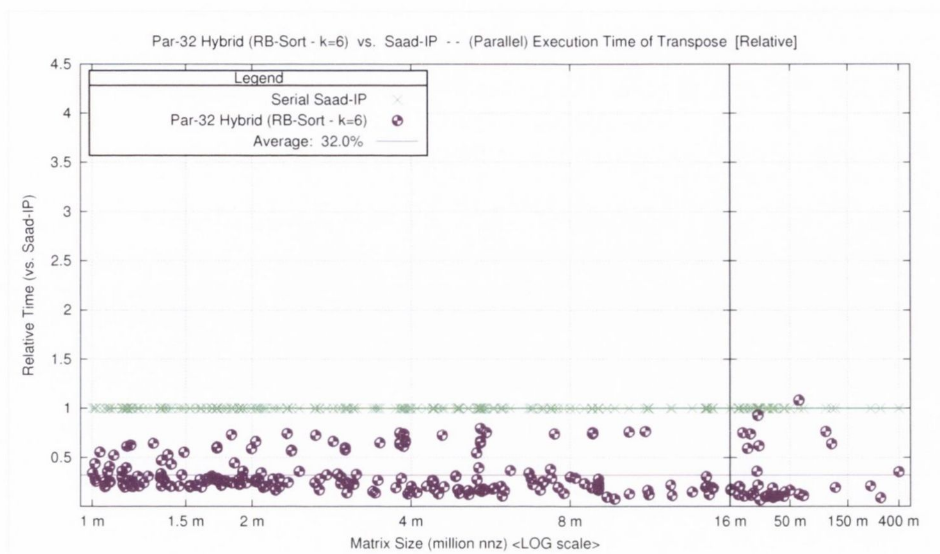


Figure 7.16: Execution Time of Parallel Hybrid HyperPartition transpose relative to that of *Serial* Saad-IP.

compared to the Parallel Saad-IP transpose. Figure 7.16 shows the performance of the parallel Hybrid transpose compared to the *Serial* Saad-IP transpose. We can see from this graph that the parallel Hybrid algorithm is much faster than the original serial Saad algorithm with the majority of the input matrices taking less than 50% of the time of the Serial Saad algorithm to transpose. On average the parallel Hybrid takes 32% of the time of the Serial Saad-IP algorithm to transpose the 259 matrices in the test suite while only requiring 9.3% of the memory overhead of Saad.

7.7 Hybrid HyperPartition Summary

We have introduced the Hybrid HyperPartition in-place transpose algorithm which uses a "Quick Test" to check for possible structural symmetry in an input matrix and uses this to decide to transpose the matrix with the Corresponding Row algorithm or the HyperPartition algorithm. At the cost of this test which is very quick and a small increase in memory to 9.3% of Saad we have improved the execution time of the Hybrid transpose to

68.2% of Saad in serial and 38.8% in Parallel.

There are still a small number of matrices for which the Hybrid HyperPartition algorithm does not perform quite as well as Saad. It is probable that for these matrices there is some aspect of the structure of the matrix which allows Saad to perform better, which we lose when we switch the matrix to the HyperPartition format. Further investigation of these matrices may also produce a simple test which could allow further improvement to the hybrid algorithm.

It is possible to improve the Hybrid algorithm some more. When we transpose a matrix that is actually structurally symmetric it not necessary to perform the sorting step of the transpose and elements will be moved into their correct position. Our "Quick Test" does not give a guarantee of structural symmetry, however if we think the matrix might be symmetrical the addition of a small check during the Corresponding Row algorithm that all cycles are no more than 2 elements long would indicate a symmetrical matrix and the unnecessary sorting step can be avoided.

Chapter 8

Conclusion and Future work

Sparse Matrix Transpose is an important linear algebra operation which occurs in calculations of numerous applications such as Fast Fourier Transforms, Computational Chemistry, Signal Processing and many others. In particular, the sparse transpose is important for converting a matrix from the row-major CSR storage format to the column-major CSC storage format (and vice versa) to improve the efficiency of accessing the matrix by columns (rows).

The transpose of a sparse matrix stored in a sparse storage format such as CSR can be complicated and difficult to optimise. In order to perform the transpose, the two existing sparse matrix algorithms, the out-of-place algorithm and the Saad In-Place algorithm both require additional storage in memory proportional to the number of non-zeros in the matrix. $\Theta(nnz + n)$ for the out-of-place and $\Theta(nnz)$ for the Saad-IP transpose. This translates to roughly 100% and 30% respectively of the size of the matrix in additional memory overhead for the algorithms which is quite considerable.

We have introduced a collection of novel space and time saving in-place sparse matrix transpose algorithms which were evaluated with extensive experimentation and analysis using a test suite of 259 large matrices from real world applications. The algorithms use a variant of the in-place cycle chasing algorithm similar to Saad yet only require $\Theta(n)$ temporary storage for an $n \times n$ sparse matrix with $n << nnz$ non-zeros, compared to the $\Theta(nnz)$ and $\Theta(nnz + n)$ required for Saad and out-of-place. Moreover, our algorithms are able to achieve this large savings in space complexity without forfeiting the $\Theta(nnz + n)$ execution time complexity of the existing algorithms. Finally, our algorithms are able to handle non square sparse matrices and matrices with empty rows and columns. Custom transpose

algorithms for specific matrix layouts could potentially be more efficient however our algorithms provide good performance across all matrix types.

8.1 Contributions

We describe (Chapter 4) the three problems associated with reducing the space complexity of the in-place transpose to $\Theta(n)$ memory overhead. Our Generic in-place algorithm solves problems (b) recording that an element has been moved and (c) finding the location of the next free slot in a row. We present two initial solutions to problem (a) finding the old row index. Binary Range Search and Radix Table which both have a space complexity of $\Theta(n)$ and a memory overhead of just 14% and 16% of Saad respectively. Experimental analysis shows that the execution time of the Radix Table transpose is broadly similar to Saad despite the increased time complexity, actually running at 98.6% of the execution time of Saad on average.

We introduced (Chapter 5) our Corresponding Row transpose algorithm that solved problem (a) reducing the memory overhead of the in-place sparse transpose to $\Theta(n)$ while maintaining the time complexity of $\Theta(nnz + n)$. A lookup table is used which can be searched and updated in amortized $\mathcal{O}(1)$ time. Two implementations of our Corresponding Row algorithm reduce the memory overhead of the transpose to 21% of Saad on average. The Corresponding Row algorithms also run faster than Saad. The Normal version runs at 92% of the execution time of Saad on average and the Cache Friendly version improves on this by 2% at 90% of Saad on average. Results of detailed experimental analysis using hardware counters are presented. These results show that the Corresponding Row algorithms run faster as they have fewer Cache and TLB misses than Saad. The Cache Friendly implementation is faster than the Normal version because of improved L1 and L2 cache reuse from the interleaved arrays. Analysis of these experimental results also demonstrates that the average cycle length has a direct influence on the cache performance of the in-place cycle chasing transpose. Transposing a matrix with a larger number of shorter cycles results in better cache reuse and reduced execution time.

Taking the knowledge that shorter cycles results in improved cache performance we developed our HyperPartition matrix storage format (Chapter 6). The HyperPartition format is a modification on the CSR storage format which allows us to groups rows together into HyperPartitions by exploiting unused data in the CSR format. Performing the cycle chasing in-place transpose on a matrix in HyperPartition format reduces the average cycle length in most cases, thus improving cache performance and reducing execution time. We present our *Remaining Bits* heuristic for calculating the number of bits to steal when converting to HyperPartition CSR that will give reasonable performance for most matrix types. We recommend values of $k = 9$ or $k = 10$ in serial and $k = 5$ or $k = 6$ in parallel for the heuristic parameter based on the matrices in our test suite. The HyperPartition transpose also has a space complexity of $\Theta(n)$ however in practice as it drastically reduces the memory overhead to less than 1% of Saad for the matrices in our test suite. For the majority of matrices, HyperPartition runs faster than Saad with some of the larger matrices taking less than 50% of the time.

HyperPartition transpose performs a higher proportion of the work during the sorting phase of the transpose. Therefore, we present (Chapter 7) a more efficient HyperPartition sorting algorithm based on the Most Significant Digit RadixSort algorithm. The complexity of the RadixSort is $\mathcal{O}(nnz\,.\,p)$ where p is proportional to the number of bits in the index integers divided by the radix length and remains constant for any given radix length. Using the MSD RadixSort we improve on the execution time of the serial HyperPartition transpose from 115% of Saad when using QuickSort to 90% of Saad with RadixSort. In parallel on 32 cores the HyperPartition transpose improves from 42.7% of parallel Saad with QuickSort to 39.6% with RadixSort. There is a small increase in memory overhead for the RadixSort. When using a radix of 8 (256 buckets) the average memory overhead remains less than 1% of Saad. In parallel each thread requires two small work arrays. This results in a slight increase up to 5% of Saad for a few input matrices, however the average remains at 0.6%.

We present results (Chapter 7) of structural analysis of the input ma-

trices and transpose algorithms which show that HyperPartition transpose does not perform as well as Saad and Corresponding Row on matrices which are structurally symmetric. Converting to HyperPartition CSR format loses the benefit of the structural symmetry. We present a heuristic algorithm that can quickly and easily detect if a matrix in CSR format is *probably* structurally symmetric. We present a Hybrid HyperPartition transpose which uses this heuristic algorithm to allow the HyperPartition transpose to efficiently handle structurally symmetric matrices. The serial Hybrid HyperPartition runs at 68% of the execution time of Saad on average at cost of a slight $\Theta(n)$ increase in memory overhead to 9.3% of Saad. In Parallel the Hybrid HyperPartition transposes the matrices in 38.8% of the time of Parallel Saad and 32% of the time of Serial Saad.

8.2 Future Work

There are a number of possible areas where this work could be extended.

Incorporating the algorithms presented here into a publicly available sparse matrix package such as the Sparse Blas or BeBOP Sparse Matrix Converter would be an easy way of making them available to the Linear Algebra Community so they can benefit from their improved efficiency.

In Section 6.6.1 we presented our Remaining Bits heuristic for choosing the number of bits to steal when converting a matrix in CSR format to HyperPartition CSR format. This heuristic allows us to choose a number of bits to steal from each matrix which gives us an improved average performance for the matrices in the test suite at values of $k = 9$ in serial and $k = 6$ in parallel. However, experimental analysis shows that while changing the k parameter of the heuristic improves performance of the HyperPartition algorithm for some matrices, for other matrices the performance degrades. A better alternative would be to: First, individually examine the performance of the RadixSort algorithm for different HyperPartition sizes in serial and parallel. Secondly, examine the performance of the HyperPartition cycle-chasing algorithm stealing different numbers of bits from different matrix sizes. Then, combining the results of the two experiments to develop a

heuristic which for each individual matrix would choose the best number of bits to steal based on its dimensions and sparsity.

In Section 7.1.2 we recommended a radix size of 8 which gives 256 buckets at each level of the RadixSort as this gives a reasonable performance of the RadixSort on average for the matrices in our test suite. A heuristic which was based on the dimensions of the matrix, the number of HyperPartitions and the size of HyperPartitions of each individual matrix would give better performance of the RadixSort. Again this would require analysis of the performance of the RadixSort algorithm on different sizes and numbers of HyperPartition.

In Chapters 6 and 7 we presented results of running the sorting phase of the HyperPartition transpose in parallel using OpenMP. This assumes a shared memory machine where each processor core can address all the available memory. In some cases it is desirable to solve large linear algebra problems on distributed memory machines where matrices are partitioned onto different nodes. The Compressed Sparse Row format is not well suited to this partitioning, instead hierarchical blocked formats such as Hyper-Matrix [Herrero 03] are used. In some cases the hierarchical partitioning is just used between nodes and then a standard format such as CSR is used locally on each nodes. In these situations our Hybrid HyperPartition transpose could be incorporated into the local portion of the distributed memory parallel transpose.

8.3 Summary

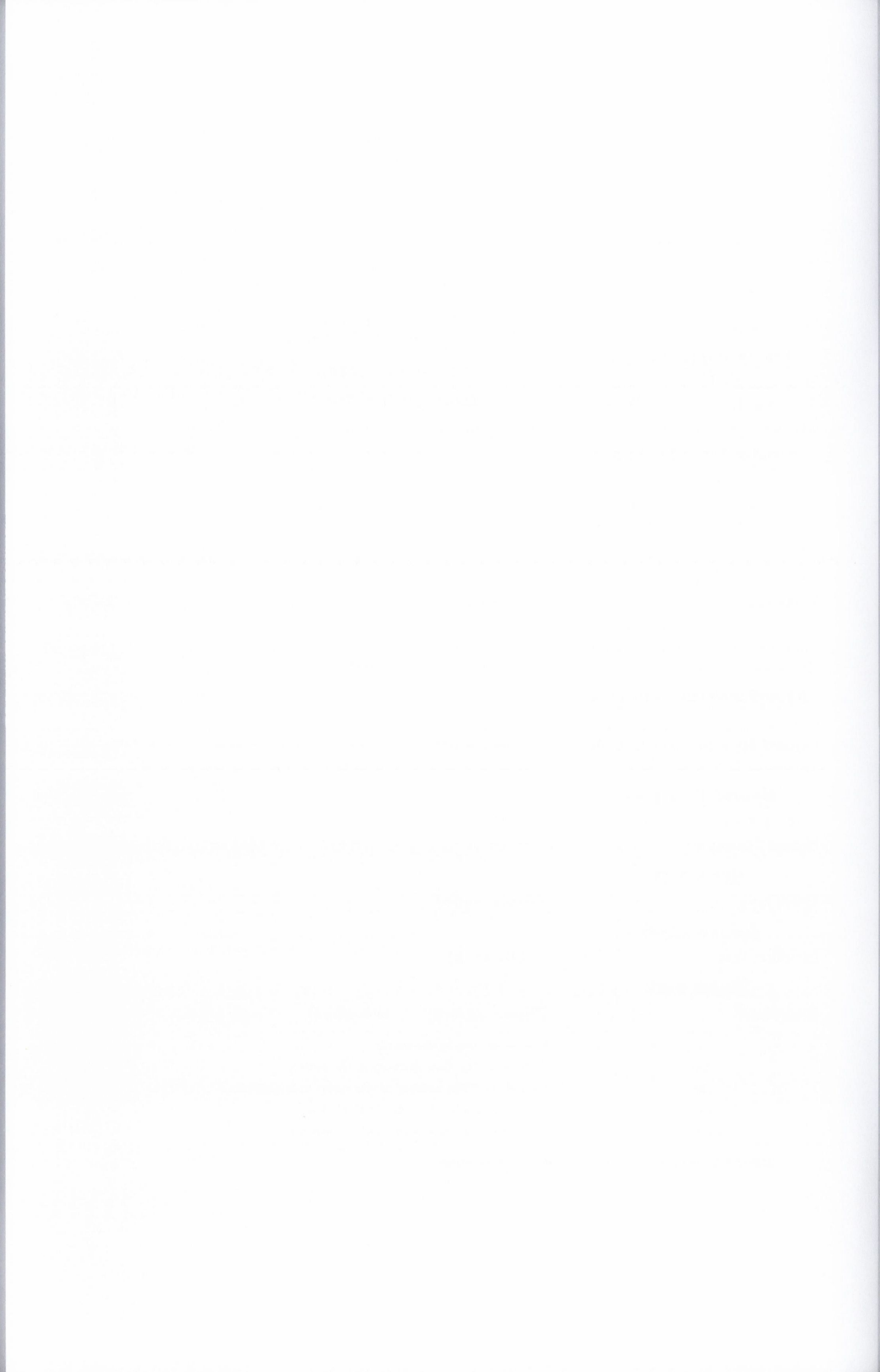
We have shown that matrices stored in Compressed Sparse Row storage format can be efficiently transposed in $\Theta(nnz + n)$ time using just $\Theta(n)$ additional space. We present three in-place transpose algorithms which adhere to these asymptotic complexities with runtime characteristics that make different trade-offs between memory usage and runtime. The Corresponding Row algorithm has a good relative execution time of 90% of Saad for all matrices with a memory cost of 21% of the space used by Saad. The HyperPartition with RadixSort algorithm has extremely low

relative memory usage under 1% of Saad and very good execution time for most input matrices also with an relative average of 90% of Saad in serial and 40% in parallel, however for some matrices it does not perform well. The Hybrid HyperPartition algorithm takes the best of both with a good relative execution time for almost all matrices at an average of 68% of the time at a cost of 9.3% of the memory. Running in parallel on 32 cores improves the execution time of the Hybrid Transpose to just 32% of Saad on average for the 259 large matrices.

Table 8.1: Algorithm Complexities

Algorithm	Runtime	Memory	Actual Memory
Out-Of-Place			
Out-Of-Place	$\Theta(nnz + n)$	$\Theta(nnz + n)$	$\sim(3nnz + n)$
Saad In-Place			
Saad In-Place	$\Theta(nnz + n)$	$\Theta(nnz)$	$\sim(nnz)$
Binary Range Lookup			
Single Binary Range Search	$\Theta(log(n))$	-	-
Total for Binary Range Searches	$\Theta(nnz\,.\,log(n))$	-	-
Transpose with Binary R.Srch	$\Theta(nnz\,.\,log(n) + n)$	$\Theta(n)$	$\sim(2n)$
Radix Table Lookup			
Build Radix Table	$\Theta(n)$	-	-
Single Radix Table Lookup	$\Omega(1) - \mathcal{O}(n)$	-	-
Total for Radix Table Lookups	$\mathcal{O}(nnz\,.\,n)$	-	-
Transpose with Radix Table	$\mathcal{O}(nnz\,.\,n + n)$	$\Theta(n)$	$\sim(2n + {}^{2}/_{n})$
Corresponding Row Lookup			
Build *coresp_table*	$\Theta(n)$	-	-
Single *corresp_table* lookup	$\Omega(1) - \mathcal{O}(n)$	-	-
Total for *corresp_table* lookups	$\mathcal{O}(nnz + n)$	-	-
Transpose with *corresp_table*	$\Theta(nnz + n)$	$\Theta(n)$	$\sim(3n)$
HyperPartition Transpose			
Convert from CSR to HypCSR	$\Theta(nnz + n)$	-	-
Convert HypCSR back to CSR	$\Theta(nnz + n)$	-	-
HyperPartition Transpose	$\Theta(nnz + n)$	$\Theta(n)$	$\sim(\frac{n}{2^{sb-1}})$ or $\sim({}^{n}/_{128})$
Hybrid Transpose			
Test for Structural Symmetry	$\Theta(n)$	-	-
Hybrid Transpose	$\Theta(nnz + n)$	$\Theta(n)$	$\sim(3n)$ or $\sim({}^{n}/_{128})$
Quick Sort			
Quick Sort	$\mathcal{O}(nnz\,.\,log(n))$	-	-
Insertion Sort			
Insertion Sort	$\mathcal{O}(nnz\,.\,n)$	-	-
Bucket Sort			
BucketSort	$\mathcal{O}(nnz\,.\,passes)$	$\Theta(buckets)$	$\sim(2\,.\,buckets)$

n	- Number of rows in the matrix
nnz	- Number of Non-Zero elements in the matrix
sb	- Steal Bits — The number of bits stolen in HyperPartition Transpose
$passes$	- Number of passes of the BucketSort algorithm
$buckets$	- Number of buckets in each pass of BucketSort



Appendix A

Matrix Tables

This appendix contains tables of information for 41 of the 259 input matrices from the University of Florida Sparse Matrix Collection (Section 3.6.1) which were used in the experiments outlined in Section 3.6.2.

Table A.1 shows structural information of the matrices. The dimensions of the matrix; number of rows, number of columns and number of non-zero elements in the matrix. The average number of elements per row, the % sparsity and a description of the structural shape of the matrix.

Table A.2 lists the source of these matrices. The type of linear algebra problem being examined which produced the matrix.

Table A.3 shows the algorithm memory usage (in MegaBytes) for Out-of-place, Saad, Binary range search, Radix Table ($\frac{n}{2}$), Corresponding Row and Serial HyperPartition with $k = 9$. The final column of the table shows the number of bits which have been *stolen*, *left behind* and *remain available* for these sample matrices when using the HyperPartition algorithm with the value of $k = 9$ for the remaining bits heuristic.

Table A.4 shows the algorithm execution time (in seconds) for a number of the transpose algorithms as follows:

OOP	- Out-of-place transpose
Saad	- Saad in-place transpose
Binary	- Binary Range Search transpose
Radix $\frac{n}{2}$	- Radix Table ($\frac{n}{2}$) transpose
Corr	- Corresponding Row transpose
Hyp	- Serial HyperPartition transpose with $k = 9$
Hybrid	- Serial Hybrid transpose with $k = 9$
Hybrid-32	- 32-Way Parallel Hybrid transpose with $k = 6$

Table A.1: Matrix Information

Matrix	Rows	Cols	NNZ	Per Row	% Sparcity	Shape
torso2	115,967	115,967	1,033,473	9	0.008	Square
nemsemm1	3,945	75,352	1,053,986	267	0.355	Rectangle
dbir2	18,906	45,877	1,158,159	61	0.134	Rectangle
darcy003	389,874	389,874	1,167,685	3	0.001	Lower Triangle
xenon1	48,600	48,600	1,181,120	24	0.050	Square Struct-Sym
ct20stif	52,329	52,329	1,375,396	26	0.050	Lower Triangle
heart1	3,557	3,557	1,387,773	390	10.969	Square Struct-Sym
venkat01	62,424	62,424	1,717,792	28	0.044	Square Struct-Sym
appu	14,000	14,000	1,853,104	132	0.945	Square Struct-Sym
pdb1HYS	36,417	36,417	2,190,591	60	0.165	Lower Triangle
s3dkq4m2	90,449	90,449	2,455,670	27	0.030	Lower Triangle
Si34H36	97,569	97,569	2,626,974	27	0.028	Lower Triangle
shipsec1	140,874	140,874	3,977,139	28	0.020	Lower Triangle
G3_circuit	1,585,478	1,585,478	4,623,152	3	0.000	Lower Triangle
Hamrle3	1,447,360	1,447,360	5,514,242	4	0.000	Square
bmw3_2	227,362	227,362	5,757,996	25	0.011	Lower Triangle
af_shell8	504,855	504,855	9,046,865	18	0.004	Lower Triangle
msdoor	415,863	415,863	10,328,399	25	0.006	Lower Triangle
ohne2	181,343	181,343	11,063,545	61	0.034	Square
bone010_M	986,703	986,703	12,437,739	13	0.001	Lower Triangle
F1	343,791	343,791	13,590,452	40	0.011	Lower Triangle
nd24k	72,000	72,000	14,393,817	200	0.278	Lower Triangle
Fault_639	638,802	638,802	14,626,683	23	0.004	Lower Triangle
nlpkkt80	1,062,400	1,062,400	14,883,536	14	0.001	Lower Triangle
inline_1	503,712	503,712	18,660,027	37	0.007	Lower Triangle
rajat31	4,690,002	4,690,002	20,316,253	4	0.000	Square Struct-Sym
ldoor	952,203	952,203	23,737,339	25	0.003	Lower Triangle
af_shell10	1,508,065	1,508,065	27,090,195	18	0.001	Lower Triangle
cage14	1,505,785	1,505,785	27,130,349	18	0.001	Square Struct-Sym
Serena	1,391,349	1,391,349	32,961,525	24	0.002	Lower Triangle
bundle1__L	10,581	10,581	35,781,540	3,382	31.960	Square
bone010	986,703	986,703	36,326,514	37	0.004	Lower Triangle
RM07R	381,689	381,689	37,464,962	98	0.026	Square
audikw_1	943,695	943,695	39,297,771	42	0.004	Lower Triangle
nlpkkt120	3,542,400	3,542,400	50,194,096	14	0.000	Lower Triangle
cage15	5,154,859	5,154,859	99,199,551	19	0.000	Square Struct-Sym
ML_Geer	1,504,002	1,504,002	110,879,972	74	0.005	Square Struct-Sym
nlpkkt160	8,345,600	8,345,600	118,931,856	14	0.000	Lower Triangle
nlpkkt200	16,240,000	16,240,000	232,232,816	14	0.000	Lower Triangle
HV15R	2,017,169	2,017,169	283,073,458	140	0.007	Square
nlpkkt240	27,993,600	27,993,600	401,232,976	14	0.000	Lower Triangle

Table A.2: Matrix Source

Matrix	Source
torso2	FEA: 2D model of torso
nemsemm1	Linear programming problem
dbir2	Linear programming problem
darcy003	FEA: 2D/3D problem
xenon1	Materials problem - Complex Crystals
ct20stif	Structural problem - Engine Block Stiffness
heart1	FEA: Heart
venkat01	Computational fluid dynamics
appu	APP Benchmark - directed weighted random graph
pdb1HYS	Protein data bank 1HYS
s3dkq4m2	FEA: Structural problem - cylindrical shell
Si34H36	Quantum chemistry problem
shipsec1	FEA: Structural problem - Ship section/detail
G3_circuit	Circuit simulation problem
Hamrle3	Circuit simulation problem
bmw3_2	Structural problem
af_shell8	Structural problem - Sheet metal forming
msdoor	Structural problem - Medium size door
ohne2	Semiconductor device problem
bone010_M	Model reduction - 3D trabecular bone
F1	Structural problem - AUDI engine crankshaft
nd24k	ND problem set - 2D/3D problem
Fault_639	Structural problem - Faulted gas reservoir
nlpkkt80	KKT Bitmedical Optimization Problem - Nonconvex 3D PDE
inline_1	Structural problem - Stiffness
rajat31	Circuit simulation problem
ldoor	Structural problem
af_shell10	Structural problem - Sheet metal forming
cage14	DNA electrophoresis - 14 monomers in polymer
Serena	Structural - Gas resevoir simulation for CO2 sequestration
bundle1__L	Unknown
bone010	Model reduction - 3D trabecular bone
RM07R	CFD: 3D viscous case
audikw_1	Structural problem
nlpkkt120	KKT Bitmedical Optimization Problem - Nonconvex 3D PDE
cage15	DNA electrophoresis - 15 monomers in polymer
ML_Geer	Poroelastic Structural Problem
nlpkkt160	KKT Bitmedical Optimization Problem - Nonconvex 3D PDE
nlpkkt200	KKT Bitmedical Optimization Problem - Nonconvex 3D PDE
HV15R	CFD: 3D engine fan
nlpkkt240	KKT Bitmedical Optimization Problem - Nonconvex 3D PDE

Table A.3: Algorithm Memory Usage (MegaBytes)

Matrix	OOP	Saad	Binary	Radix $\frac{n}{2}$	Corr	Hyp	$(\mathbf{S}^{tln}/\mathbf{L}^{ft}/\mathbf{A}^{vl})$
torso2	12.270	3.942	0.885	1.010	1.327	0.000923	15 / 2 / 0
nemsemm1	12.077	4.021	0.030	0.155	0.045	0.005257	12 / - / 3
dbir2	13.326	4.418	0.144	0.207	0.216	0.000820	15 / - / 1
darcy003	14.850	4.454	2.975	3.475	4.462	0.000786	13 / 6 / 0
xenon1	13.702	4.506	0.371	0.433	0.556	0.000786	16 / - / 0
ct20stif	15.940	5.247	0.399	0.462	0.599	0.000847	16 / - / 0
heart1	15.895	5.294	0.027	0.031	0.041	0.000908	12 / - / 8
venkat01	19.897	6.553	0.476	0.539	0.714	0.000984	16 / - / 0
appu	21.261	7.069	0.107	0.122	0.160	0.000893	14 / - / 4
pdb1HYS	25.208	8.356	0.278	0.340	0.417	0.000603	16 / - / 0
s3dkq4m2	28.448	9.368	0.690	0.815	1.035	0.000740	15 / 2 / 0
Si34H36	30.436	10.021	0.744	0.869	1.117	0.000786	15 / 2 / 0
shipsec1	46.052	15.172	1.075	1.325	1.612	0.000587	14 / 4 / 0
G3_circuit	58.956	17.636	12.096	14.096	18.144	0.011879	11 / 10 / 0
Hamrle3	68.627	21.035	11.042	13.042	16.564	0.010841	11 / 10 / 0
bmw3_2	66.762	21.965	1.735	1.985	2.602	0.000908	14 / 4 / 0
af_shell8	105.459	34.511	3.852	4.352	5.778	0.000999	13 / 6 / 0
msdoor	119.786	39.400	3.173	3.673	4.759	0.000832	13 / 6 / 0
ohne2	127.304	42.204	1.384	1.634	2.075	0.000740	14 / 4 / 0
bone010_M	146.103	47.446	7.528	8.528	11.292	0.003731	12 / 8 / 0
F1	156.842	51.843	2.623	3.123	3.934	0.000694	13 / 6 / 0
nd24k	164.999	54.908	0.549	0.674	0.824	0.000603	15 / 2 / 0
Fault_639	169.826	55.796	4.874	5.874	7.311	0.002434	12 / 8 / 0
nlpkkt80	174.381	56.776	8.105	10.105	12.158	0.007973	11 / 10 / 0
inline_1	215.469	71.182	3.843	4.343	5.765	0.000999	13 / 6 / 0
rajat31	250.392	77.500	35.782	43.782	53.673	0.139839	9 / 14 / 0
ldoor	275.285	90.551	7.265	8.265	10.897	0.003609	12 / 8 / 0
af_shell10	315.775	103.341	11.506	13.506	17.258	0.011299	11 / 10 / 0
cage14	316.226	103.494	11.488	13.488	17.232	0.011284	11 / 10 / 0
Serena	382.522	125.738	10.615	12.615	15.923	0.010429	11 / 10 / 0
bundle1__L	409.528	136.496	0.081	0.096	0.121	0.000694	14 / - / 4
bone010	419.488	138.575	7.528	8.528	11.292	0.003731	12 / 8 / 0
RM07R	430.208	142.917	2.912	3.412	4.368	0.000771	13 / 6 / 0
audikw_1	453.327	149.909	7.200	8.200	10.800	0.003578	12 / 8 / 0
nlpkkt120	587.939	191.475	27.026	31.026	40.540	0.052849	10 / 12 / 0
cage15	1154.913	378.416	39.328	47.328	58.993	0.153694	9 / 14 / 0
ML_Geer	1274.658	422.974	11.475	13.475	17.212	0.011269	11 / 10 / 0
nlpkkt160	1392.903	453.689	63.672	71.672	95.508	0.248772	9 / 14 / 0
nlpkkt200	2719.644	885.898	123.901	139.901	185.852	0.968040	8 / 16 / 0
HV15R	3247.214	1079.840	15.390	17.390	23.085	0.015083	11 / 10 / 0
nlpkkt240	4698.534	1530.582	213.574	245.574	320.361	3.337151	7 / 18 / 0

Table A.4: Algorithm Execution Time (seconds)

Matrix	OOP	Saad	Binary	Radix $\frac{n}{2}$	Corr	Hyp	Hybrid	Hybrid-32
torso2	0.014	0.026	0.105	0.029	0.023	0.038	0.042	0.014
nemsemm1	0.032	0.107	0.207	0.115	0.100	0.086	0.076	0.027
dbir2	0.039	0.120	0.204	0.122	0.112	0.097	0.097	0.047
darcy003	0.052	0.125	0.332	0.122	0.120	0.065	0.078	0.032
xenon1	0.033	0.030	0.102	0.033	0.022	0.058	0.022	0.018
ct20stif	0.028	0.144	0.250	0.143	0.133	0.091	0.089	0.033
heart1	0.065	0.050	0.111	0.046	0.040	0.130	0.039	0.026
venkat01	0.055	0.048	0.157	0.051	0.038	0.094	0.038	0.029
appu	0.113	0.120	0.186	0.085	0.072	0.216	0.075	0.053
pdb1HYS	0.060	0.260	0.412	0.256	0.244	0.211	0.211	0.073
s3dkq4m2	0.038	0.143	0.336	0.153	0.129	0.141	0.179	0.051
Si34H36	0.104	0.342	0.593	0.324	0.303	0.237	0.238	0.083
shipsec1	0.074	0.572	0.912	0.561	0.491	0.293	0.403	0.091
G3_circuit	0.102	1.063	2.423	1.101	1.099	0.344	0.344	0.214
Hamrle3	0.099	1.458	3.172	1.565	1.527	0.427	0.427	0.247
bmw3_2	0.123	1.157	1.940	1.226	1.041	0.478	0.689	0.220
af_shell8	0.130	0.756	1.527	0.743	0.689	0.562	0.877	0.178
msdoor	0.305	2.570	4.169	2.374	2.335	0.851	1.023	0.334
ohne2	0.490	0.446	1.316	0.442	0.364	0.927	0.367	0.339
bone010_M	0.189	2.879	5.079	3.205	2.640	0.961	1.020	0.323
F1	0.760	3.411	5.483	3.656	3.079	1.251	1.304	0.506
nd24k	0.657	2.465	3.465	2.278	2.225	1.797	1.635	0.486
Fault_639	0.256	3.776	5.638	3.480	3.488	1.247	1.506	0.458
nlpkkt80	0.367	3.623	6.823	3.584	3.331	1.449	1.449	0.682
inline_1	0.799	5.947	7.843	4.680	5.674	1.661	1.804	0.728
rajat31	0.343	0.616	2.592	0.632	0.572	0.790	0.570	0.455
ldoor	0.741	8.535	11.965	6.913	8.469	2.252	2.721	0.989
af_shell10	0.372	2.321	4.484	2.043	2.193	1.752	1.752	0.506
cage14	1.892	1.392	3.739	1.088	1.126	2.029	1.154	1.289
Serena	0.905	10.673	18.215	10.597	10.391	3.034	3.034	1.215
bundle1__L	2.087	6.635	7.387	6.043	6.183	2.975	2.758	1.262
bone010	0.602	9.701	13.552	7.555	9.128	3.568	3.721	0.911
RM07R	1.246	9.380	13.854	8.441	8.488	4.881	3.744	1.001
audikw_1	1.497	14.650	20.802	11.987	14.529	4.519	4.416	1.638
nlpkkt120	1.931	18.307	31.829	15.837	18.431	5.102	5.102	3.033
cage15	8.012	5.453	15.235	4.194	4.474	9.176	4.595	4.162
ML_Geer	2.182	3.727	12.894	3.408	2.961	11.504	2.982	2.381
nlpkkt160	5.858	53.069	111.354	48.345	55.650	14.475	14.475	10.283
nlpkkt200	11.796	116.659	276.918	110.659	124.855	31.985	31.985	24.500
HV15R	12.718	115.562	198.619	115.123	111.141	31.879	31.879	10.396
nlpkkt240	20.422	223.478	610.015	267.240	237.771	80.765	80.765	79.097

Appendix B

Detailed HyperPartition Performance Graphs

This appendix shows additional graphs with more detailed breakdowns of the performance of the HyperPartition algorithms with different values of the Remaining Bits parameter between $k = 1$ and $k = 10$.

Figure B.1 shows Serial Hyperpartition with QuickSort with values of the remaining bits heuristic of $k = 1$ to $k = 10$.

Figure B.2 shows Parallel HyperPartition with QuickSort with values of the remaining bits heuristic of $k = 1$ to $k = 10$.

Figure B.3 shows Serial HyperPartition with QuickSort of just the UnSymmetric matrices with values of the remaining bits heuristic of $k = 1$ to $k = 10$.

Figure B.4 shows Serial HyperPartition with QuickSort of just the Symmetric matrices with values of the remaining bits heuristic of $k = 1$ to $k = 10$.

Figure B.5 shows Serial HyperPartition with RadixSort with values of the remaining bits heuristic of $k = 1$ to $k = 10$.

Figure B.6 shows Parallel HyperPartition with RadixSort with values of the remaining bits heuristic of $k = 1$ to $k = 10$.

Figure B.7 shows Serial Hybrid with RadixSort with values of the remaining bits heuristic of $k = 1$ to $k = 10$.

Figure B.8 shows Parallel Hybrid with RadixSort with values of the remaining bits heuristic of $k = 1$ to $k = 10$.

Figure B.1 (a,b): Serial Hyperpartition With QuickSort: $k = 1 \rightarrow 10$

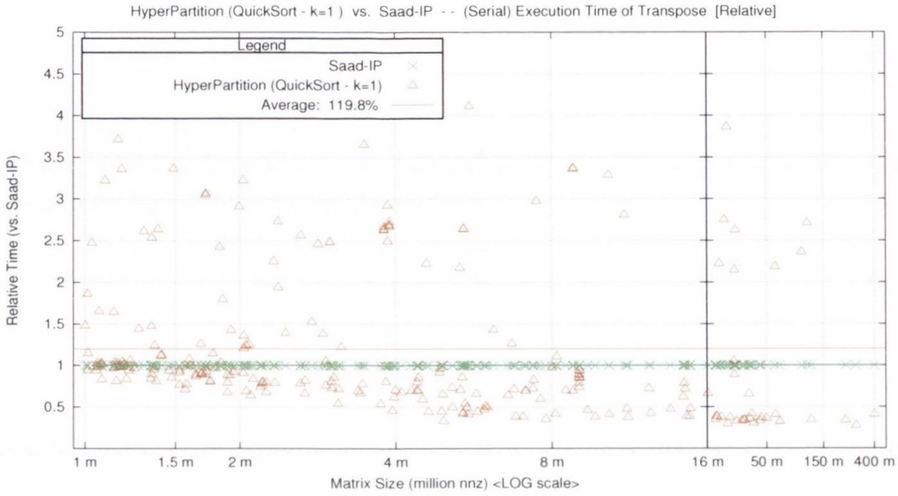


Figure B.1 (a): Serial Hyperpartition With QuickSort: $k = 1$

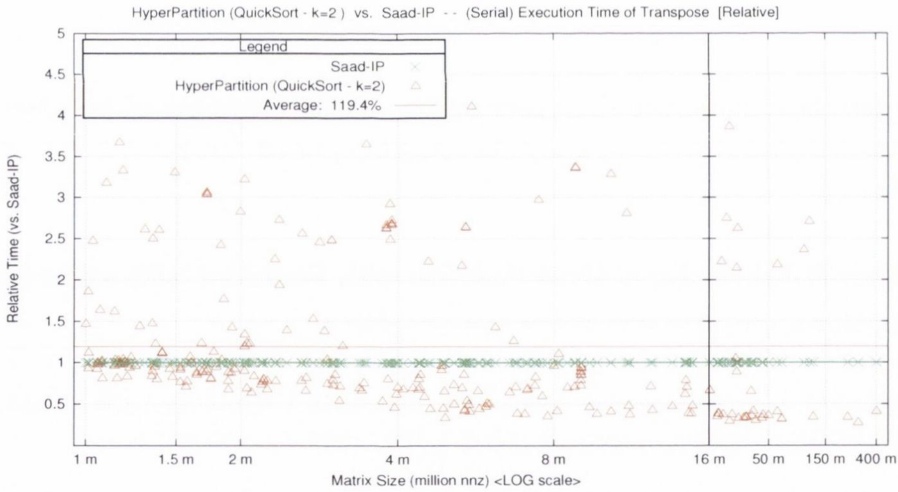


Figure B.1 (b): Serial Hyperpartition With QuickSort: $k = 2$

Figure B.1 (c,d): Serial Hyperpartition With QuickSort: $k = 1 \to 10$

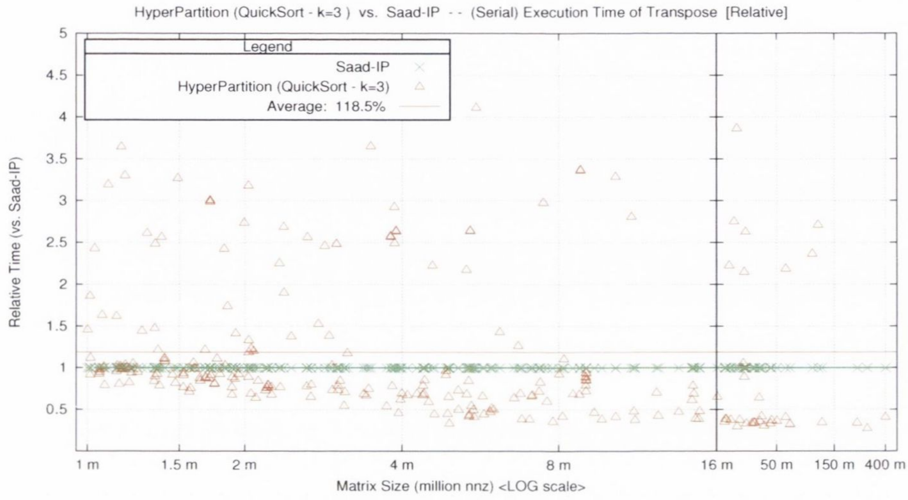


Figure B.1 (c): Serial Hyperpartition With QuickSort: $k = 3$

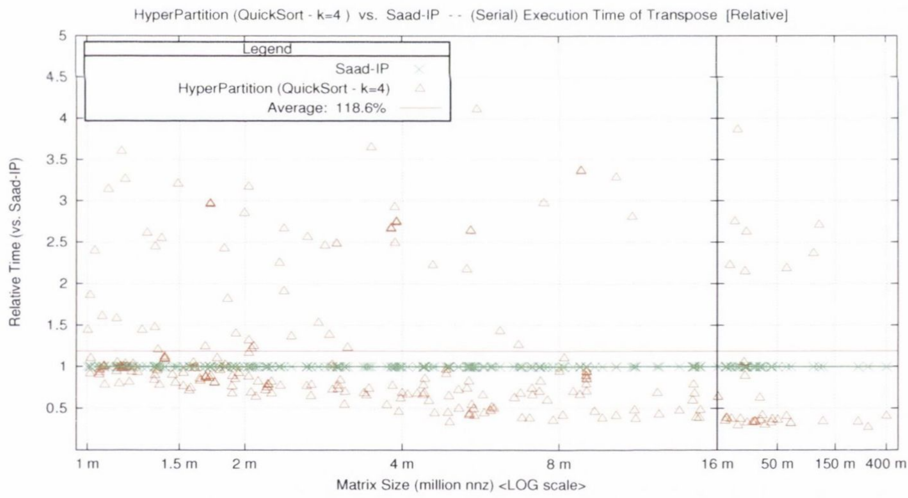


Figure B.1 (d): Serial Hyperpartition With QuickSort: $k = 4$

Figure B.1 (e,f): Serial Hyperpartition With QuickSort: $k = 1 \rightarrow 10$

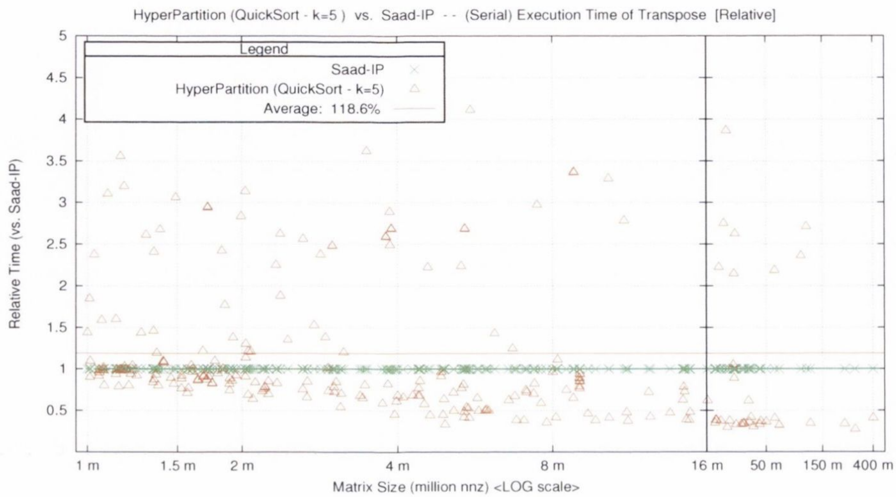


Figure B.1 (e): Serial Hyperpartition With QuickSort: $k = 5$

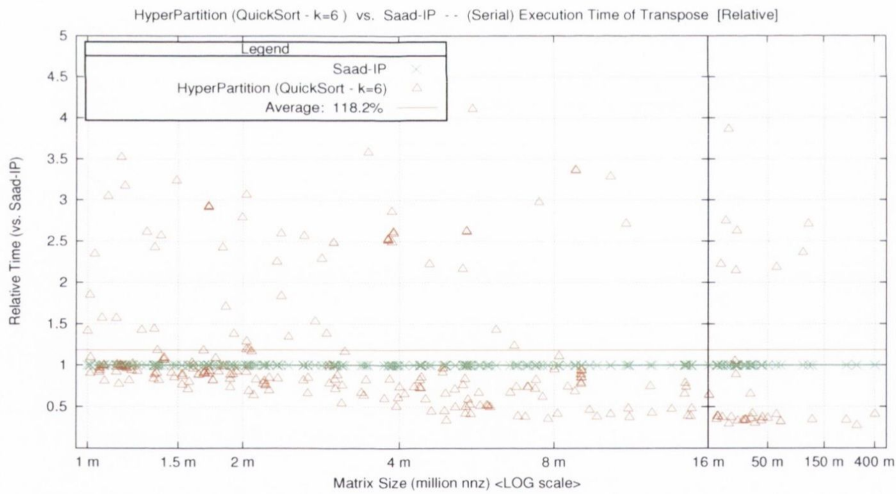


Figure B.1 (f): Serial Hyperpartition With QuickSort: $k = 6$

Figure B.1 (g,h): Serial Hyperpartition With QuickSort: $k = 1 \rightarrow 10$

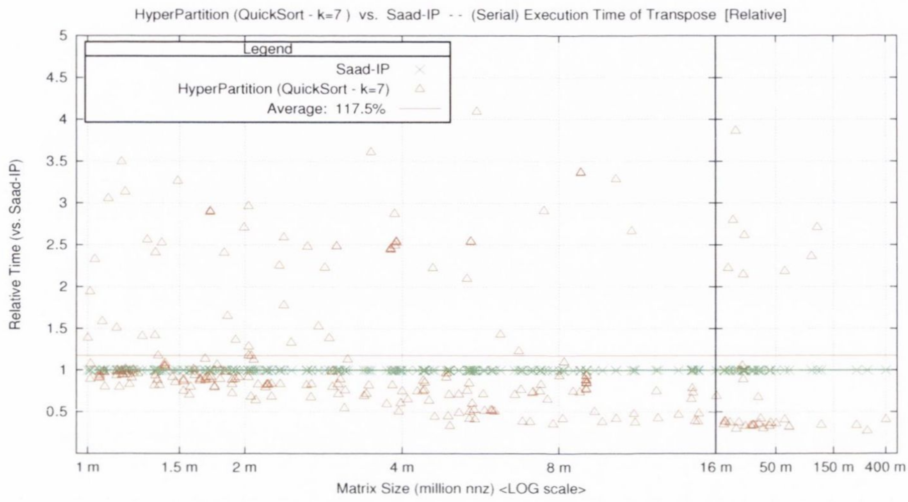


Figure B.1 (g): Serial Hyperpartition With QuickSort: $k = 7$

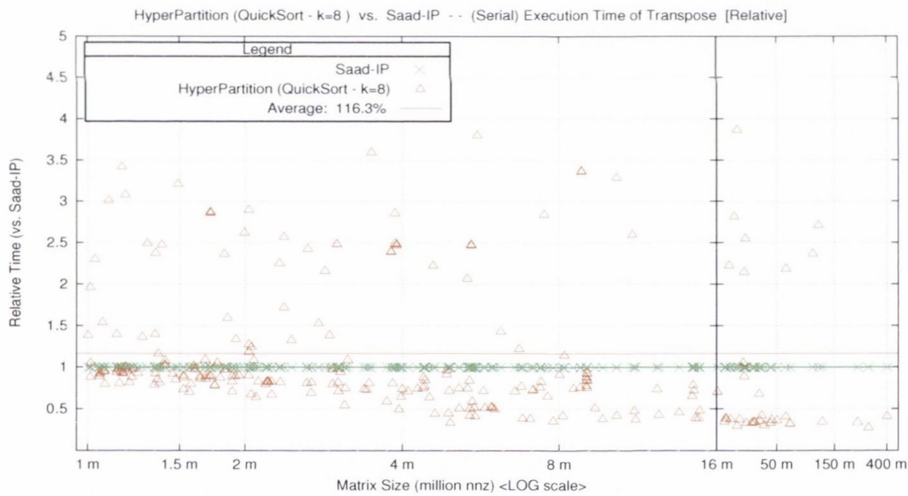


Figure B.1 (h): Serial Hyperpartition With QuickSort: $k = 8$

Figure B.1 (i,j): Serial Hyperpartition With QuickSort: $k = 1 \rightarrow 10$

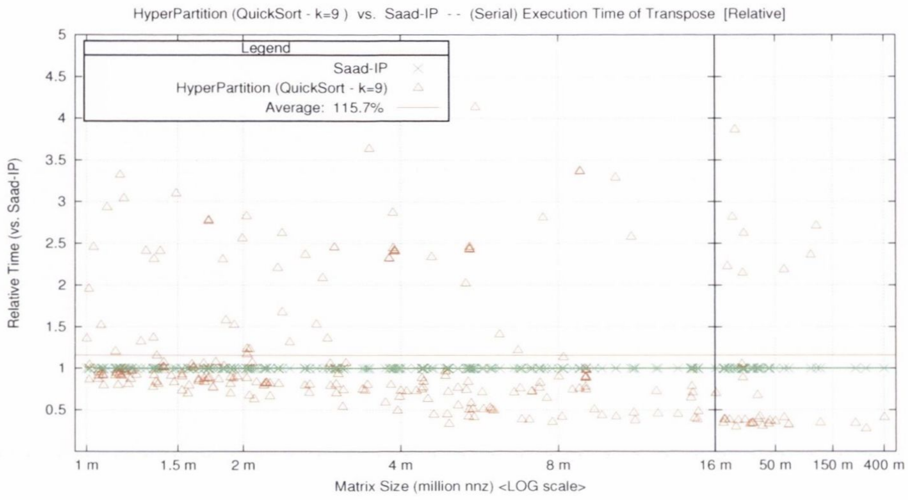


Figure B.1 (i): Serial Hyperpartition With QuickSort: $k = 9$

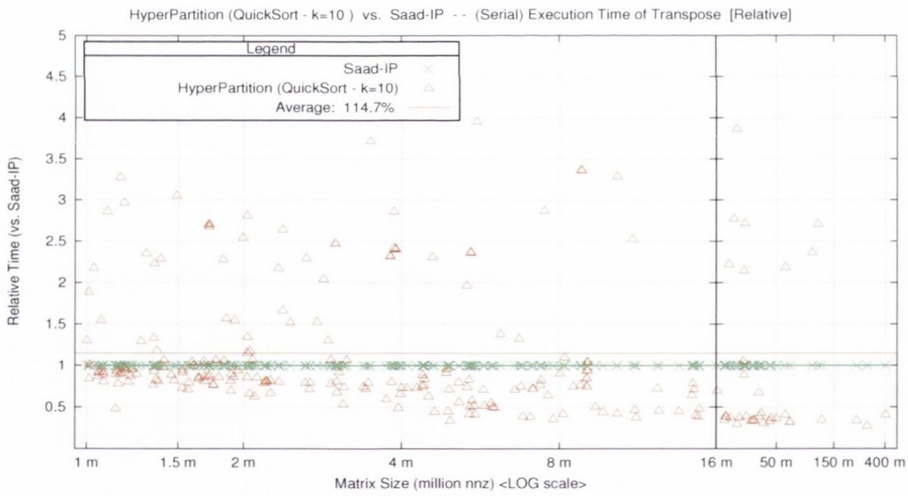


Figure B.1 (j): Serial Hyperpartition With QuickSort: $k = 10$

Figure B.2 (a,b): Parallel HyperPartition with QuickSort: $k = 1 \rightarrow 10$

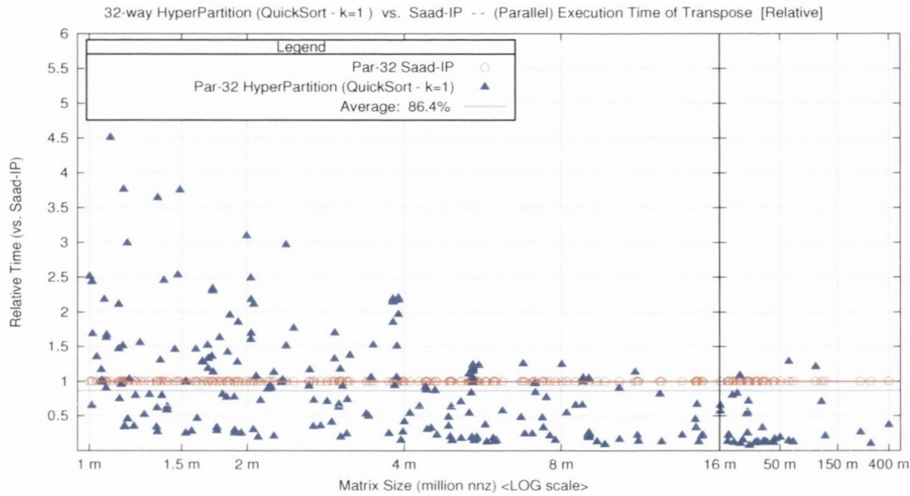


Figure B.2 (a): Parallel HyperPartition with QuickSort: $k = 1$

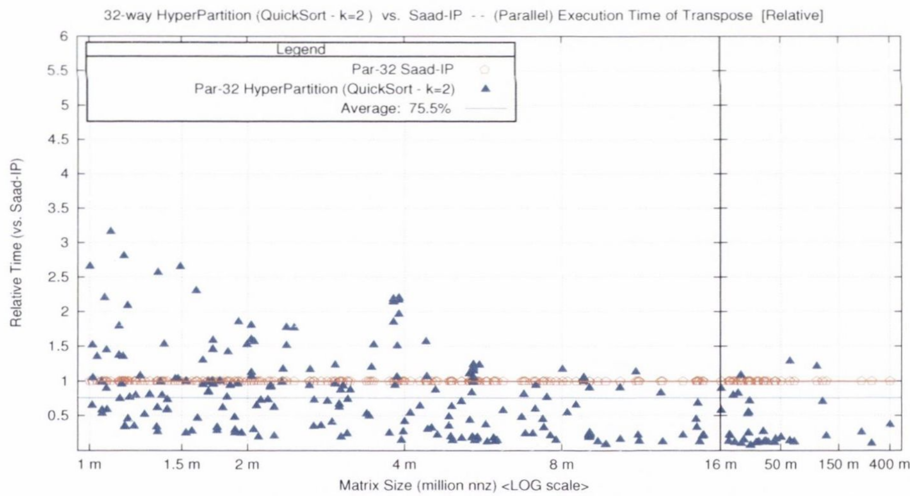


Figure B.2 (b): Parallel HyperPartition with QuickSort: $k = 2$

Figure B.2 (c,d): Parallel HyperPartition with QuickSort: $k = 1 \rightarrow 10$

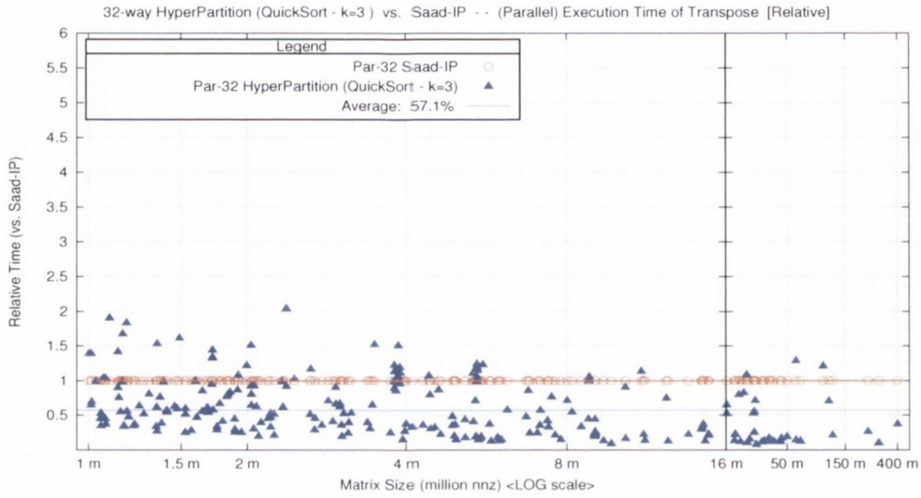


Figure B.2 (c): Parallel HyperPartition with QuickSort: $k = 3$

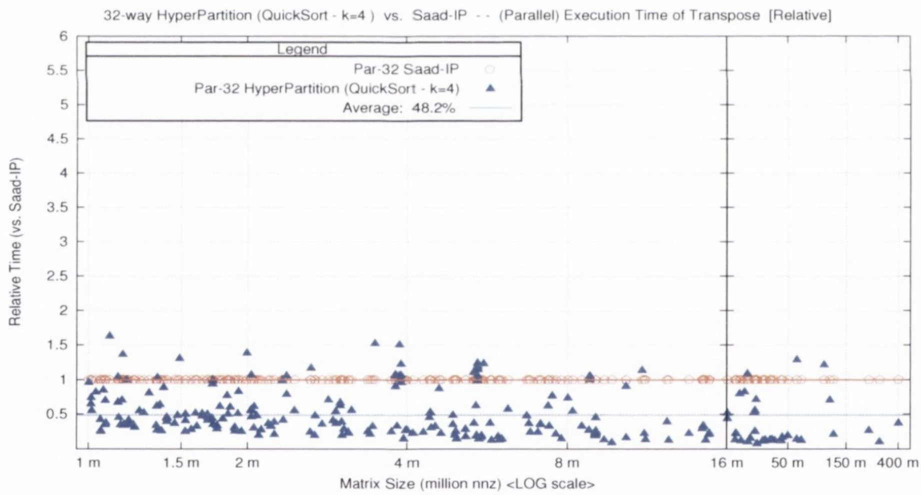


Figure B.2 (d): Parallel HyperPartition with QuickSort: $k = 4$

Figure B.2 (e,f): Parallel HyperPartition with QuickSort: $k = 1 \rightarrow 10$

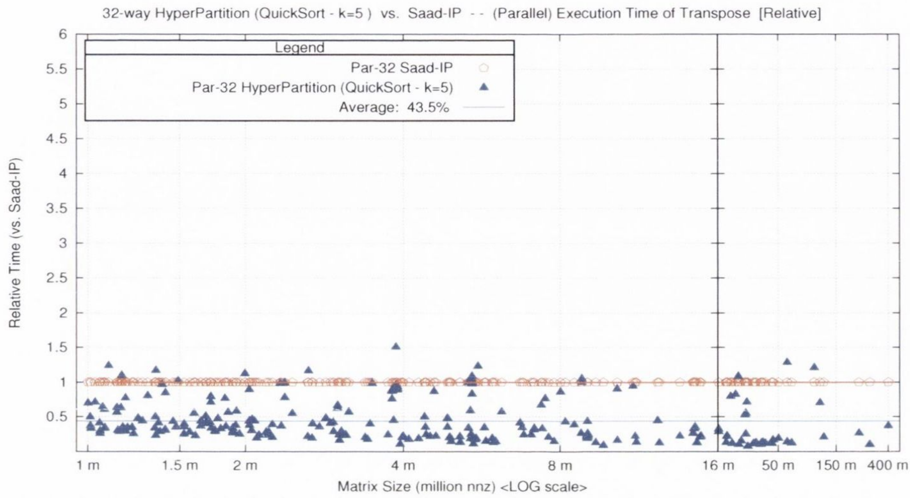


Figure B.2 (e): Parallel HyperPartition with QuickSort: $k = 5$

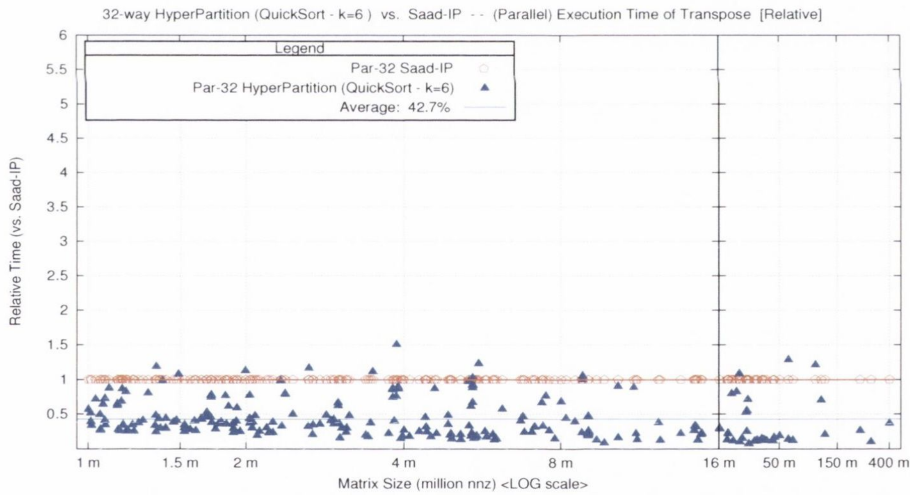


Figure B.2 (f): Parallel HyperPartition with QuickSort: $k = 6$

Figure B.2 (g,h): Parallel HyperPartition with QuickSort: $k = 1 \rightarrow 10$

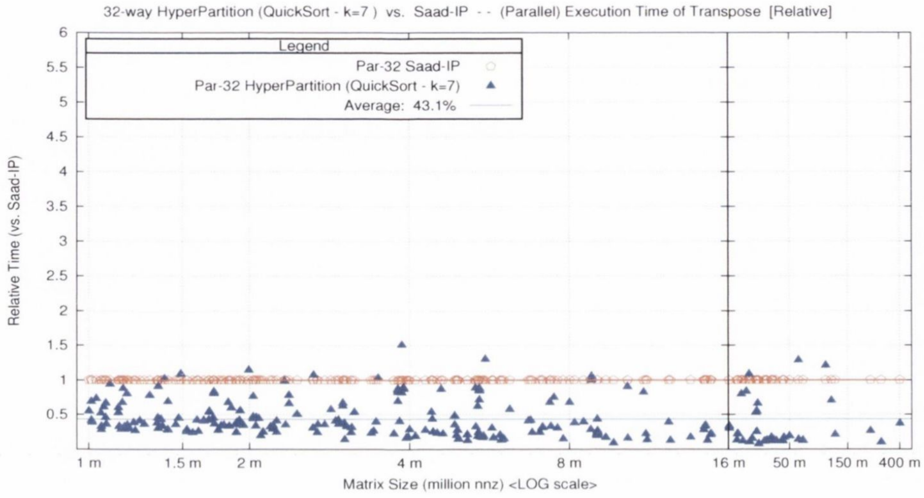


Figure B.2 (g): Parallel HyperPartition with QuickSort: $k = 7$

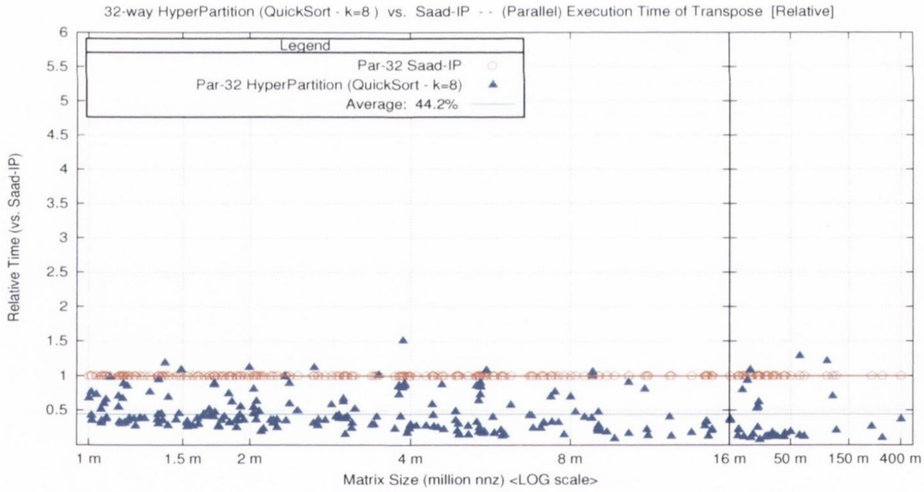


Figure B.2 (h): Parallel HyperPartition with QuickSort: $k = 8$

Figure B.2 (i,j): Parallel HyperPartition with QuickSort: $k = 1 \rightarrow 10$

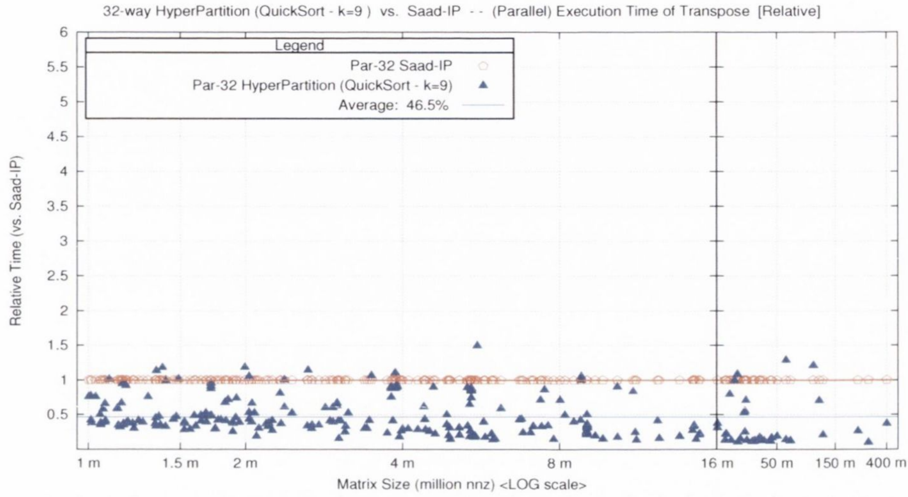


Figure B.2 (i): Parallel HyperPartition with QuickSort: $k = 9$

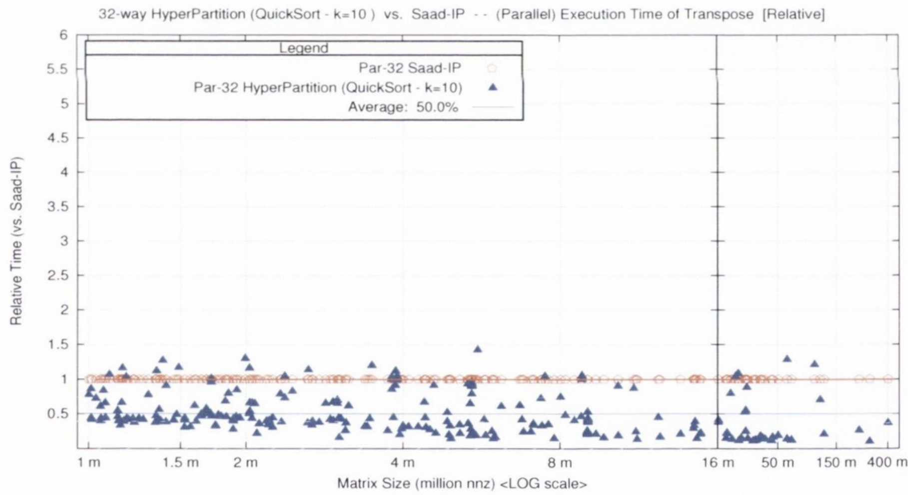


Figure B.2 (j): Parallel HyperPartition with QuickSort: $k = 10$

Figure B.3 (a,b): Serial UnSymmetric HyperPartition QuickSort: $k = 1 \rightarrow 10$

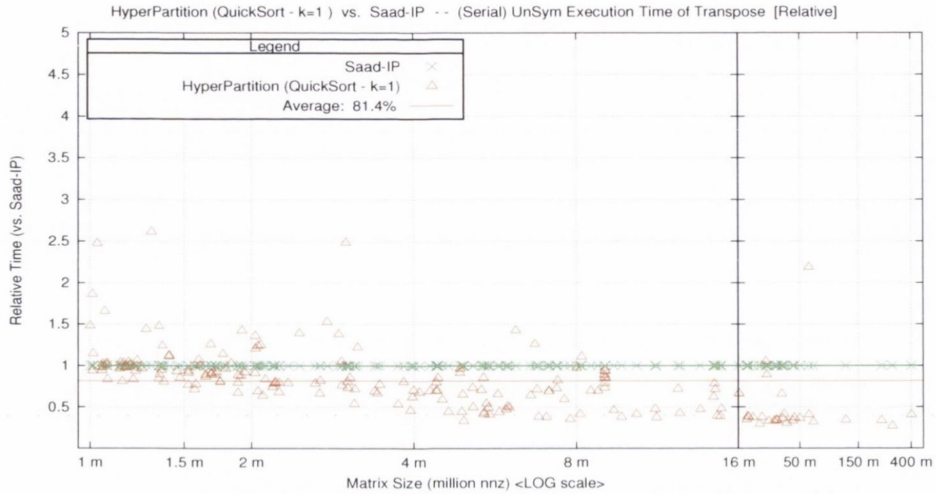


Figure B.3 (a): Serial UnSymmetric HyperPartition QuickSort: $k = 1$

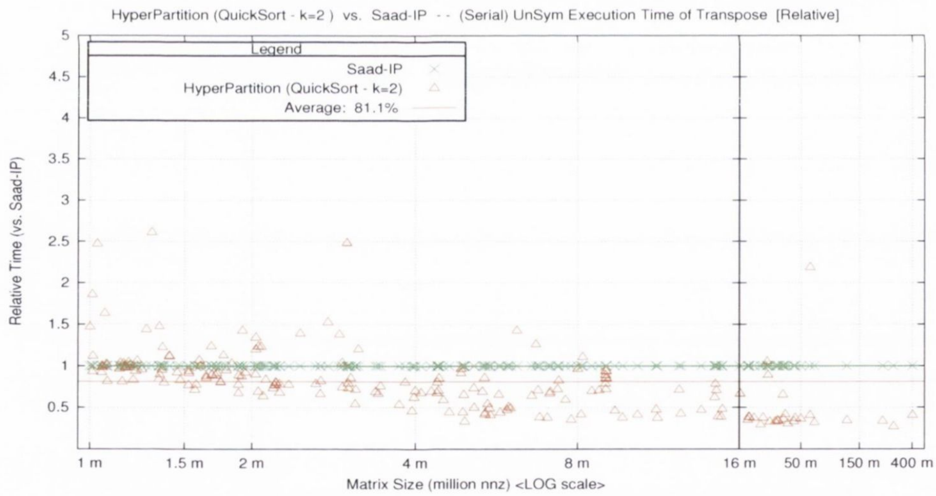


Figure B.3 (b): Serial UnSymmetric HyperPartition QuickSort: $k = 2$

Figure B.3 (c,d): Serial UnSymmetric HyperPartition QuickSort: $k = 1 \rightarrow 10$

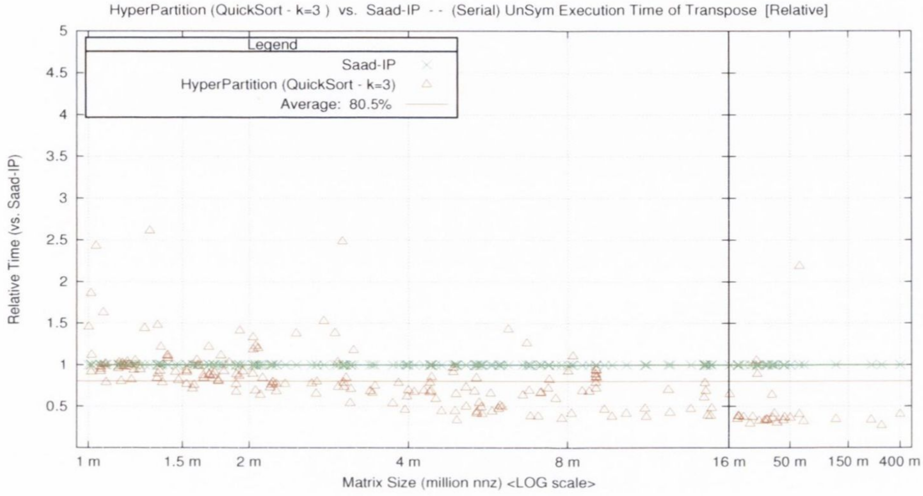


Figure B.3 (c): Serial UnSymmetric HyperPartition QuickSort: $k = 3$

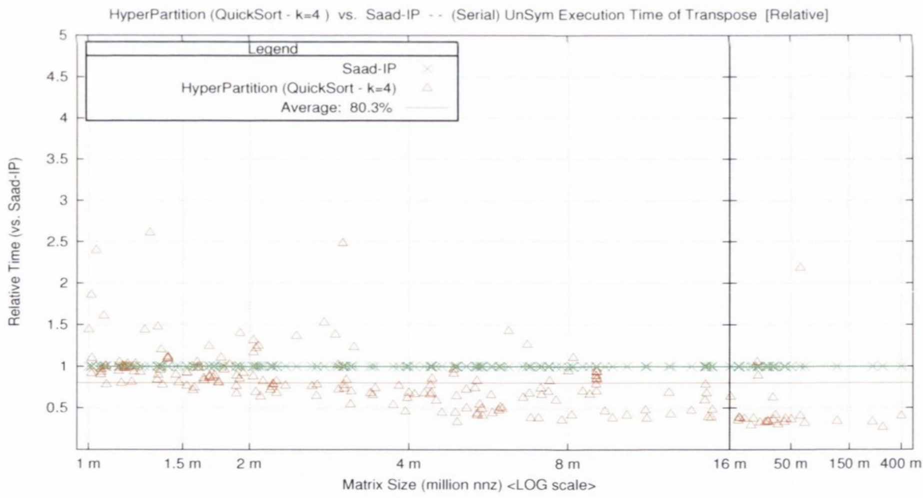


Figure B.3 (d): Serial UnSymmetric HyperPartition QuickSort: $k = 4$

Figure B.3 (e,f): Serial UnSymmetric HyperPartition QuickSort: $k = 1 \rightarrow 10$

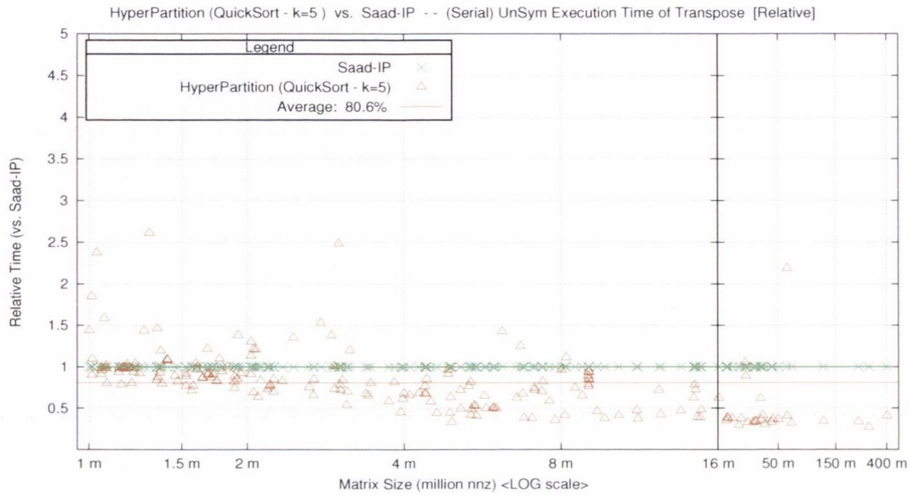


Figure B.3 (e): Serial UnSymmetric HyperPartition QuickSort: $k = 5$

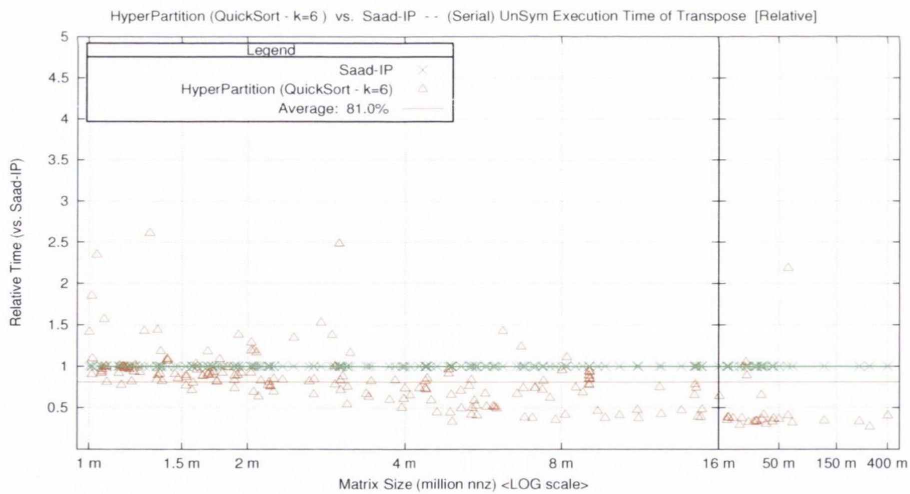


Figure B.3 (f): Serial UnSymmetric HyperPartition QuickSort: $k = 6$

Figure B.3 (g,h): Serial UnSymmetric HyperPartition QuickSort: $k = 1 \rightarrow 10$

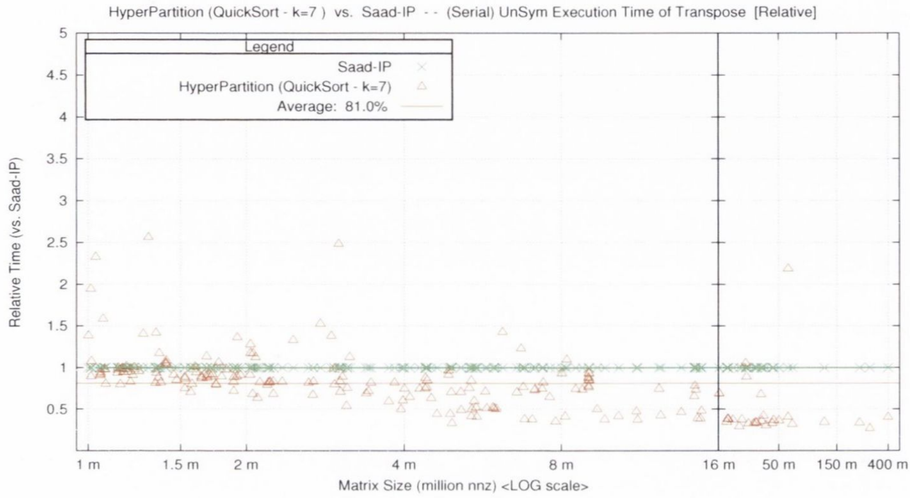


Figure B.3 (g): Serial UnSymmetric HyperPartition QuickSort: $k = 7$

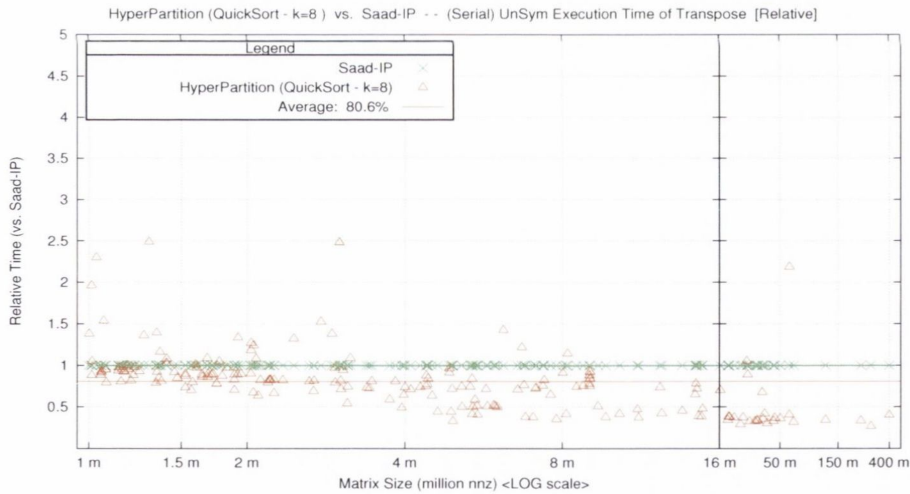


Figure B.3 (h): Serial UnSymmetric HyperPartition QuickSort: $k = 8$

Figure B.3 (i,j): Serial UnSymmetric HyperPartition QuickSort: $k = 1 \rightarrow 10$

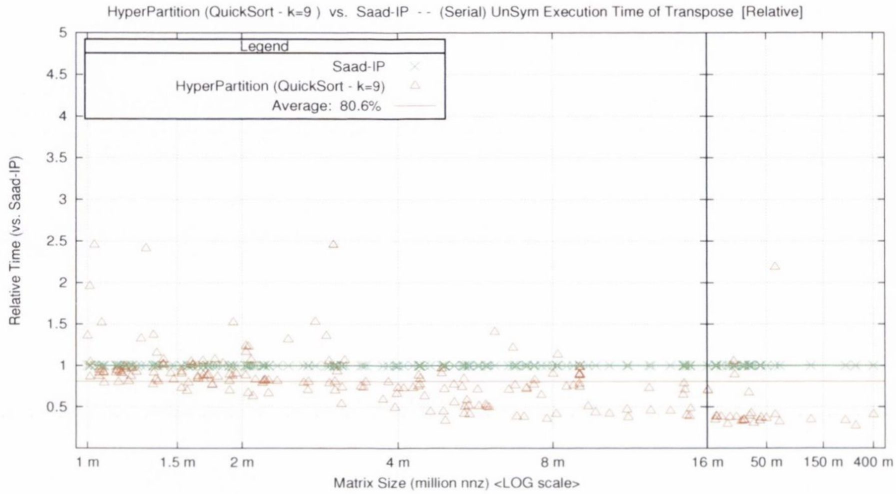


Figure B.3 (i): Serial UnSymmetric HyperPartition QuickSort: $k = 9$

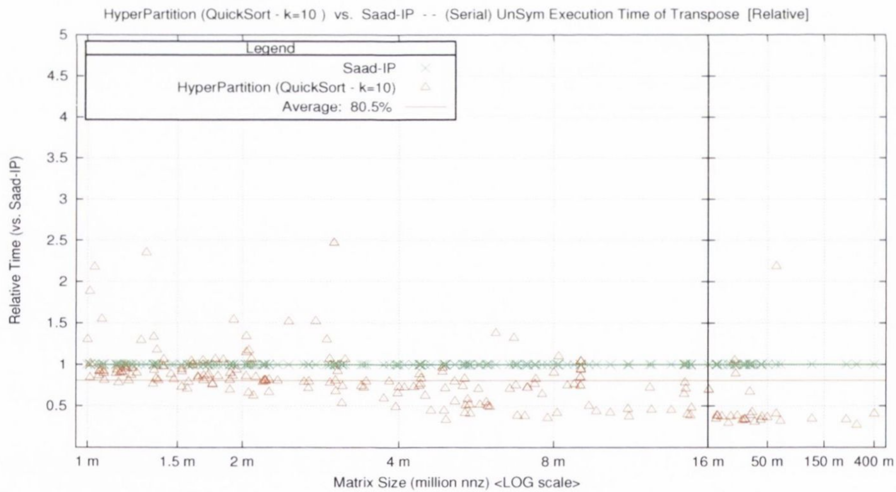


Figure B.3 (j): Serial UnSymmetric HyperPartition QuickSort: $k = 10$

Figure B.4 (a,b): Serial Symmetric HyperPartition with QuickSort: $k = 1 \rightarrow 10$

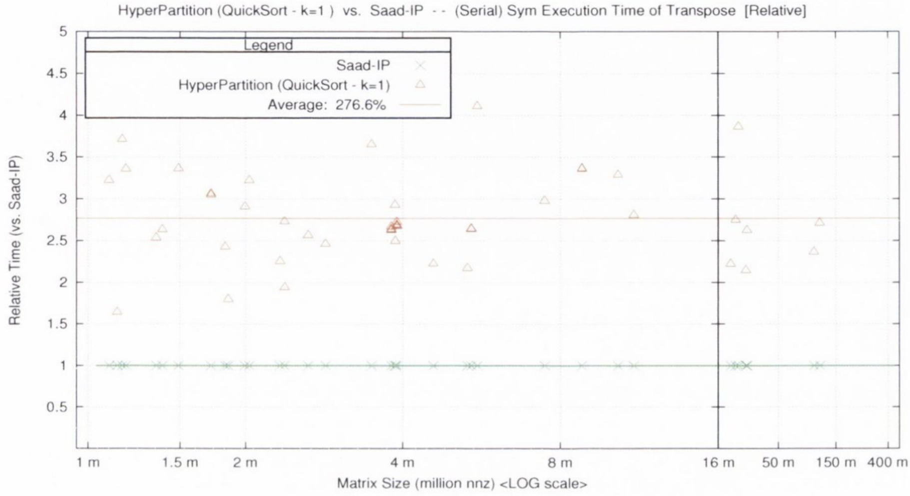


Figure B.4 (a): Serial Symmetric HyperPartition with QuickSort: $k = 1$

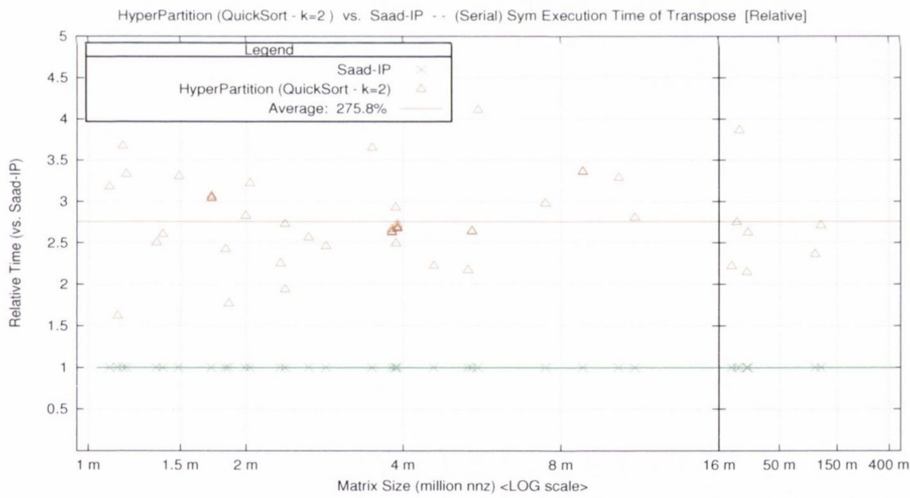


Figure B.4 (b): Serial Symmetric HyperPartition with QuickSort: $k = 2$

Figure B.4 (c,d): Serial Symmetric HyperPartition with QuickSort: $k = 1 \to 10$

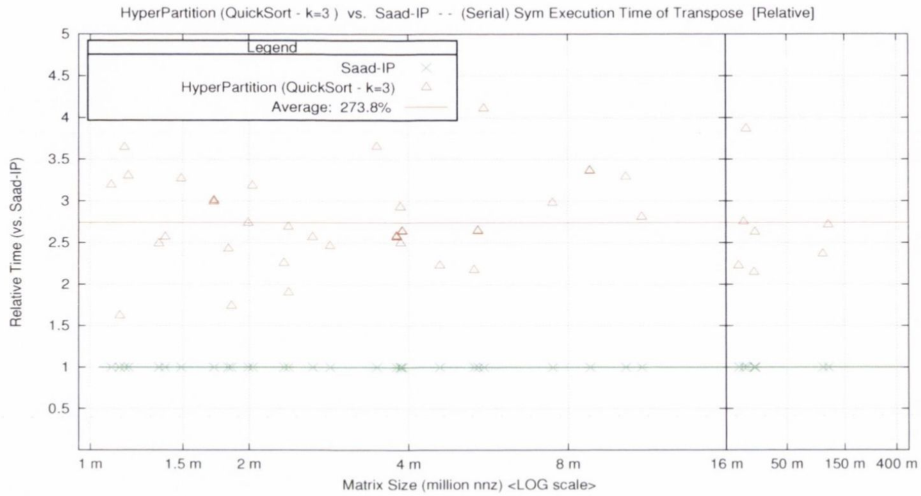


Figure B.4 (c): Serial Symmetric HyperPartition with QuickSort: $k = 3$

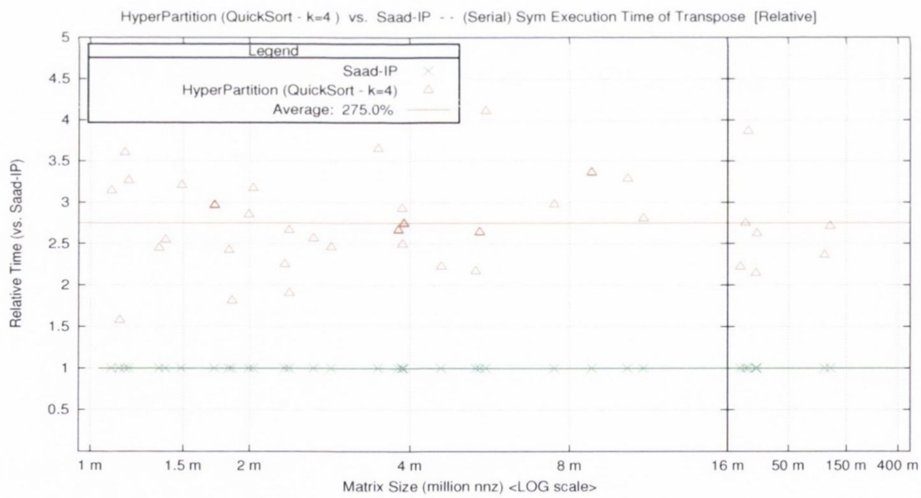


Figure B.4 (d): Serial Symmetric HyperPartition with QuickSort: $k = 4$

Figure B.4 (e,f): Serial Symmetric HyperPartition with QuickSort: $k = 1 \rightarrow 10$

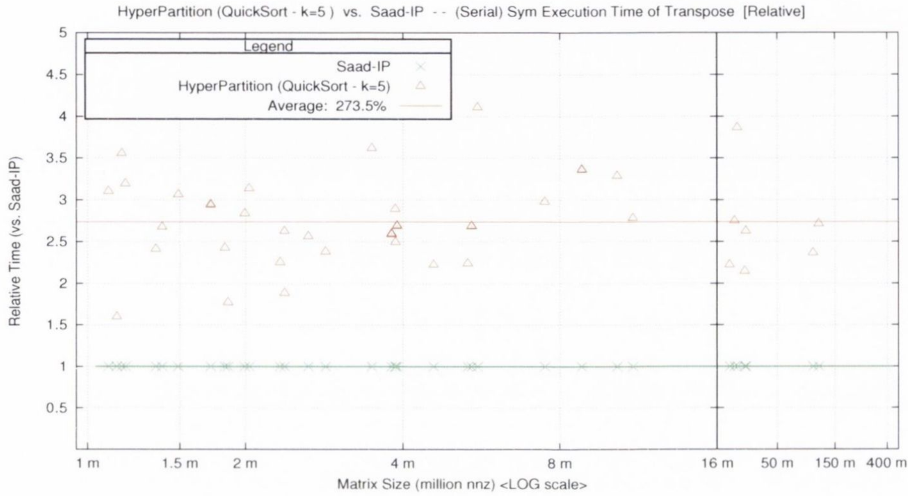


Figure B.4 (e): Serial Symmetric HyperPartition with QuickSort: $k = 5$

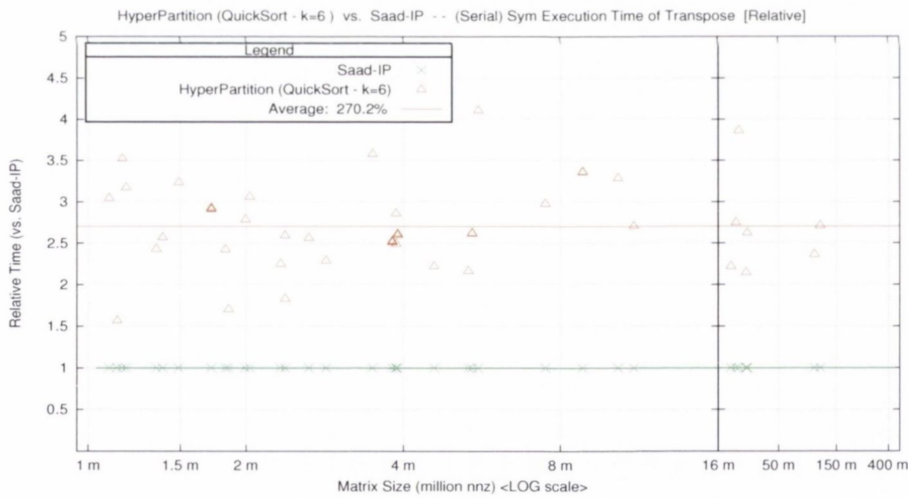


Figure B.4 (f): Serial Symmetric HyperPartition with QuickSort: $k = 6$

Figure B.4 (g,h): Serial Symmetric HyperPartition with QuickSort: $k = 1 \rightarrow 10$

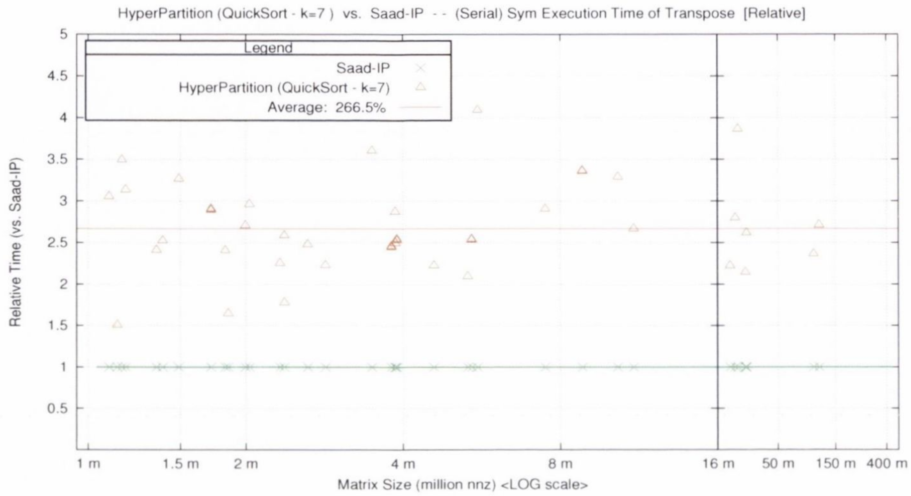


Figure B.4 (g): Serial Symmetric HyperPartition with QuickSort: $k = 7$

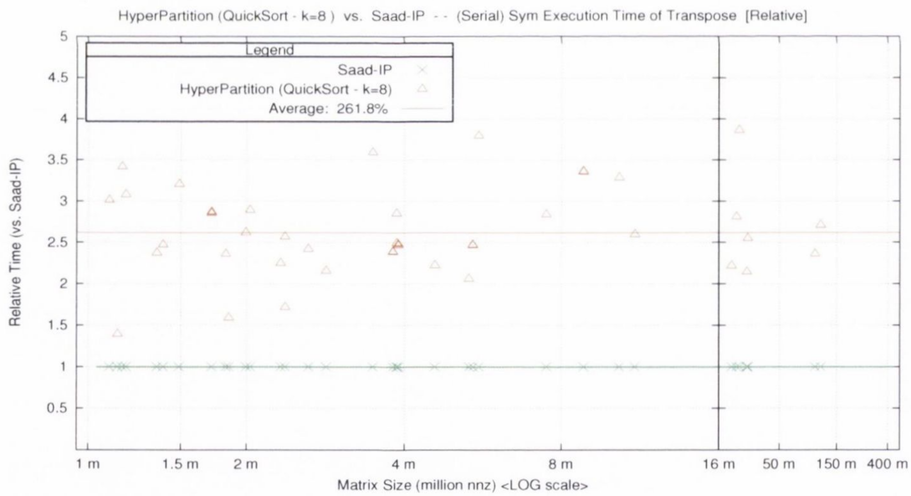


Figure B.4 (h): Serial Symmetric HyperPartition with QuickSort: $k = 8$

Figure B.4 (i,j): Serial Symmetric HyperPartition with QuickSort: $k = 1 \rightarrow 10$

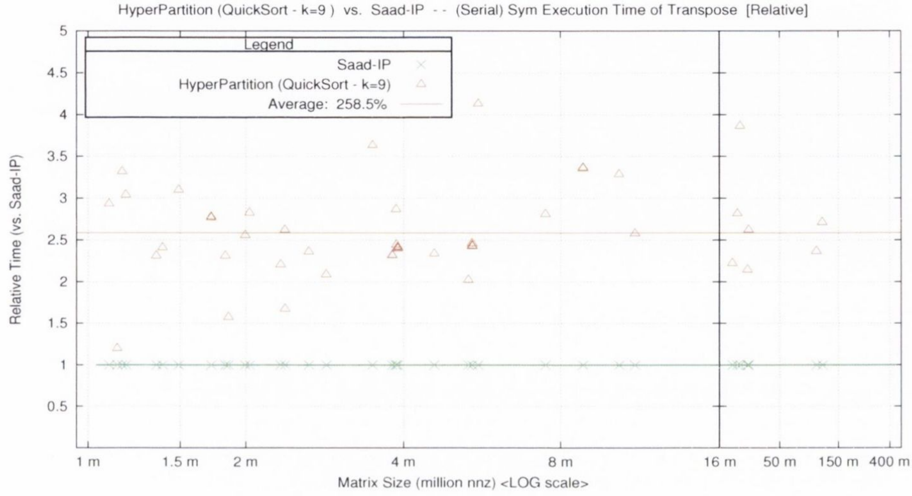


Figure B.4 (i): Serial Symmetric HyperPartition with QuickSort: $k = 9$

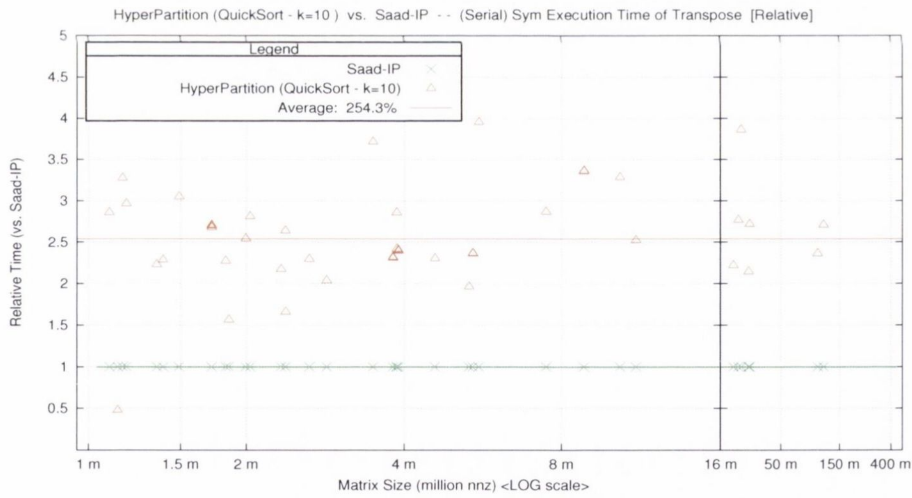


Figure B.4 (j): Serial Symmetric HyperPartition with QuickSort: $k = 10$

Figure B.5 (a,b): Serial HyperPartition with RadixSort: $k = 1 \rightarrow 10$

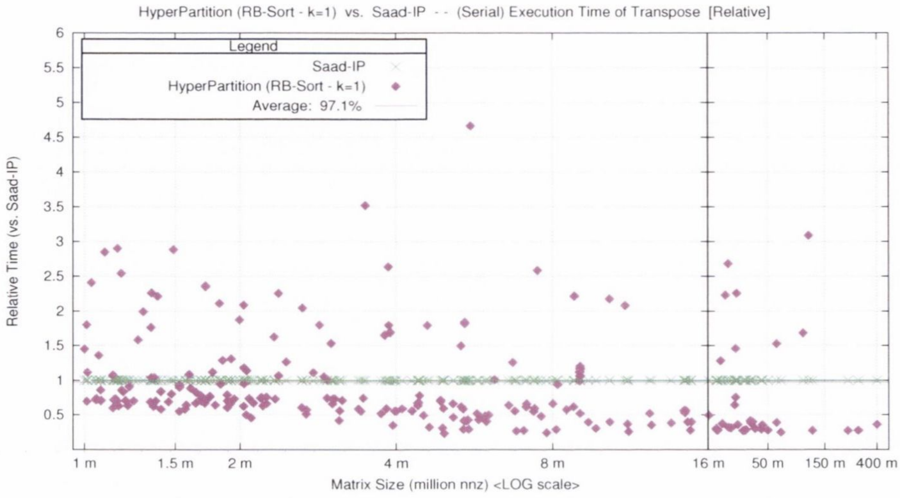


Figure B.5 (a): Serial HyperPartition with RadixSort: $k = 1$

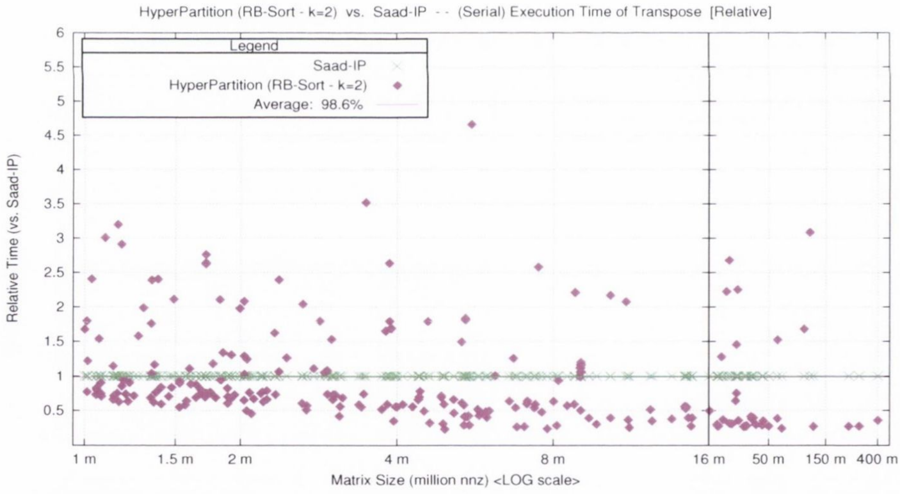


Figure B.5 (b): Serial HyperPartition with RadixSort: $k = 2$

Figure B.5 (c,d): Serial HyperPartition with RadixSort: $k = 1 \rightarrow 10$

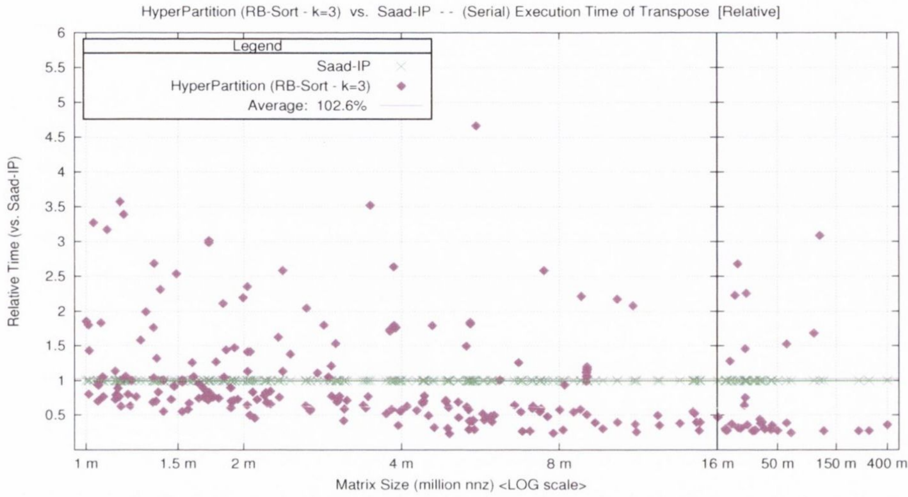


Figure B.5 (c): Serial HyperPartition with RadixSort: $k = 3$

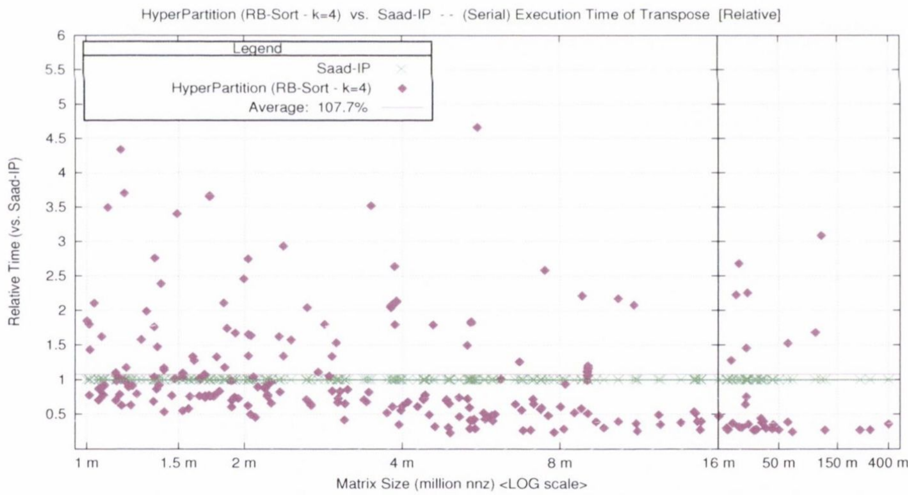


Figure B.5 (d): Serial HyperPartition with RadixSort: $k = 4$

Figure B.5 (e,f): Serial HyperPartition with RadixSort: $k = 1 \rightarrow 10$

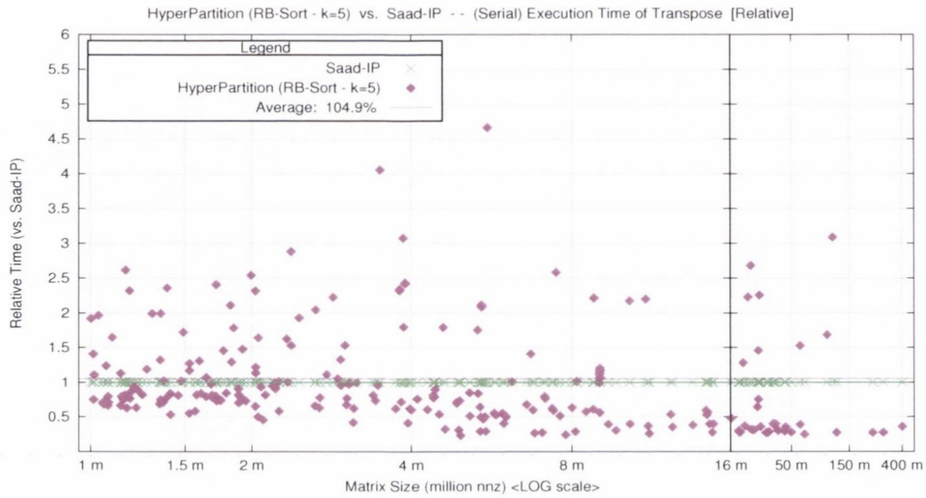


Figure B.5 (e): Serial HyperPartition with RadixSort: $k = 5$

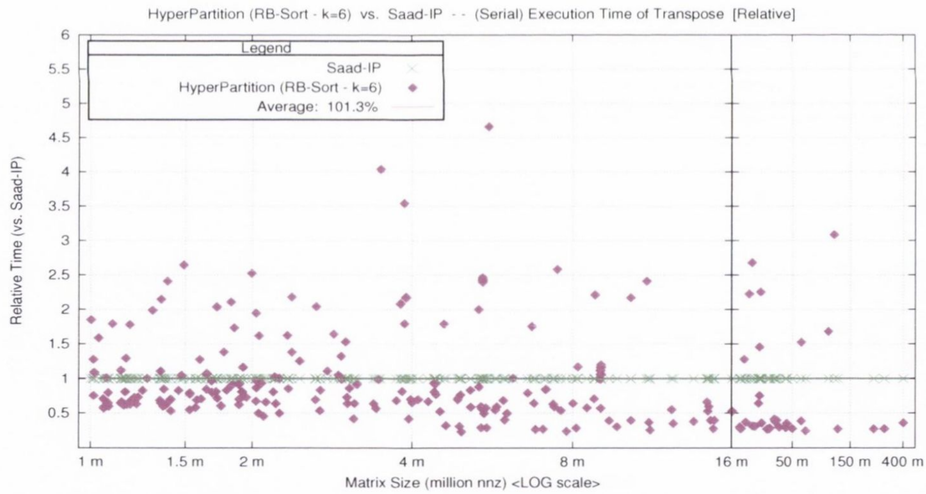


Figure B.5 (f): Serial HyperPartition with RadixSort: $k = 6$

Figure B.5 (g,h): Serial HyperPartition with RadixSort: $k = 1 \rightarrow 10$

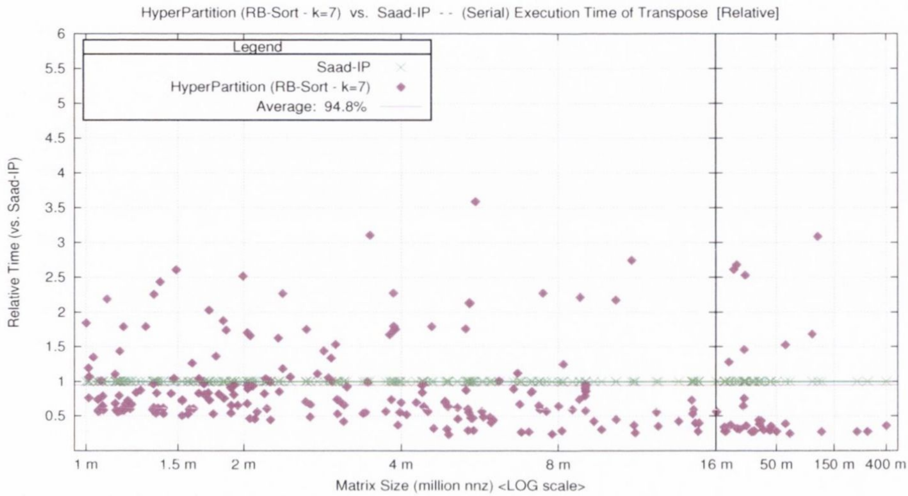


Figure B.5 (g): Serial HyperPartition with RadixSort: $k = 7$

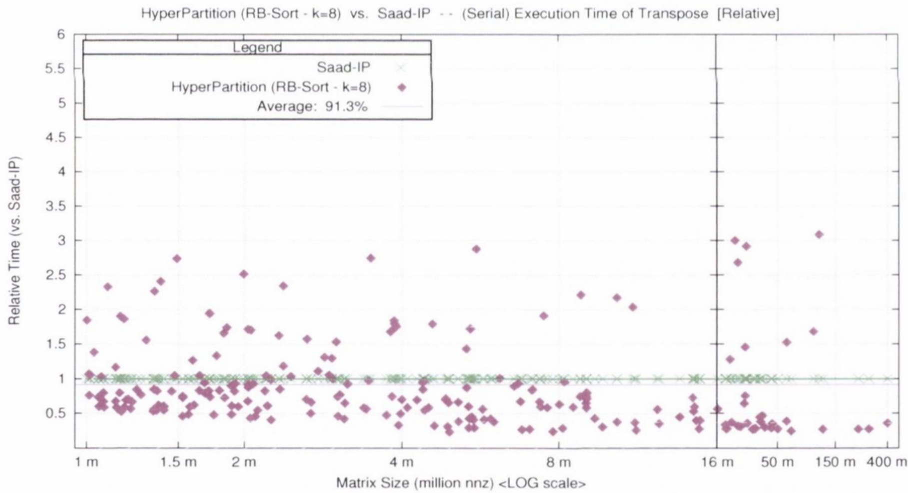


Figure B.5 (h): Serial HyperPartition with RadixSort: $k = 8$

Figure B.5 (i,j): Serial HyperPartition with RadixSort: $k = 1 \rightarrow 10$

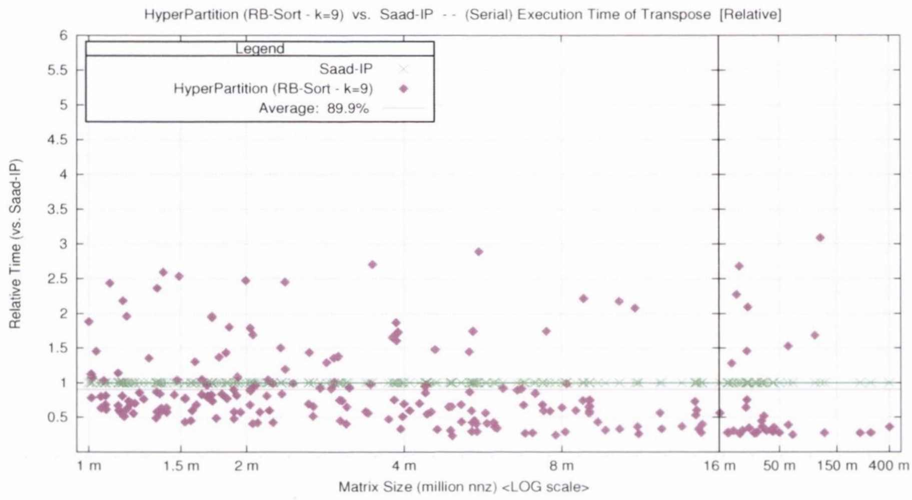


Figure B.5 (i): Serial HyperPartition with RadixSort: $k = 9$

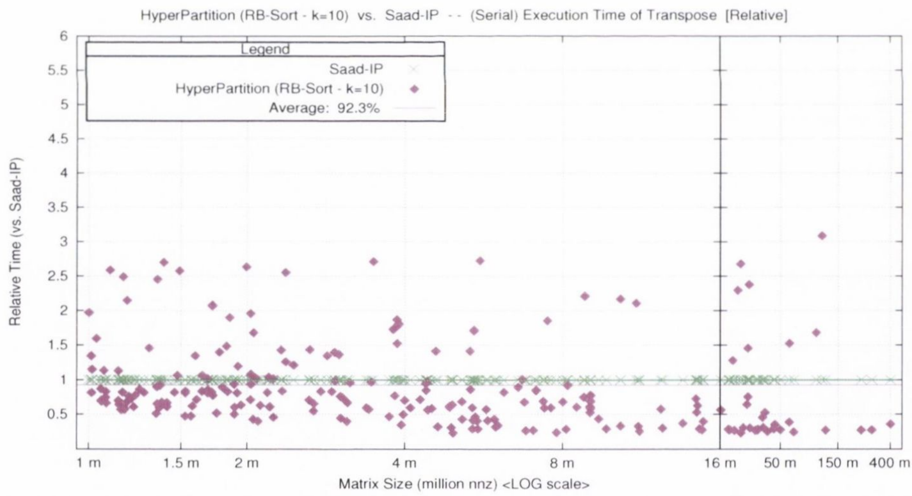


Figure B.5 (j): Serial HyperPartition with RadixSort: $k = 10$

Figure B.6 (a,b): Parallel HyperPartition with RadixSort: $k = 1 \rightarrow 10$

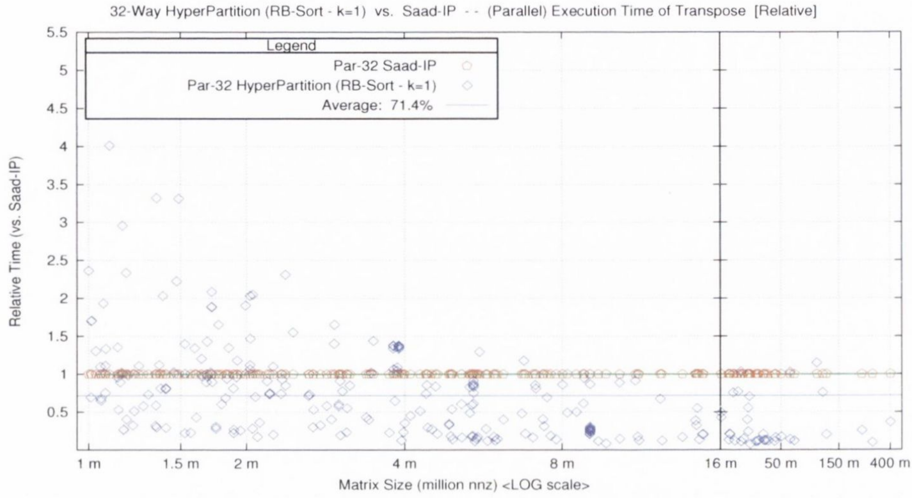


Figure B.6 (a): Parallel HyperPartition with RadixSort: $k = 1$

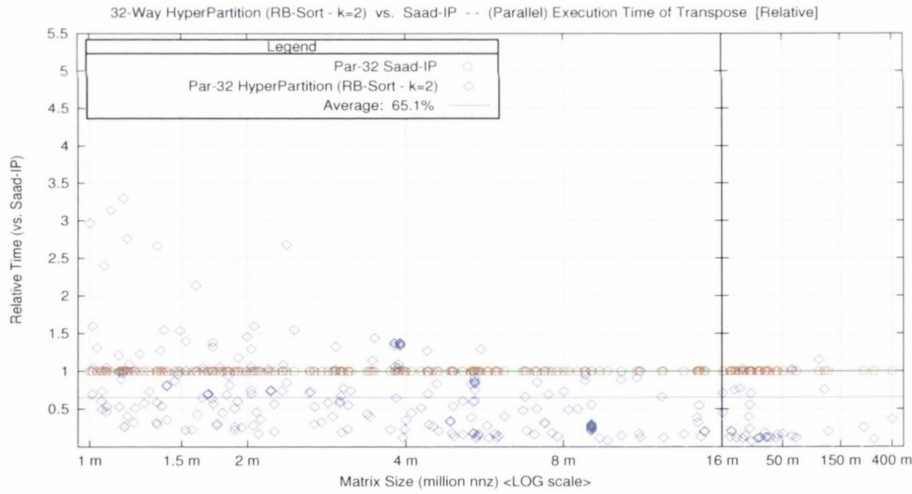


Figure B.6 (b): Parallel HyperPartition with RadixSort: $k = 2$

Figure B.6 (c,d): Parallel HyperPartition with RadixSort: $k = 1 \to 10$

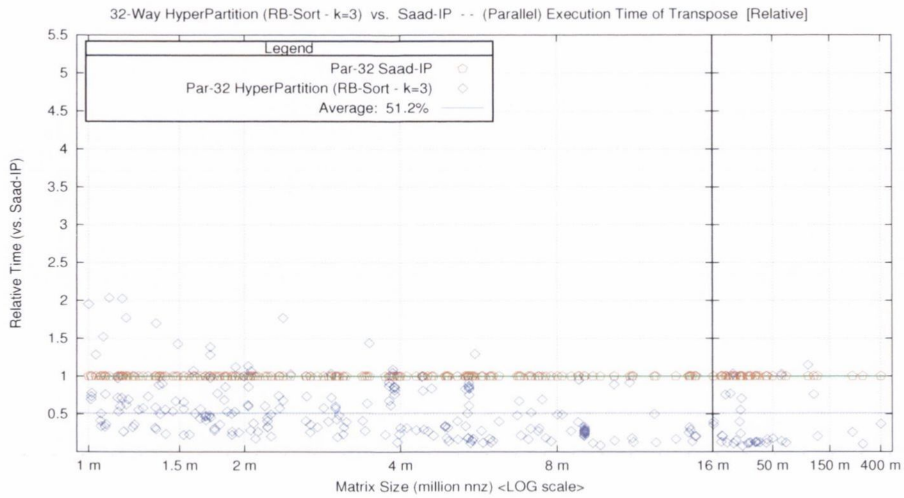


Figure B.6 (c): Parallel HyperPartition with RadixSort: $k = 3$

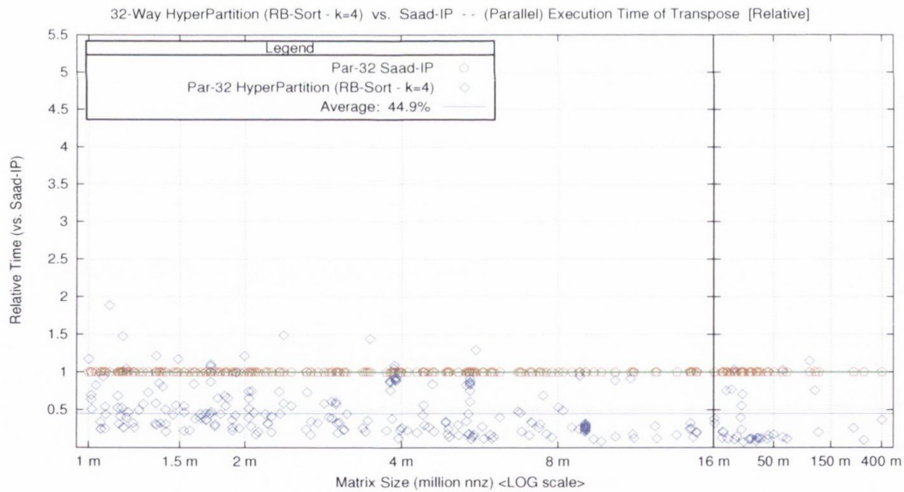


Figure B.6 (d): Parallel HyperPartition with RadixSort: $k = 4$

Figure B.6 (e,f): Parallel HyperPartition with RadixSort: $k = 1 \rightarrow 10$

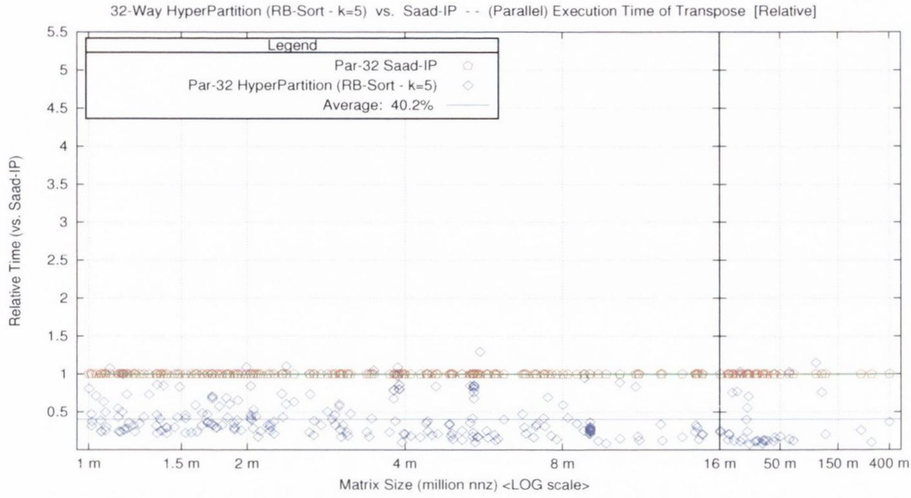


Figure B.6 (e): Parallel HyperPartition with RadixSort: $k = 5$

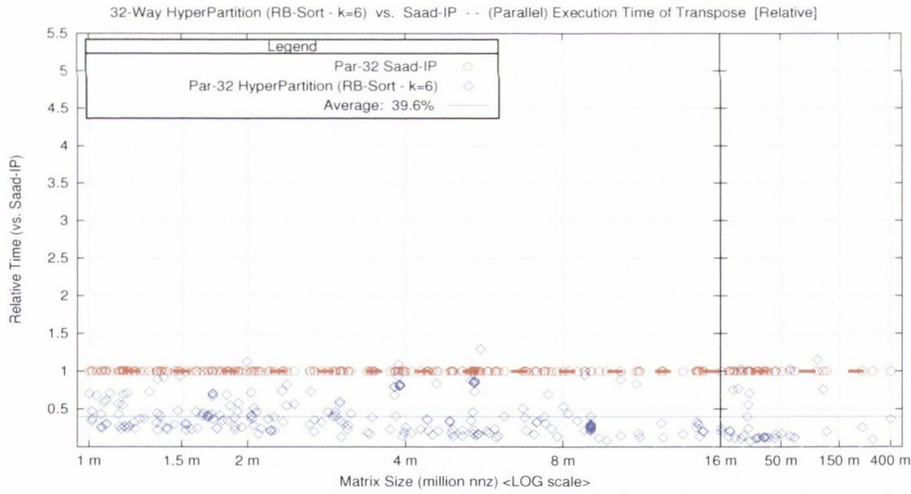


Figure B.6 (f): Parallel HyperPartition with RadixSort: $k = 6$

Figure B.6 (g,h): Parallel HyperPartition with RadixSort: $k = 1 \rightarrow 10$

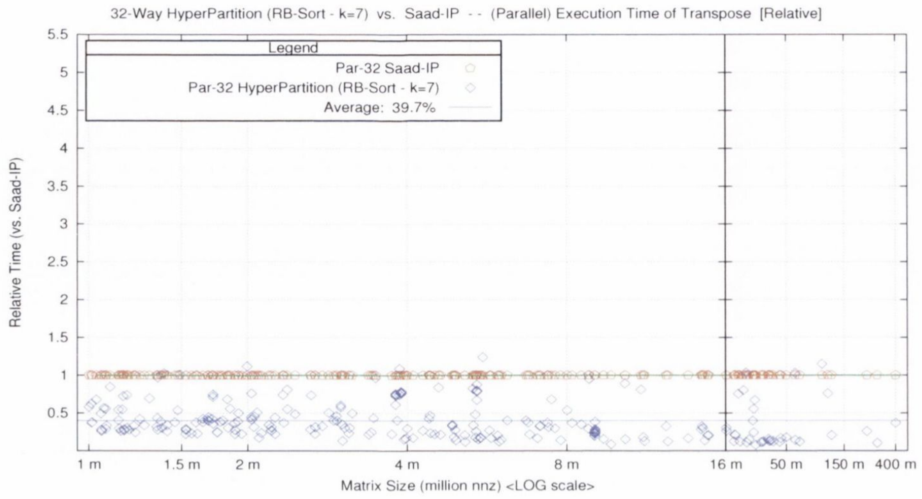


Figure B.6 (g): Parallel HyperPartition with RadixSort: $k = 7$

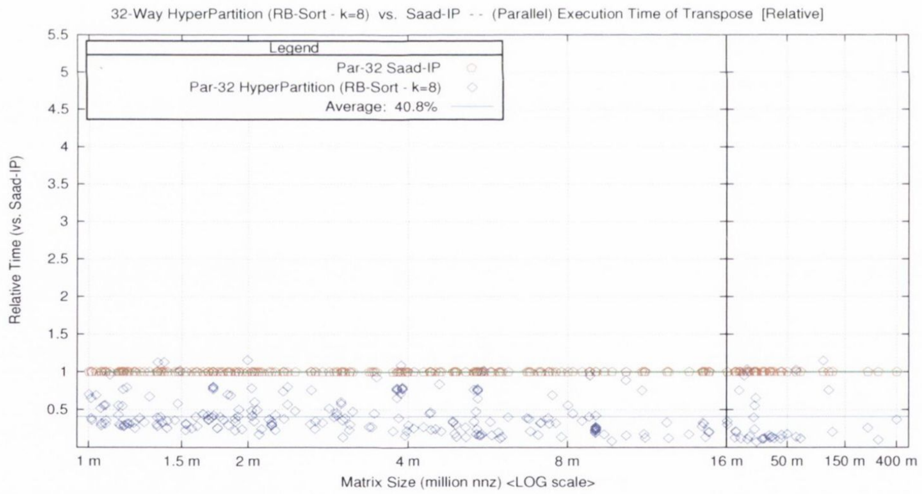


Figure B.6 (h): Parallel HyperPartition with RadixSort: $k = 8$

Figure B.6 (i,j): Parallel HyperPartition with RadixSort: $k = 1 \to 10$

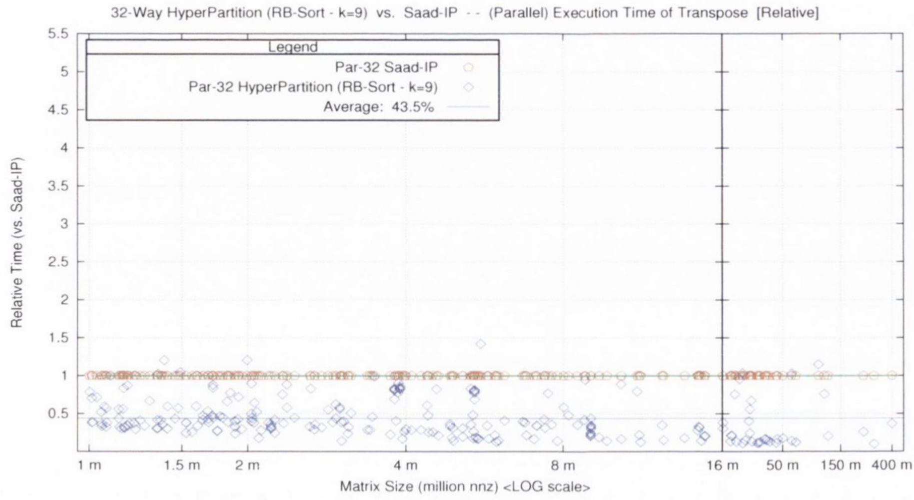


Figure B.6 (i): Parallel HyperPartition with RadixSort: $k = 9$

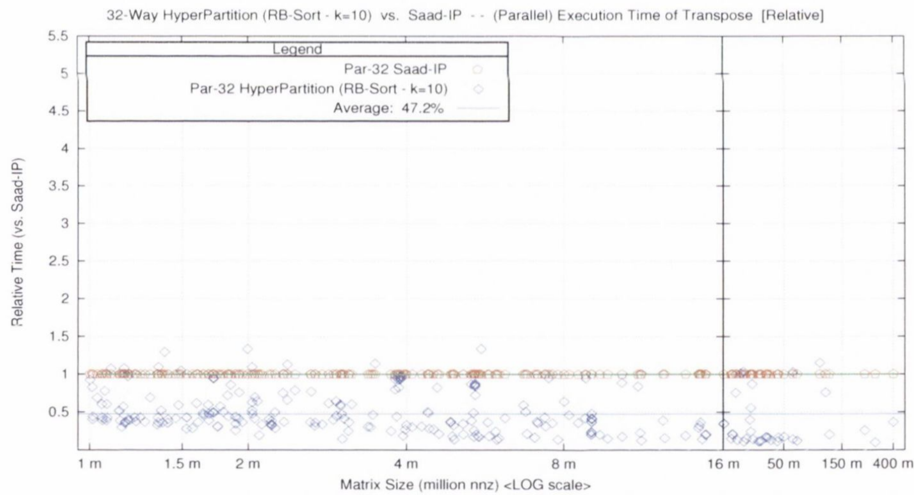


Figure B.6 (j): Parallel HyperPartition with RadixSort: $k = 10$

Figure B.7 (a,b): Serial Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

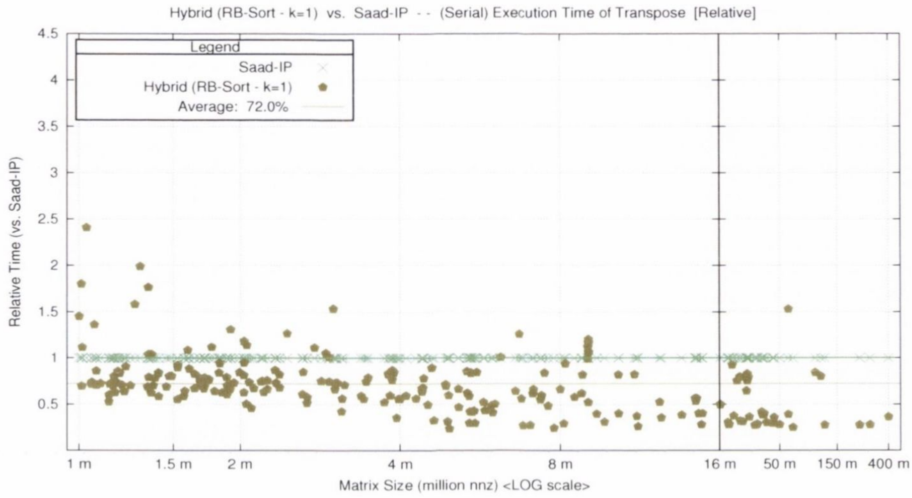


Figure B.7 (a): Serial Hybrid with RadixSort leaving: $k = 1$

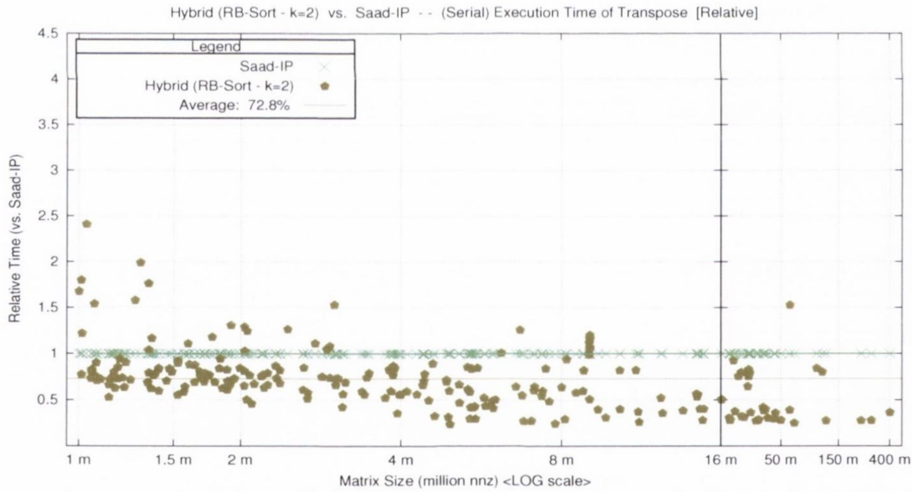


Figure B.7 (b): Serial Hybrid with RadixSort leaving: $k = 2$

Figure B.7 (c,d): Serial Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

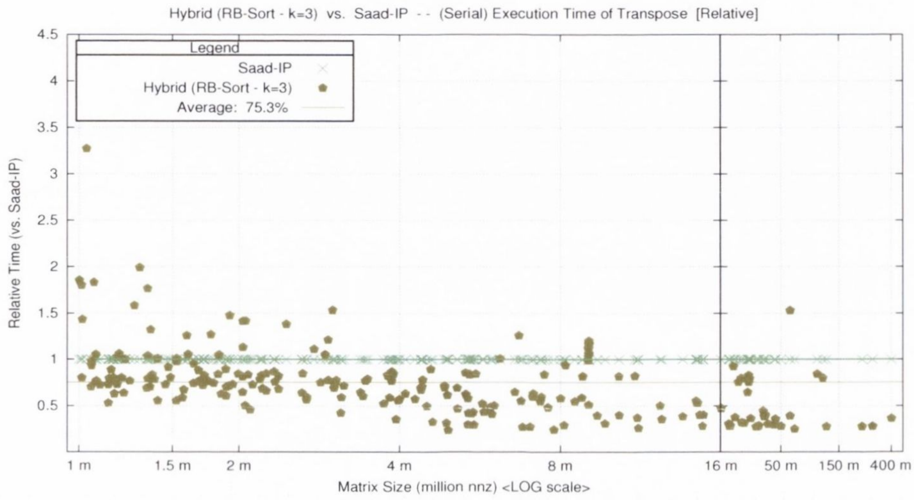


Figure B.7 (c): Serial Hybrid with RadixSort leaving: $k = 3$

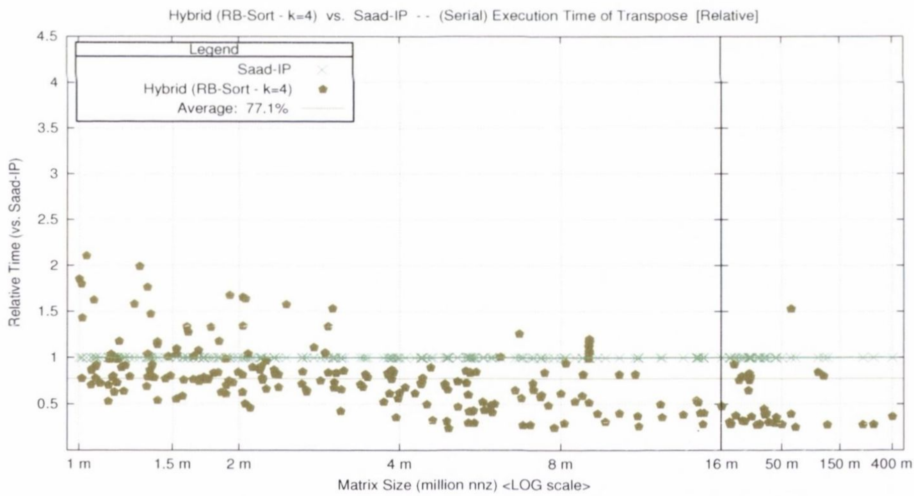


Figure B.7 (d): Serial Hybrid with RadixSort leaving: $k = 4$

Figure B.7 (e,f): Serial Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

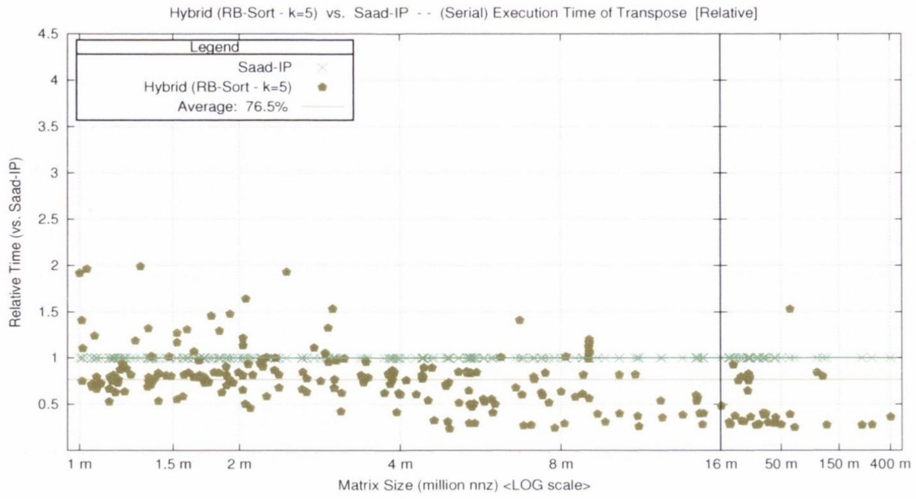


Figure B.7 (e): Serial Hybrid with RadixSort leaving: $k = 5$

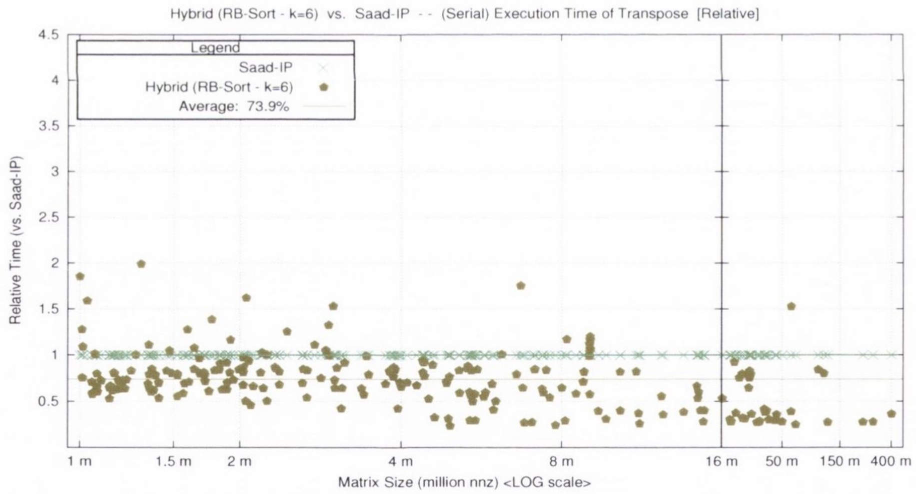


Figure B.7 (f): Serial Hybrid with RadixSort leaving: $k = 6$

Figure B.7 (g,h): Serial Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

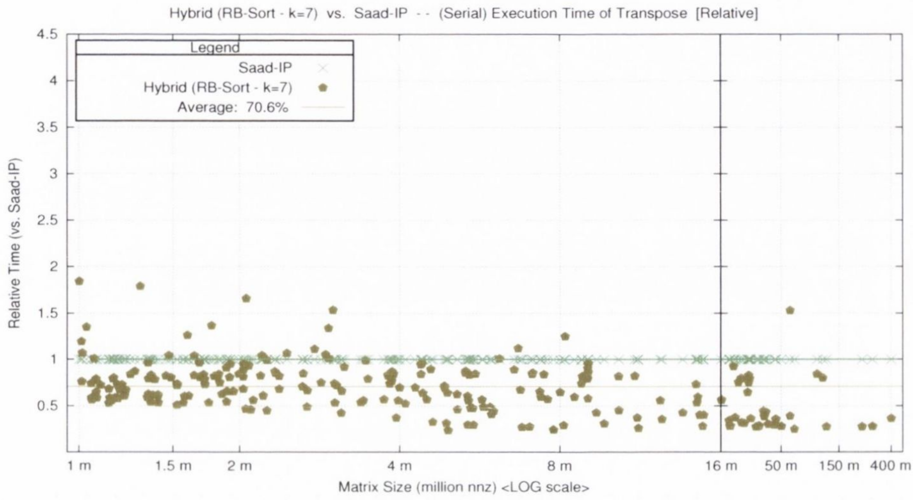


Figure B.7 (g): Serial Hybrid with RadixSort leaving: $k = 7$

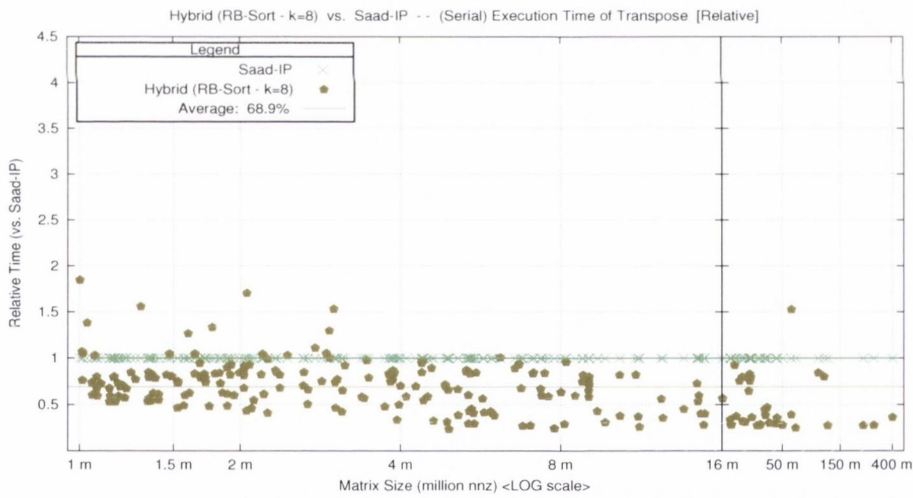


Figure B.7 (h): Serial Hybrid with RadixSort leaving: $k = 8$

Figure B.7 (i,j): Serial Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

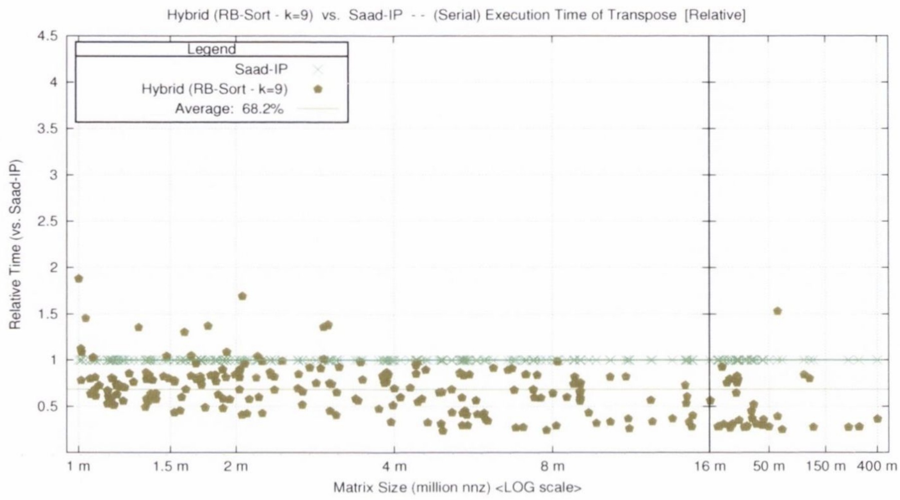


Figure B.7 (i): Serial Hybrid with RadixSort leaving: $k = 9$

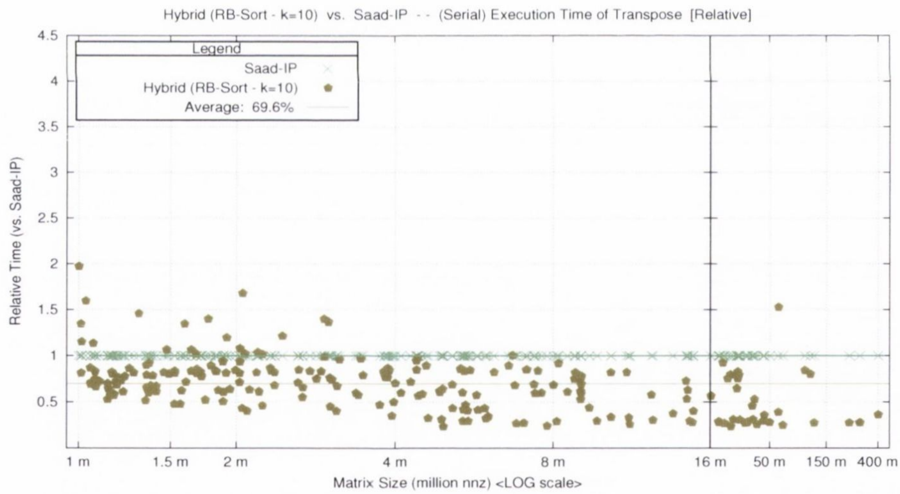


Figure B.7 (j): Serial Hybrid with RadixSort leaving: $k = 10$

Figure B.8 (a,b): Parallel Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

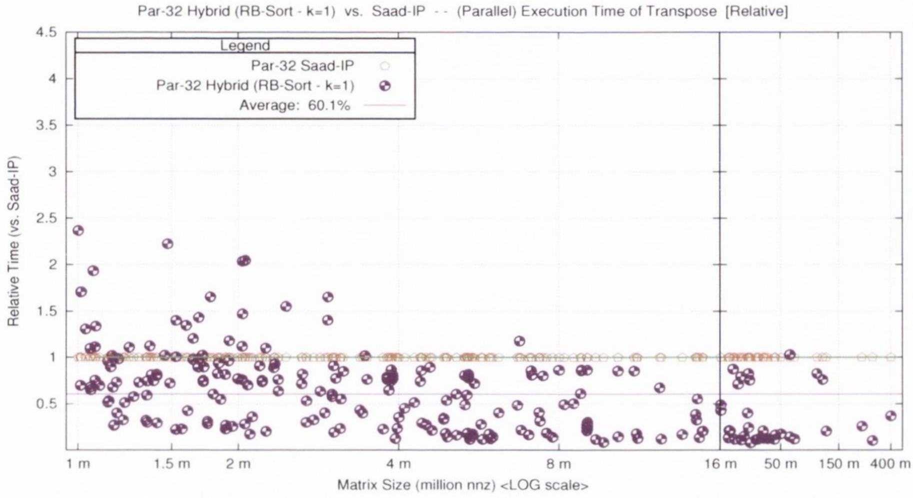


Figure B.8 (a): Parallel Hybrid with RadixSort leaving: $k = 1$

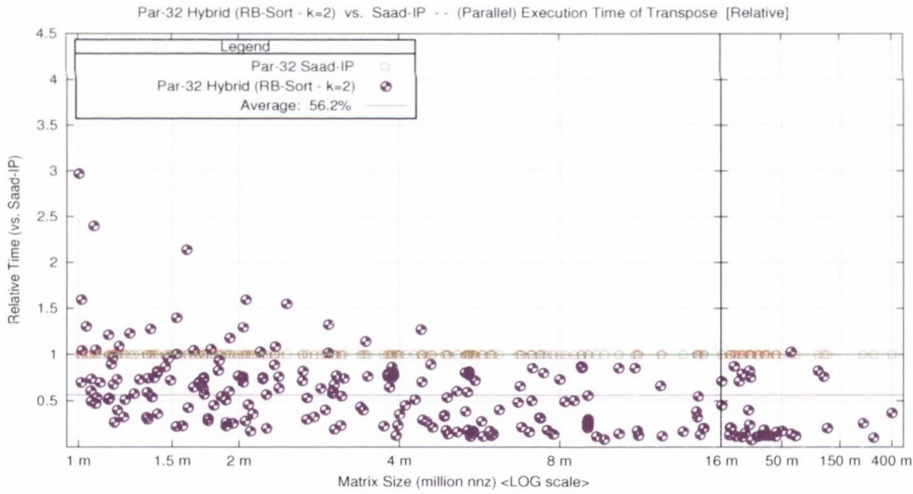


Figure B.8 (b): Parallel Hybrid with RadixSort leaving: $k = 2$

Figure B.8 (c,d): Parallel Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

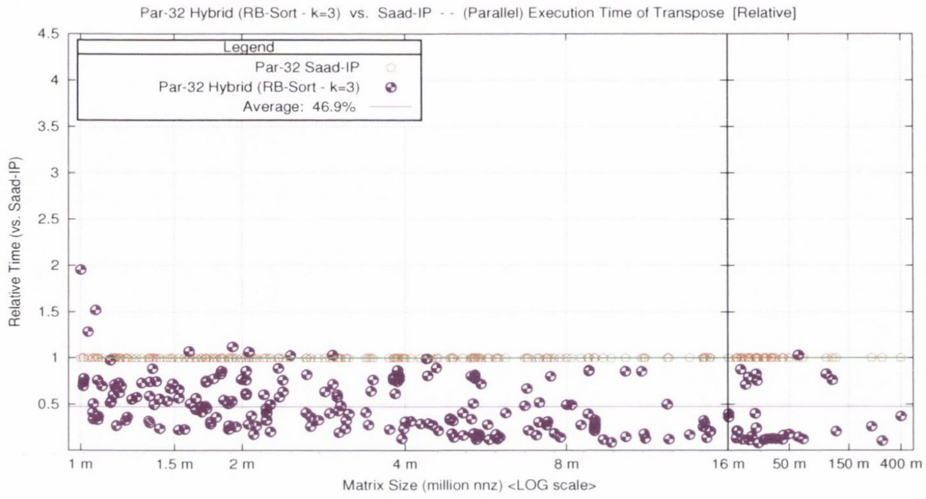


Figure B.8 (c): Parallel Hybrid with RadixSort leaving: $k = 3$

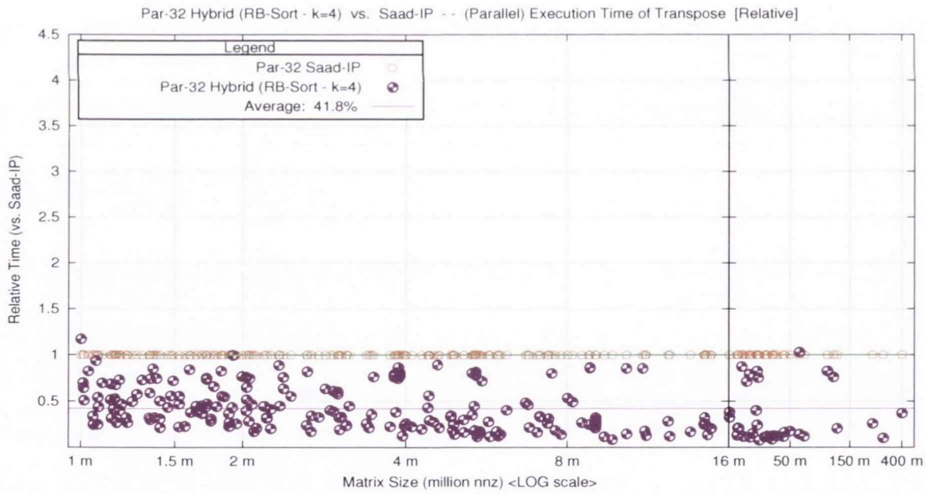


Figure B.8 (d): Parallel Hybrid with RadixSort leaving: $k = 4$

Figure B.8 (e,f): Parallel Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

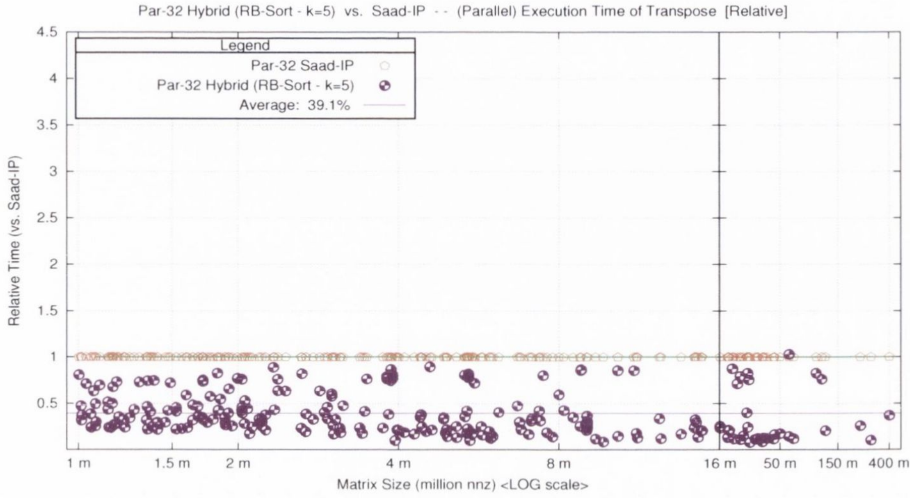


Figure B.8 (e): Parallel Hybrid with RadixSort leaving: $k = 5$

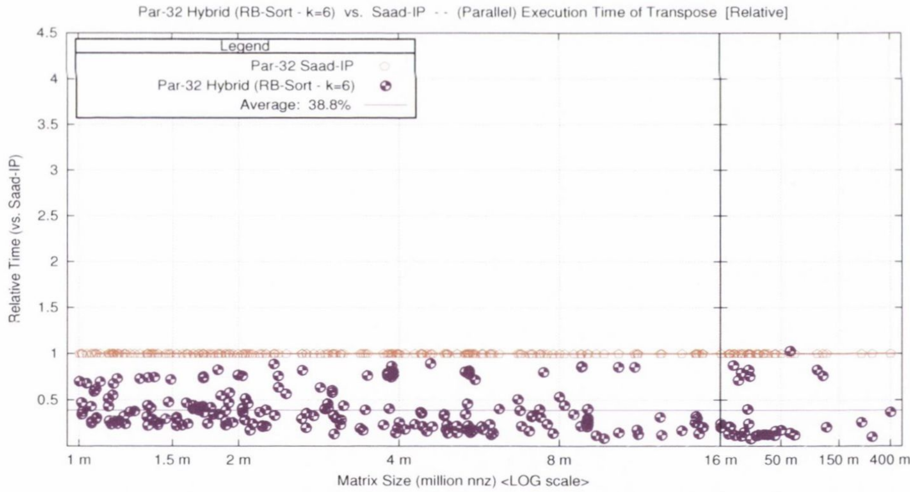


Figure B.8 (f): Parallel Hybrid with RadixSort leaving: $k = 6$

Figure B.8 (g,h): Parallel Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

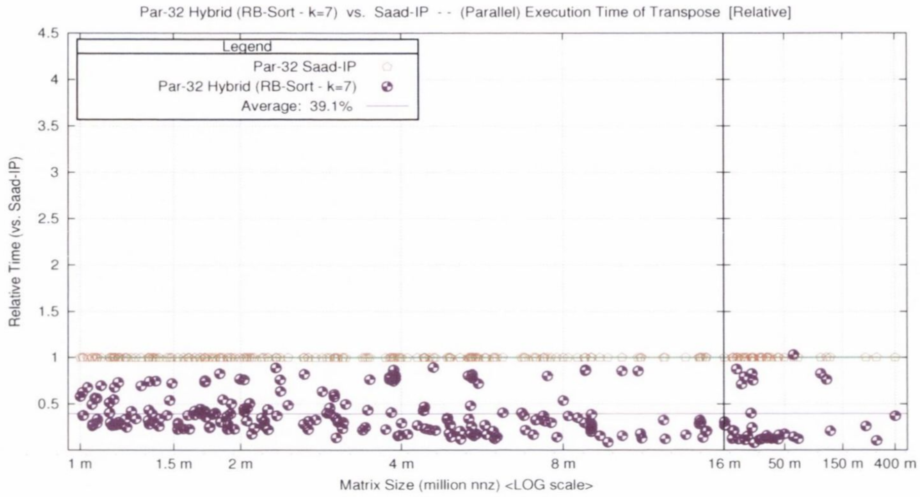


Figure B.8 (g): Parallel Hybrid with RadixSort leaving: $k = 7$

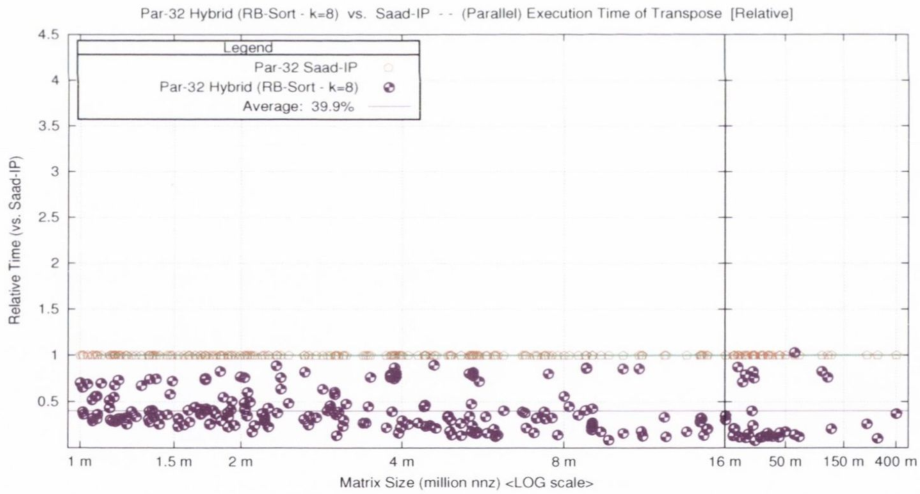


Figure B.8 (h): Parallel Hybrid with RadixSort leaving: $k = 8$

Figure B.8 (i,j): Parallel Hybrid with RadixSort leaving: $k = 1 \rightarrow 10$

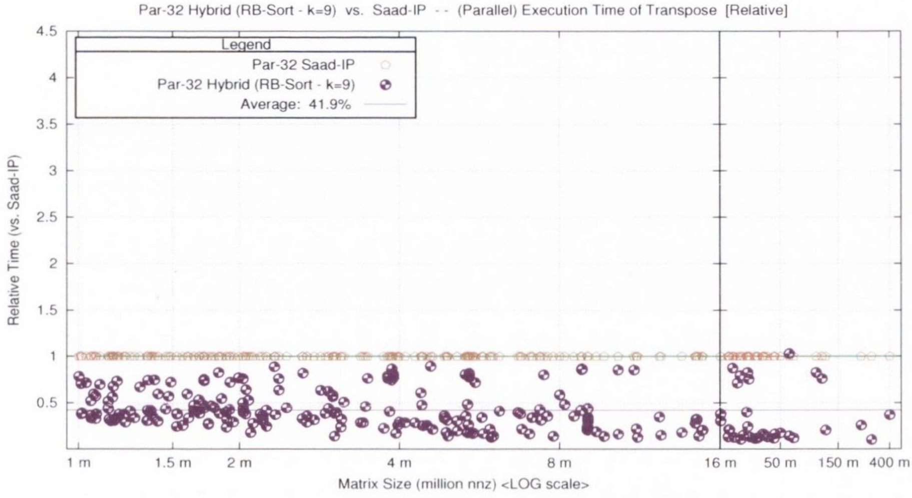


Figure B.8 (i): Parallel Hybrid with RadixSort leaving: $k = 9$

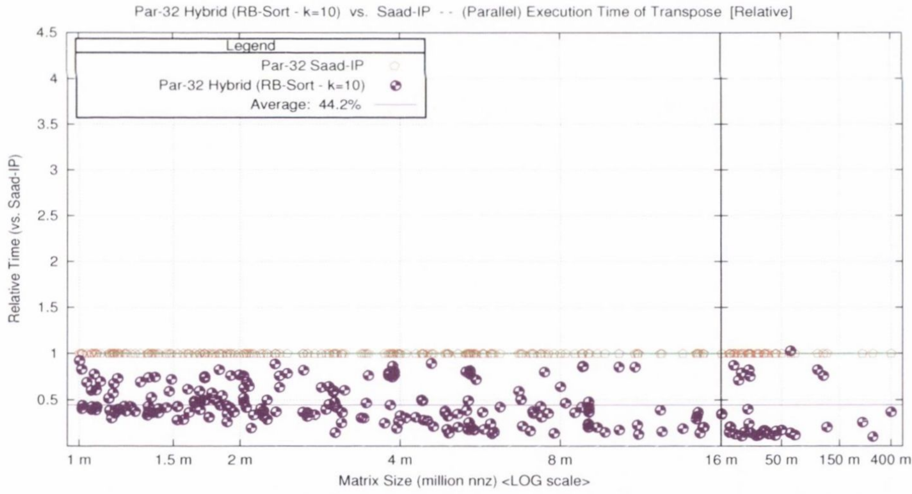
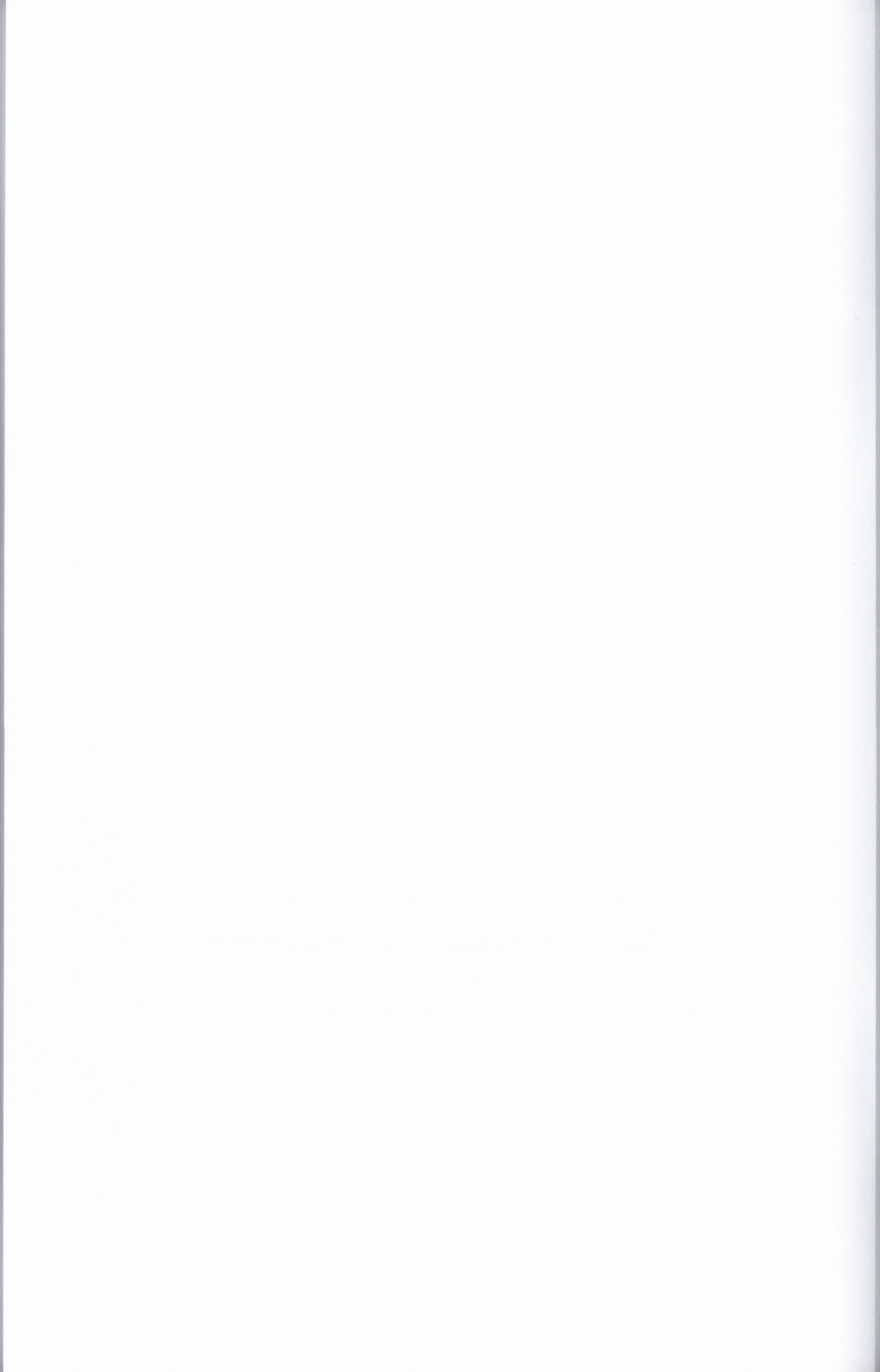


Figure B.8 (j): Parallel Hybrid with RadixSort leaving: $k = 10$



Appendix C

MatrixMarket File Format

There are a number of different file formats which can be used to persistently store sparse matrices on disk. There are binary formats and text formats, some formats require more overhead than others and some are more flexible. The matrices used in our test suite (Section 3.6.1) were obtained in the MatrixMarket [Boisvert 95, Boisvert 97] file format. The MatrixMarket file essentially stores the elements in the matrix in a format similar to the Compressed Coordinate format (Section 2.3.4).

An example of the MatrixMarket file format for the matrix M is shown in Listing C.1. The file begins with the MatrixMarket header which is a single line that begins with **%%MatrixMarket** followed by a number of declarations which define the structure of the matrix contained within the file. The first non-comment, non-header line contains 3 integer numbers which declare the size of the matrix by giving respectively "the number of rows", "the number of columns" and "the number of non-zeros" in the

```
%%MatrixMarket matrix coordinate real general
6 6 15
1 1 a
1 5 b
2 1 c
2 2 d
2 6 e
3 2 f
3 3 g
4 1 h
4 4 i
4 5 j
5 5 k
5 6 l
6 2 m
6 5 n
6 6 o
```

Listing C.1: MatrixMarket File Format

matrix. The format of the remaining lines of the file will be determined by the contents of the MatrixMarket header.

In the case of "matrix coordinate real general" which is a normal(general) format matrix, with "real" values (as against integer or complex) each line contains two integers and a floating point (real) number. The integers specify the row and column coordinates respectively of the location of the floating point value in the matrix. The MatrixMarket format is *one-indexed*, the indexes start at 1 rather than 0 as in our examples. Again we use the letters a through o to represent the non-zero values in the matrix.

Bibliography

[Aggarwal 87] Alok Aggarwal, Ashok K. Chandra & Marc Snir. *Hierarchical memory with block transfer.* In Proceedings of the 28th Annual Symposium on Foundations of Computer Science, SFCS '87, pages 204–216, Washington, DC, USA, 1987. IEEE Computer Society.

[Aggarwal 88] Alok Aggarwal & S. Vitter Jeffrey. *The input/output complexity of sorting and related problems.* Commun. ACM, vol. 31, no. 9, pages 1116–1127, September 1988.

[Aho 74] A.V. Aho, J.E. Hopcroft & J.D. Ullman. The design and analysis of computer algorithms. Addison-Wesley series in computer science and information processing. Addison-Wesley Pub. Co., 1974.

[Al Na'Mneh 05] R. Al Na'Mneh, W.D. Pan & R. Adhami. *Communication efficient adaptive matrix transpose algorithm for FFT on symmetric multiprocessors.* In System Theory, 2005. SSST '05. Proceedings of the Thirty-Seventh Southeastern Symposium on, pages 312–315, 2005.

[Alltop 75] W. O. Alltop. *A Computer Algorithm for Transposing Nonsquare Matrices.* IEEE Trans. Comput., vol. 24, pages 1038–1040, October 1975.

[AMD 03] AMD. *ACML: AMD Core Math Library*, 2003.

[Amestoy 96] Patrick R. Amestoy, Timothy A. Davis & Iain S. Duff. *An Approximate Minimum Degree Ordering Algorithm.* SIAM J. Matrix Anal. Appl., vol. 17, no. 4, pages 886–905, 1996.

[Amestoy 98] P. Amestoy, I. Duff & J. L'Excellent. *Mumps multifrontal massively parallel solver version*, 1998.

[And 62] And. *An Algorithm for the Organization of Information.* Doklady Akademii Nauk USSR, vol. 146, no. 2, pages 263–266, 1962.

[Anderson 90] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof & D. Sorensen. *LAPACK: a portable linear algebra library for high-performance computers.* In Proceedings of the 1990 ACM/IEEE conference on

Supercomputing, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[Anderson 99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney & D. Sorensen. LAPACK users' guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[Anton 02] Howard Anton & Robert C. Busby. Contemporary linear algebra. Wiley, 2002.

[Ari 79] M.B. Ari. *On Transposing Large 2^n x 2^n Matrices.* Computers, IEEE Transactions on, vol. C-28, no. 1, pages 72–75, 1979.

[Arora 09] Sanjeev Arora & Boaz Barak. Computational complexity: A modern approach. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[Ashcraft 99] Cleve Ashcraft & Roger Grimes. *SPOOLES: An Object-Oriented Sparse Matrix Library.* In In Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing, page pages, 1999.

[Bachmann 23] P.G.H. Bachmann. Zahlentheorie: th. die arithmetik de quadratischen formen (1898). Zahlentheorie: Versuch einer Gesammtdarstellung dieser Wissenschaft in ihren Haupttheilen. B. G. Teubner, 1923.

[Backus 56] J. W. Backus. The fortran automatic coding system for the ibm 704 edpm. IBM Corp., 1956.

[Backus 57] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes & R. Nutt. *The FORTRAN automatic coding system.* In Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA, 1957. ACM.

[Bader 07] Michael Bader & Christian Mayer. *Cache Oblivious Matrix Operations Using Peano Curves.* In Bo Kagstrom, Erik Elmroth, Jack Dongarra & Jerzy Wasniewski, editeurs, Applied Parallel Computing. State of the Art in Scientific Computing, volume 4699 of *Lecture Notes in Computer Science*, pages 521–530. Springer Berlin Heidelberg, 2007.

[Baker 92] Henry G. Baker Jr. *On the permutations of a vector obtainable through the restructure and transpose functions of APL.* SIGAPL APL Quote Quad, vol. 23, no. 2, pages 27–32, December 1992.

[Balay 97] Satish Balay, William D. Gropp, Lois Curfman McInnes & Barry F. Smith. *Efficient Management of Parallelism in Object Oriented Numerical Software Libraries.* In E. Arge, A. M. Bruaset & H. P. Langtangen, editeurs, Modern Software Tools in Scientific Computing, pages 163–202. Birkhäuser Press, 1997.

[Baumstark 03] Lewis Baumstark, Murat Guler & Linda Wills. *Extracting an Explicitly Data-Parallel Representation of Image-Processing Programs.* In Proceedings of the 10th Working Conference on Reverse Engineering, WCRE '03, pages 24–, Washington, DC, USA, 2003. IEEE Computer Society.

[Bayer 70] R. Bayer & E. McCreight. *Organization and maintenance of large ordered indices.* In Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[Berman 58] Martin F. Berman. *A Method for Transposing a Matrix.* J. ACM, vol. 5, no. 4, pages 383–384, October 1958.

[Biggar 08a] Paul Biggar, Nicholas Nash, Kevin Williams & David Gregg. *An experimental study of sorting and branch prediction.* J. Exp. Algorithmics, vol. 12, pages 1.8:1–1.8:39, June 2008.

[Biggar 08b] Paul Biggar, Nicholas Nash, Kevin Williams & David Gregg. *An experimental study of sorting and branch prediction.* J. Exp. Algorithmics, vol. 12, pages 1–39, 2008.

[Bikshandi 06] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua & Christoph von Praun. *Programming for Parallelism and Locality with Hierarchically Tiled Arrays.* In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM.

[Blumofe 95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall & Yuli Zhou. *Cilk: An Efficient Multithreaded Runtime System.* In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of

Parallel Programming, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[Boisvert 95] Ronald F. Boisvert, Ronald F. Boisvert, Roldan Pozo, Roldan Pozo, Karin A. Remington & Karin A. Remington. *The Matrix Market Exchange Formats: Initial Design.* NISTIR, vol. 5935, 1995.

[Boisvert 97] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett & Jack J. Dongarra. *Matrix Market: A Web Resource for Test Matrix Collections.* In The Quality of Numerical Software: Assessment and Enhancement, pages 125–137. Chapman & Hall, 1997.

[Boncz 08] Peter A. Boncz, Martin L. Kersten & Stefan Manegold. *Breaking the Memory Wall in MonetDB.* Commun. ACM, vol. 51, no. 12, pages 77–85, December 2008.

[Boothroyd 67] J. Boothroyd. *Algorithm 302: Transpose Vector Stored Array.* Commun. ACM, vol. 10, no. 5, pages 292–293, May 1967.

[Bornstein 99] Claudson F. Bornstein, Bruce M. Maggs & Gary L. Miller. *Tradeoffs between parallelism and fill in nested dissection.* In SPAA '99: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures, pages 191–200, New York, NY, USA, 1999. ACM.

[Bosma 97] Wieb Bosma, John Cannon & Catherine Playoust. *The Magma algebra system. I. The user language.* J. Symbolic Comput., vol. 24, no. 3-4, pages 235–265, 1997. Computational algebra and number theory (London, 1993).

[Brenner 73] Norman Brenner. *Algorithm 467: matrix transposition in place [F1].* Commun. ACM, vol. 16, no. 11, pages 692–694, November 1973.

[Brodal 05] GerthStlting Brodal & Gabriel Moruz. *Tradeoffs Between Branch Mispredictions and Comparisons for Sorting Algorithms.* In Frank Dehne, Alejandro Lpez-Ortiz & Jrg-Rdiger Sack, editeurs, Algorithms and Data Structures, volume 3608 of *Lecture Notes in Computer Science*, pages 385–395. Springer Berlin Heidelberg, 2005.

[Browne 00] S. Browne, J. Dongarra, N. Garner, G. Ho & P. Mucci. *A Portable Programming Interface for Performance Evaluation on Modern*

Processors. The International Journal of High Performance Computing Applications, vol. 14, no. 3, pages 189–204, fall 2000.

[Bruijn 70] N.G. Bruijn. Asymptotic methods in analysis. Bibliotheca mathematica. Dover Publications, 1970.

[Buluç 09] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert & Charles E. Leiserson. *Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks.* In Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, SPAA '09, pages 233–244, New York, NY, USA, 2009. ACM.

[Bunch 76] J. R. Bunch & D. J. Rose, editeurs. Sparse matrix computations. Academic Press, New York, NY, USA, 1976.

[Buttari 07] Alfredo Buttari, Julien Langou, Jakub Kurzak & Jack Dongarra. *A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures*, 2007.

[Calvin 96] C. Calvin. *Implementation of Parallel FFT Algorithms on Distributed Memory Machines with a Minimum Overhead of Communication.* Parallel Comput., vol. 22, no. 9, pages 1255–1279, November 1996.

[Cate 77a] Esko G. Cate & David W. Twigg. *Algorithm 513: Analysis of In-Situ Transposition [F1].* ACM Trans. Math. Softw., vol. 3, pages 104–110, March 1977.

[Cate 77b] Esko G. Cate & David W. Twigg. *Algorithm 513: Analysis of In-Situ Transposition [F1].* ACM Trans. Math. Softw., vol. 3, no. 1, pages 104–110, March 1977.

[Cayley 59] Arthur Cayley. *A memoir on the theory of matrices.* Philosophical Transactions of the Royal Society of London: Giving Some Accounts of the Present Undertakings, Studies, and Labours, of the Ingenious, in Many Considerable Parts of the World, vol. 148, pages pp. 17–37, 1859.

[Chatterjee 00] Siddhartha Chatterjee & Sandeep Sen. *Cache-Efficient Matrix Transposition.* In HPCA, pages 195–205. IEEE Computer Society, 2000.

[Chen 08] Yanqing Chen, Timothy A. Davis, William W. Hager & Sivasankaran Rajamanickam. *Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate.* ACM Transactions on Mathematical Software, vol. 35, no. 3, 2008.

[Chevalier 08] C. Chevalier & F. Pellegrini. *PT-Scotch: A tool for efficient parallel graph ordering.* Parallel Computing, vol. 34, no. 6-8, pages 318 – 331, 2008. Parallel Matrix Algorithms and Applications.

[Choi 95] Jaeyoung Choi, Jack J. Dongarra & David W. Walker. *Parallel matrix transpose algorithms on distributed memory concurrent computers.* Parallel Computing, vol. 21, no. 9, pages 1387 – 1405, 1995.

[Claasen 79] T. A C M Claasen & W. F G Mecklenbrauker. *Application of transposition to decimation and interpolation in digital signal processing systems.* In Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '79., volume 4, pages 832–835, 1979.

[Comer 79] Douglas Comer. *The ubiquitous B-tree.* ACM Computing Surveys, vol. 11, pages 121–137, 1979.

[Conrad 77] V. Conrad & Y. Wallach. *Iterative Solution of Linear Equations on a Parallel Processor System.* IEEE Trans. Comput., vol. 26, no. 9, pages 838–847, 1977.

[Cooley 65] James W. Cooley & John W. Tukey. *An Algorithm for the Machine Calculation of Complex Fourier Series.* Mathematics of Computation, vol. 19, no. 90, pages 297–301, 1965.

[Coppersmith 90] Don Coppersmith & Shmuel Winograd. *Matrix multiplication via arithmetic progressions.* Journal of Symbolic Computation, vol. 9, no. 3, pages 251 – 280, 1990. ¡ce:title¿Computational algebraic complexity editorial¡/ce:title¿.

[Cuthill 69] E. Cuthill & J. McKee. *Reducing the bandwidth of sparse symmetric matrices.* In Proceedings of the 1969 24th national conference, pages 157–172, New York, NY, USA, 1969. ACM Press.

[Cuthill 72] E. Cuthill. Several strategies for reducing the bandwidth of matrices. Plenum Press, New York, 1972.

[Dagum 98] Leonardo Dagum & Ramesh Menon. *OpenMP: An Industry-Standard API for Shared-Memory Programming.* IEEE Computational Science and Engineering, vol. 05, no. 1, pages 46–55, 1998.

[Davis 94] Timothy A. Davis. *University of Florida Sparse Matrix Collection.* NA Digest, vol. 92, 1994.

[Davis 97] Timothy A. Davis & Iain S. Duff. *An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization.* SIAM J. Matrix Anal. Appl., vol. 18, no. 1, pages 140–158, 1997.

[Davis 04] Timothy A. Davis. *Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method.* ACM Trans. Math. Softw., vol. 30, no. 2, pages 196–199, 2004.

[Davis 05a] Timothy A. Davis. *Algorithm 849: A concise sparse Cholesky factorization package.* ACM Trans. Math. Softw., vol. 31, no. 4, pages 587–591, 2005.

[Davis 05b] Timothy A. Davis. *SuiteSparse: Collection of sparse software*, 2005.

[Davis 06] Timothy A. Davis. Direct methods for sparse linear systems (fundamentals of algorithms 2). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[Davis 09] Timothy A. Davis & Yifan Hu. *University of florida sparse matrix collection*, 2009.

[Davis 11a] Timothy A. Davis & Yifan Hu. *The University of Florida Sparse Matrix Collection.* ACM Transactions on Mathematical Software, vol. 38, no. 1, 2011.

[Davis 11b] Timothy A. Davis & Yifan Hu. *The University of Florida Sparse Matrix Collection.* Rapport technique, ACM Transactions on Mathematical Software, January 2011.

[Delcaro 74] L. G. Delcaro & G. L. Sicuranza. *A Method for Transposing Externally Stored Matrices.* IEEE Trans. Comput., vol. 23, no. 9, pages 967–970, September 1974.

[Demmel 01] James Demmel, Katherine Yelick*et al. BeBOP: The Berkey Benchmarking and OPtimisation Group*, 2001.

[Demmel 05] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R.C. Whaley & K. Yelick. *Self-Adapting Linear Algebra Algorithms and Software.* Proceedings of the IEEE, vol. 93, no. 2, pages 293–312, Feb. 2005.

[Dobrian 04] Florin Dobrian & Alex Pothen. *Oblio: Design and Performance.* In Jack Dongarra, Kaj Madsen & Jerzy Wasniewski, editeurs, PARA, volume 3732 of *Lecture Notes in Computer Science*, pages 758–767. Springer, 2004.

[Dongarra 88] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling & Richard J. Hanson. *An extended set of FORTRAN basic linear algebra subprograms*. ACM Trans. Math. Softw., vol. 14, no. 1, pages 1–17, 1988.

[Dongarra 90] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling & I. S. Duff. *A set of level 3 basic linear algebra subprograms*. ACM Trans. Math. Softw., vol. 16, no. 1, pages 1–17, 1990.

[Dow 95] Murray Dow. *Transposing a matrix on a vector computer*. Parallel Computing, vol. 21, no. 12, pages 1997 – 2005, 1995.

[Drepper 07] Ulrich Drepper. What every programmer should know about memory. Red Hat, Inc., 2007.

[Duff 86] Iain S Duff, Albert M Erisman & John K Reid. Direct methods for sparse matrices. Oxford University Press, Inc., New York, NY, USA, 1986.

[Duijvestijn 72] A.J.W. Duijvestijn. *Correctness proof of an in-place permutation*. BIT Numerical Mathematics, vol. 12, no. 3, pages 318–324, 1972.

[Durstenfeld 64] Richard Durstenfeld. *Algorithm 235: Random permutation*. Commun. ACM, vol. 7, no. 7, pages 420–, July 1964.

[Eaton 09] John W. Eaton, David Bateman & Soren Hauberg. GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations. CreateSpace Independent Publishing Platform, 2009. ISBN 1441413006.

[Eklundh 72] J. O. Eklundh. *A Fast Computer Method for Matrix Transposing*. IEEE Trans. Comput., vol. 21, no. 7, pages 801–803, July 1972.

[Eklundh 73] J. O. Eklundh. *Author's Reply*. IEEE Trans. Comput., vol. 22, no. 5, pages 543–544, May 1973.

[El-Hadedy 10] M. El-Hadedy, S. Purohit, M. Margala & S.J. Knapskog. *Low latency transpose memory for high throughput signal processing*. In NEWCAS Conference (NEWCAS), 2010 8th IEEE International, pages 373–376, 2010.

[El-Moursy 08] Ali El-Moursy, Ahmed El-Mahdy & Hisham El-Shishiny. *An Efficient In-place 3D Transpose for Multicore Processors with Software Managed Memory Hierarchy*. In Proceedings of the 1st International Forum on Next-generation Multicore/Manycore Technologies, IFMT '08, pages 10:1–10:6, New York, NY, USA, 2008. ACM.

[Elmroth 04] Erik Elmroth, Fred Gustavson, Isak Jonsson & Bo Kågström. *Recursive blocked algorithms and hybrid data structures for dense matrix library software.* SIAM Review, vol. 46, pages 3–45, 2004.

[Ercal 10] Burin Ercal & William Stein. *The Sage Project: Unifying Free Mathematical Software to Create a Viable Alternative to Magma, Maple, Mathematica and MATLAB.* In Komei Fukuda, Joris-vander Hoeven, Michael Joswig & Nobuki Takayama, editeurs, Mathematical Software ICMS 2010, volume 6327 of *Lecture Notes in Computer Science*, pages 12–27. Springer Berlin Heidelberg, 2010.

[Feijen 87] W. H. J. Feijen, A. J. M. Van Gasteren & David Gries. *In-situ inversion of a cyclic permutation.* Inf. Process. Lett., vol. 24, no. 1, pages 11–14, January 1987.

[Fernandes 02] Edil S. Tavares Fernandes, Valmir C. Barbosa & Fabiano Ramos. *Instruction Usage and the Memory Gap Problem.* In SBAC-PAD, pages 169–175. IEEE Computer Society, 2002.

[Fich 95] Faith E. Fich, J. Ian Munro & Patricio V. Poblete. *Permuting In Place.* SIAM J. Comput., vol. 24, no. 2, pages 266–278, April 1995.

[Finkel 74] R.A. Finkel & J.L. Bentley. *Quad trees a data structure for retrieval on composite keys.* Acta Informatica, vol. 4, no. 1, pages 1–9, 1974.

[Floyd 72] Robert W. Floyd. *Permuting Information in Idealized Two-Level Storage.* In Raymond E. Miller & James W. Thatcher, editeurs, Complexity of Computer Computations, The IBM Research Symposia Series, pages 105–109. Plenum Press, New York, 1972.

[Fraser 76] Donald Fraser. *Array Permutation by Index-Digit Permutation.* J. ACM, vol. 23, no. 2, pages 298–309, April 1976.

[Frigo 98] Matteo Frigo & Steven G. Johnson. *FFTW: An adaptive software architecture for the FFT.* In Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[Frigo 99] Matteo Frigo, Charles E. Leiserson, Harald Prokop & Sridhar Ramachandran. *Cache-Oblivious Algorithms.* In Proceedings of the 40th Annual Symposium on Foundations of Computer Science, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[Frigo 05] M. Frigo & S.G. Johnson. *The Design and Implementation of FFTW3.* Proceedings of the IEEE, vol. 93, no. 2, pages 216–231, 2005.

[Gatlin 99] Kang Su Gatlin & L. Carter. *Memory hierarchy considerations for fast transpose and bit-reversals.* In High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On, pages 33–42. IEEE, 1999.

[George 81] Alan George & Joseph W. Liu. Computer solution of large sparse positive definite systems. Prentice Hall Professional Technical Reference, 1981.

[Goedecker 01] Stefan Goedecker & Adolfy Hoisie. Performance optimization of numerically intensive codes. Software, environments, tools. Philadelphia, Pa. Society for Industrial and Applied Mathematics, 2001.

[Goldbogen 81] G. C. Goldbogen. *PRIM: A Fast Matrix Transpose Method.* IEEE Trans. Softw. Eng., vol. 7, no. 2, pages 255–257, March 1981.

[Golub 96] Gene H. Golub & Charles F. Van Loan. Matrix computations (3rd ed.). Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[Gonzalez-Mesa 13] Miguel A. Gonzalez-Mesa, Eladio D. Gutierrez & Oscar Plata. *Parallelizing the Sparse Matrix Transposition: Reducing the Programmer Effort Using Transactional Memory.* Procedia Computer Science, vol. 18, no. 0, pages 501–510, 2013. 2013 International Conference on Computational Science.

[Goto 02] K. Goto & R. van de Geijn. *On reducing TLB misses in matrix multiplication.* Rapport technique, Univ. of Texas at Austin, 2002.

[Goto 08a] Kazushige Goto & Robert Van De Geijn. *High-performance implementation of the level-3 BLAS.* ACM Transactions on Mathematical Software, vol. 35, no. 1, pages 1–14, 2008.

[Goto 08b] Kazushige Goto & Robert A. van de Geijn. *Anatomy of high-performance matrix multiplication.* ACM Transactions on Mathematical Software, vol. 34, no. 3, pages 1–25, 2008.

[Gould 04] Nicholas I. M. Gould & Jennifer A. Scott. *A numerical evaluation of HSL packages for the direct solution of large sparse, symmetric linear systems of equations.* ACM Trans. Math. Softw., vol. 30, no. 3, pages 300–325, 2004.

[Gould 05] N.I.M. Gould, Y. Hu & J. A. Scott. *A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations.* Rapport technique, Rutherford Appleton Laboratory, April 2005.

[Greene 81] D.H. Greene & D.E. Knuth. Mathematics for the analysis of algorithms. Progress in computer science. Birkhäuser, 1981.

[Group 63] Numerical Analysis Group. *A collection of Fortran codes for large scale scientific computation. http://www.hsl.rl.ac.uk*, 1963.

[Gupta 97] Anshul Gupta, George Karypis & Vipin Kumar. *Highly scalable parallel algorithms for sparse matrix factorization.* Parallel and Distributed Systems, IEEE Transactions on, vol. 8, no. 5, pages 502–520, May 1997.

[Gupta 01] Anshul Gupta & Mahesh Joshi. *WSMP: A High-Performance Shared- and Distributed-Memory Parallel Sparse Linear Equation Solver*, 2001.

[Gustavson 78a] Fred G. Gustavson. *Remark on Algorithm 408: A Sparse Matrix Package (Part 1) [F4].* ACM Trans. Math. Softw., vol. 4, no. 3, pages 295–, September 1978.

[Gustavson 78b] Fred G. Gustavson. *Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition.* ACM Trans. Math. Softw., vol. 4, no. 3, pages 250–269, September 1978.

[Gustavson 98] F. Gustavson, A. Henriksson, I. Jonsson & B. Kaagstrom. Recursive blocked data formats and blas's for dense linear algebra algorithms., volume 1541 of *LNCS*, pages 195–206. Springer, 1998.

[Gustavson 12] FredG. Gustavson. *Cache Blocking.* In Kristjn Jnasson, editeur, Applied Parallel and Scientific Computing, volume 7133 of *Lecture Notes in Computer Science*, pages 22–32. Springer Berlin Heidelberg, 2012.

[Harlow 57] Francis H. Harlow. *Hydrodynamic Problems Involving Large Fluid Distortions.* J. ACM, vol. 4, no. 2, pages 137–142, April 1957.

[He 02] Yun He & Chris H. Q. Ding. *MPI and OpenMP Paradigms on Cluster of SMP Architectures: The Vacancy Tracking Algorithm for Multi-dimensional Array Transposition.* In Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, Supercomputing '02, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[Hegland 96] Markus Hegland. *Real and complex fast Fourier transforms on the Fujitsu VPP 500.* Parallel Computing, vol. 22, no. 4, pages 539 – 553, 1996.

[Hendrickson 98] Bruce Hendrickson & Tamara G. Kolda. *Partitioning Sparse Rectangular Matrices for Parallel Computations of Ax and ATv.* In Proceedings of the 4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems, PARA '98, pages 239–247, London, UK, UK, 1998. Springer-Verlag.

[Herlihy 93] Maurice Herlihy & J. Eliot B. Moss. *Transactional Memory: Architectural Support for Lock-free Data Structures.* In Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[Herrero 03] José R. Herrero & Juan J. Navarro. *Improving Performance of Hypermatrix Cholesky Factorization.* In Euro-Par 2003, volume 2790 of *LNCS*, pages 461–469. Springer, Aug 2003.

[Hestenes 52] M. R. Hestenes & E. Stiefel. *Methods of conjugate gradients for solving linear systems.* Journal of research of the National Bureau of Standards, vol. 49, pages 409–436, 1952.

[Higham 02] Nicholas J. Higham. Accuracy and stability of numerical algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[Hoare 61] C. A. R. Hoare. *Algorithm 64: Quicksort.* Commun. ACM, vol. 4, no. 7, pages 321–, July 1961.

[Hoare 62] C. A. R. Hoare. *Quicksort.* The Computer Journal, vol. 5, no. 1, pages 10–16, January 1962.

[Houscholder 52] A. S. Houscholder. *Errors in iterative solutions of linear systems.* In ACM '52: Proceedings of the 1952 ACM national meeting (Toronto), pages 30–33, New York, NY, USA, 1952. ACM.

[Huffman 52] David Huffman. *A Method for the Construction of Minimum-Redundancy Codes.* Proceedings of the IRE, vol. 40, no. 9, pages 1098–1101, September 1952.

[IBM 70] IBM. *Engineering and Scientific Subroutine Library (ESSL)*, 1970.

[IBM 76] International Business Machines Corporation IBM & F.G. Gustavson. Permuting matrices stored in sparse format. IBM, 1976.

[Im 04] Eun-Jin Im, Katherine A. Yelick & Richard Vuduc. *SPARSITY: Framework for Optimizing Sparse Matrix-Vector Multiply.* International Journal of High Performance Computing Applications, vol. 18, no. 1, pages 135–158, February 2004.

[Intel 93] Intel. *Intel MKL: Intel Math Kernel Library*, 1993.

[Jie 10] Yuan Jie, Wu Jian-ping & Wang Zheng-hua. *Parallel transposing and communication strategies for FFT on cluster of SMP architectures with multicore processors.* In Image and Signal Processing (CISP), 2010 3rd International Congress on, volume 7, pages 3280–3283, 2010.

[Johnson 92] R. Johnson, C.H. Huang & R. W. Johnson. *Tensor permutations and block matrix allocation.* In G. gains & L. Mullin, editeurs, Second International Workshop on Array Structures (ATABLE). Dept. of Information and Operation Research, 1992.

[Johnson 93] R. Johnson. *A tensor product formulation of matrix transposition.* Appl. Math Letters, 1993.

[Kågström 06] Bo Kågström. *Management of deep memory hierarchies: recursive blocked algorithms and hybrid data structures for dense matrix computations.* In Proceedings of the 7th international conference on Applied Parallel Computing: state of the Art in Scientific Computing, PARA'04, pages 21–32, Berlin, Heidelberg, 2006. Springer-Verlag.

[Kaligosi 06] Kanela Kaligosi & Peter Sanders. *How branch mispredictions affect quicksort.* In Proceedings of the 14th conference on Annual European Symposium - Volume 14, ESA'06, pages 780–791, London, UK, UK, 2006. Springer-Verlag.

[Karypis 98] George Karypis & Vipin Kumar. *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.* SIAM J. Sci. Comput., vol. 20, no. 1, pages 359–392, 1998.

[Kaushik 93] S. D. Kaushik, C.-H. Huang, R. W. Johnson, P. Sadayappan & J. R. Johnson. *Efficient transposition algorithms for large matrices.* In Proceedings of the 1993 ACM/IEEE conference on Supercomputing, Supercomputing '93, pages 656–665, New York, NY, USA, 1993. ACM.

[Keller 02] Jorg Keller. *A heuristic to accelerate in-situ permutation algorithms.* Inf. Process. Lett., vol. 81, no. 3, pages 119–125, February 2002.

[Kernighan 88] Brian W. Kernighan & Dennis M. Ritchie. The c programming language second edition. Prentice-Hall, Inc., 1988.

[Knott 75] G. D. Knott. *Hashing functions*. The Computer Journal, vol. 18, no. 3, pages 265–278, 1975.

[Knuth 71] Donald E. Knuth. *Mathematical Analysis of Algorithms*. In IFIP Congress (1), pages 19–27, 1971.

[Knuth 76] Donald E. Knuth. *Big Omicron and big Omega and big Theta*. SIGACT News, vol. 8, no. 2, pages 18–24, 1976.

[Knuth 98] Donald E. Knuth. The art of computer programming. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, second edition, 1998.

[Krishnamoorthy 04] Sriram Krishnamoorthy, Gerald Baumgartner, Daniel Cociorva, Chi-Chung Lam & P. Sadayappan. *Efficient parallel out-of-core matrix transposition*. Int. J. High Perform. Comput. Netw., vol. 2, no. 2-4, pages 110–119, February 2004.

[Kruskal 89] Clyde P. Kruskal, Larry Rudolph & Marc Snir. *Techniques for parallel manipulation of sparse matrices*. Theoretical Computer Science, vol. 64, no. 2, pages 135 – 157, 1989.

[Kundert 86] K. S. Kundert. *Sparse Matrix Techniques and Their Applications to Circuit Simulation*. In A. E. Ruehli, editeur, Circuit Analysis, Simulation and Design. New York: North-Holland, 1986.

[Laflin 70a] Susan Laflin & M. A. Brebner. *Algorithm 380: in-situ transposition of a rectangular matrix [F1]*. Commun. ACM, vol. 13, pages 324–326, May 1970.

[Laflin 70b] Susan Laflin & M. A. Brebner. *Algorithm 380: in-situ transposition of a rectangular matrix [F1]*. Commun. ACM, vol. 13, no. 5, pages 324–326, May 1970.

[Lam 91] Monica D. Lam, Edward E. Rothberg & Michael E. Wolf. *The Cache Performance and Optimizations of Blocked Algorithms*. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IV, pages 63–74, New York, NY, USA, 1991. ACM.

[Landau 24] E. Landau. *ber die Anzahl der Gitterpunkte in gewissen Bereichen*. Nachrichten von der Gesellschaft der Wissenschaften zu Gttingen, Mathematisch-Physikalische Klasse, vol. 1924, pages 137–150, 1924.

[Larimore 98] S. Larimore. *An approximate minimum degree column ordering algorithm.* Rapport technique TR-98-016, CISE, University of Florida, 1998. MS Thesis.

[Lawson 79] C. L. Lawson, R. J. Hanson, D. R. Kincaid & F. T. Krogh. *Basic Linear Algebra Subprograms for Fortran Usage.* ACM Trans. Math. Softw., vol. 5, no. 3, pages 308–323, 1979.

[Leathers 79] Burton L. Leathers. *Remark on Algorithm 513: Analysis of In-Situ Transposition [F1] and Remark on Algorithm 467: Matrix Transposition in Place [F1].* ACM Trans. Math. Softw., vol. 5, no. 4, pages 520–, December 1979.

[Lee 08] Seyong Lee & Rudolf Eigenmann. *Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems.* In ICS '08: Proceedings of the 22nd annual international conference on Supercomputing, pages 195–204, New York, NY, USA, 2008. ACM.

[Lewars 03] E.G. Lewars. Computational chemistry: Introduction to the theory and applications of molecular and quantum mechanics. Springer, 2003.

[Lewis 81] H.R. Lewis & C.H. Papimitriou. Elements of the theory of computation. Prentice-Hall Software Series. Pearson Education Canada, 1981.

[Li 03] Xiaoye S. Li & James W. Demmel. *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems.* ACM Trans. Math. Softw., vol. 29, no. 2, pages 110–140, 2003.

[Li 05] Xiaoye S. Li. *An overview of SuperLU: Algorithms, implementation, and user interface.* ACM Trans. Math. Softw., vol. 31, no. 3, pages 302–325, 2005.

[Lippert 98] Th. Lippert, K. Schilling, F. Toschi, S. Trentmann & R. Tripiccione. *Transpose algorithm for FFT on APE/Quadrics.* In Peter Sloot, Marian Bubak & Bob Hertzberger, editeurs, High-Performance Computing and Networking, volume 1401 of *Lecture Notes in Computer Science*, pages 439–448. Springer Berlin Heidelberg, 1998.

[Lorin 75] Harold Lorin. Sorting and sort systems (the systems programming series). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1975.

[Manegold] Stefan Manegold. *The calibrator: a cache-memory and TLB calibration tool.*

[Martone 10a] M. Martone, S. Filippone, S. Tucci & M. Paprzycki. *Assembling recursively stored sparse matrices.* In Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on, pages 317–325, 2010.

[Martone 10b] Michele Martone, Salvatore Filippone, Salvatore Tucci, Marcin Paprzycki & Maria Ganzha. *Utilizing Recursive Storage in Sparse Matrix-Vector Multiplication - Preliminary Considerations.* In Thomas Philips, editeur, CATA, pages 300–305. ISCA, 2010.

[Martone 11] Michele Martone, Marcin Paprzycki & Salvatore Filippone. *An Improved Sparse Matrix-Vector Multiply Based on Recursive Sparse Blocks Layout.* In Ivan Lirkov, Svetozar Margenov & Jerzy Wasniewski, editeurs, LSSC, volume 7116 of *Lecture Notes in Computer Science*, pages 606–613. Springer, 2011.

[MATLAB 10] MATLAB. version 7.10.0 (r2010a). The MathWorks Inc., Natick, Massachusetts, 2010.

[McKee 04] Sally A. McKee. *Reflections on the Memory Wall.* In Proceedings of the 1st Conference on Computing Frontiers, CF '04, pages 162–, New York, NY, USA, 2004. ACM.

[McNamee 71] John Michael McNamee. *Algorithm 408: A Sparse Matrix Package (Part I) [F4].* Commun. ACM, vol. 14, no. 4, pages 265–273, April 1971.

[Melville 79] Robert C. Melville. *A Time-Space Tradeoff for In-Place Array Permutation.* Rapport technique, Cornell University, Ithaca, NY, USA, 1979.

[Monagan 05] Michael B. Monagan, Keith O. Geddes, K. Michael Heal, George Labahn, Stefan M. Vorkoetter, James McCarron & Paul DeMarco. Maple 10 programming guide. Maplesoft, Waterloo ON, Canada, 2005.

[Moore 65] Gordon E. Moore. *Cramming more components onto integrated circuits.* Electronics, vol. 38, no. 8, April 1965.

[Morton 66] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. International Business Machines Company, 1966.

[NAG 93] The Numerical Algorithms Group NAG. *NAG Fortran Library Mark 15*, 1993.

[Nagel 73] Laurence W. Nagel & D.O. Pederson. *SPICE (Simulation Program with Integrated Circuit Emphasis)*. Rapport technique UCB/ERL M382, EECS Department, University of California, Berkeley, Apr 1973.

[Na'mneh 06] Rami Al Na'mneh, W. David Pan & Seong-Moo Yoo. *Efficient Adaptive Algorithms for Transposing Small and Large Matrices on Symmetric Multiprocessors*. Informatica, vol. 17, no. 4, pages 535–550, December 2006.

[Navarro 96] Juan J. Navarro, E. García & José R. Herrero. *Data Prefetching and Multilevel Blocking for Linear Algebra Operations*. In Proceedings of the 10th international conference on Supercomputing, pages 109–116. ACM Press, May 1996.

[Nethercote 03] Nicholas Nethercote & Julian Seward. *Valgrind: A Program Supervision Framework*. Electronic Notes in Theoretical Computer Science, vol. 89, pages 44–66, 2003.

[Nethercote 07] Nicholas Nethercote & Julian Seward. *Valgrind: a framework for heavyweight dynamic binary instrumentation*. SIGPLAN Not., vol. 42, no. 6, pages 89–100, 2007.

[Nishtala 04] Rajesh Nishtala, Richard Vuduc, James Demmel & Katherine Yelick. *When cache blocking sparse matrix vector multiply works and why*. In Proceedings of the PARA'04 Workshop on the State-of-the-art in Scientific Computing, Copenhagen, Denmark, June 2004.

[Padgett 09] W.T. Padgett & D.V. Anderson. Fixed-point signal processing. Synthesis lectures on signal processing. Morgan & Claypool, 2009.

[Pall 60] Gordon Pall & Esther Seiden. *A problem in abelian groups, with application to the transposition of a matrix on an electronic computer*. Mathematics of Computation, vol. 14, pages 189–192, 1960.

[Pettersson 05] M. Pettersson. *Perfctr: Linux Performance Monitoring Counters Driver*. Rapport technique, Uppsala University, 2005.

[Pissanetzky 84] S. Pissanetzky. Sparse matrix technology. Academic Press, 1984.

[Portnoff 99] M. R. Portnoff. *An efficient parallel-processing method for transposing large matrices in place*. IEEE Transactions on Image Processing, vol. 8, no. 9, pages 1265–1275, September 1999.

[Ramapriyan 75] H. K. Ramapriyan. *A Generalization of Eklundh's Algorithm for Transposing Large Matrices.* IEEE Trans. Comput., vol. 24, no. 12, pages 1221–1226, December 1975.

[Ravankar 11] A.A. Ravankar & S.G. Sedukhin. *An O(n) Time-Complexity Matrix Transpose on Torus Array Processor.* In Networking and Computing (ICNC), 2011 Second International Conference on, pages 242–247, 2011.

[Remington 96] Karin Remington & Roldan Pozo. *NIST Sparse BLAS User's Guide.* Rapport technique, Internal Report NISTIR 6744, National Institute of Standards and Technology, 1996.

[Rogers 03] D.W. Rogers. Computational chemistry using the pc. Wiley, 2003.

[Saad 94] Youcef Saad. *SPARSKIT: a basic tool kit for sparse matrix computations - Version 2*, 1994.

[Saad 03] Y. Saad. Iterative methods for sparse linear systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[Sanders 08] Beverly A. Sanders, Erik Deumens, Victor Lotrich & Mark Ponton. *Refactoring a Language for Parallel Computational Chemistry.* In Proceedings of the 2Nd Workshop on Refactoring Tools, WRT '08, pages 11:1–11:4, New York, NY, USA, 2008. ACM.

[Schaller 97] Robert R. Schaller. *Moore's law: past, present, and future.* IEEE Spectr., vol. 34, no. 6, pages 52–59, June 1997.

[Schenk 01] O. Schenk & K. Gartner. *Sparse Factorization with Two-Level Scheduling in PARADISO*, 2001.

[Schrijver 86] Alexander Schrijver. Theory of linear and integer programming. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[Schumann 72] U. Schumann. *Ein Verfahren zum Transponieren grosser, sequentiell gespeicherter Matrizen.* Angew. Inform., pages 213–216, 1972.

[Schumann 73] U. Schumann. *Comments on "A Fast Computer Method for Matrix Transposing" and Application to the Solution of Poisson's Equation.* IEEE Trans. Comput., vol. 22, no. 5, pages 542–543, May 1973.

[Sedgewick 11] R. Sedgewick & K. Wayne. Algorithms. Pearson Education, 2011.

[Sedgewick 13] R. Sedgewick & P. Flajolet. An introduction to the analysis of algorithms. Pearson Education, 2013.

[Sen 02] Sandeep Sen, Siddhartha Chatterjee & Neeraj Dumir. *Towards a theory of cache-efficient algorithms*. J. ACM, vol. 49, no. 6, pages 828–858, November 2002.

[Shutler 08] Paul M. E. Shutler, Seok Woon Sim & Wei Yin Selina Lim. *Analysis of Linear Time Sorting Algorithms*. Comput. J., vol. 51, no. 4, pages 451–469, July 2008.

[Simecek 09] I. Simecek. *Memory Hierarchy Behavior Study during the Execution of Recursive Linear Algebra Library*. Acta Polytechnica, vol. 49, no. 5/2008, pages 29–36, 2009.

[Sinha 04] R. Sinha. *Using Compact Tries for Cache-Efficient Sorting of Integers*. In C. C. Ribeiro, editeur, Proceedings of the Third International Workshop on Efficient and Experimental Algorithms (WEA 2004), pages 513–528, Angra dos Reis, Rio de Janeiro, Brazil, may 2004. LNCS 3059.

[Sipala 77] Paolo Sipala. *Remark on "Algorithm 408: A Sparse Matrix Package (Part I) [F4]"*. ACM Trans. Math. Softw., vol. 3, no. 3, pages 303–, September 1977.

[Sipser 96] Michael Sipser. Introduction to the theory of computation. International Thomson Publishing, 1st edition, 1996.

[Snir 95] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra & Steven Huss-Lederman. Mpi: The complete reference. MIT Press, Cambridge, MA, USA, 1995.

[Snir 98] Marc Snir & Steve Otto. Mpi-the complete reference: The mpi core. MIT Press, Cambridge, MA, USA, 1998.

[Stathis 03a] P. Stathis, S. Vassiliadis & S. Cotofana. *A Hierarchical sparse matrix storage format for vector processors*. In Parallel and Distributed Processing Symposium, 2003. Proceedings. International, pages 8 pp.–, 2003.

[Stathis 03b] Pyrrhos Stathis, Stamatis Vassiliadis & Sorin Cotofana. *D-SAB: A Sparse Matrix Benchmark Suite*. In VictorE. Malyshkin, editeur, Parallel Computing Technologies, volume 2763 of *Lecture Notes in Computer Science*, pages 549–554. Springer Berlin Heidelberg, 2003.

[Stathis 04] P. Stathis, D. Cheresiz, S. Vassiliadis & B. Juurlink. *Sparse matrix transpose unit.* In Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, pages 90–, 2004.

[Stewart 01] G. W. Stewart. Matrix algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[Stoer 02] Josef Stoer & Roland Bulirsch. Introduction to Numerical Analysis. Springer Verlag, 2002.

[Strassen 69] Volker Strassen. *Gaussian elimination is not optimal.* Numerische Mathematik, vol. 13, no. 4, pages 354–356, August 1969.

[Straubhaar 08] Julien Straubhaar. *Parallel preconditioners for the conjugate gradient algorithm using Gram-Schmidt and least squares methods.* Parallel Computing, 2008. In Press, Accepted Manuscript, Available online 19 June 2008.

[Suh 02] J. Suh & V. K. Prasanna. *An Efficient Algorithm for Out-of-Core Matrix Transposition.* IEEE Trans. Comput., vol. 51, no. 4, pages 420–438, April 2002.

[Szab'o 91] B. Szab'o & I. Babuska. Finite element analysis. John Wiley & Sons, 1991.

[Toledo 03] Sivan Toledo. *Taucs: A library of sparse linear solvers*, 2003.

[Tsifakis 04] Dimitrios Tsifakis, AlistairP. Rendell & PeterE. Strazdins. *Cache Oblivious Matrix Transposition: Simulation and Experiment.* In Marian Bubak, GeertDick Albada, PeterM.A. Sloot & Jack Dongarra, editeurs, Computational Science - ICCS 2004, volume 3037 of *Lecture Notes in Computer Science*, pages 17–25. Springer Berlin Heidelberg, 2004.

[Twigg 83] D. W. Twigg. *Transposition of Matrix Stored on Sequential File.* IEEE Trans. Comput., vol. 32, no. 12, pages 1185–1188, December 1983.

[Twogood 76] R.E. Twogood & M.P. Ekstrom. *An Extension of Eklundh's Matrix Transposition Algorithm and Its Application in Digital Image Processing.* Computers, IEEE Transactions on, vol. C-25, no. 9, pages 950–952, 1976.

[Uht 97] Augustus K. Uht, Vijay Sindagi & Sajee Somanathan. *Branch Effect Reduction Techniques.* Computer, vol. 30, no. 5, pages 71–81, May 1997.

[Valsalam 02] Vinod Valsalam & Anthony Skjellum. *A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels.* Concurrency and Computation: Practice and Experience, vol. 14, no. 10, pages 805–839, 2002.

[Van Voorhis 77] D. C. Van Voorhis. *Comments on "A Computer Algorithm for Transposing Nonsquare Matrices".* IEEE Trans. Comput., vol. 26, no. 6, pages 607–608, June 1977.

[Vuduc 05] Richard Vuduc, James W. Demmel & Katherine A. Yelick. *OSKI: A library of automatically tuned sparse matrix kernels.* In Proceedings of SciDAC 2005, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[Wapperom 06] P. Wapperom, A. N. Beris & M. A. Straka. *A New Transpose Split Method for Three-dimensional FFTs: Performance on an Origin2000 and Alphaserver Cluster.* Parallel Comput., vol. 32, no. 1, pages 1–13, January 2006.

[Welch 84] T.A. Welch. *A Technique for High-Performance Data Compression.* Computer, vol. 17, no. 6, pages 8–19, 1984.

[Whaley 97] R. Clint Whaley & Jack Dongarra. *Automatically Tuned Linear Algebra Software.* Rapport technique UT-CS-97-366, University of Tennessee, December 1997.

[Whaley 98] R. Clint Whaley & Jack Dongarra. *Automatically Tuned Linear Algebra Software.* In SuperComputing 1998: High Performance Networking and Computing, 1998. CD-ROM Proceedings. **Winner, best paper in the systems category**.

[Whaley 01] R. Clint Whaley, Antoine Petitet & Jack J. Dongarra. *Automated Empirical Optimization of Software and the ATLAS Project.* Parallel Computing, vol. 27, no. 1–2, pages 3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000.

[Wilkes 01] Maurice V. Wilkes. *The memory gap and the future of high performance memories.* SIGARCH Computer Architecture News, vol. 29, no. 1, pages 2–7, 2001.

[Willard 84] Dan E. Willard. *New trie data structures which support very fast search operations.* J. Comput. Syst. Sci., vol. 28, no. 3, pages 379–394, 1984.

[Williams 90] Thomas Williams & Colin Kelley. *GNUPLOT - An Interactive Plotting Program*, 1990.

[Windley 59] P. F. Windley. *Transposing Matrices in a Digital Computer*. j-COMP-J, vol. 2, no. 1, pages 47–48, apr 1959.

[Wise 01] David S. Wise, Jeremy D. Frens, Yuhong Gu & Gregory A. Alexander. *Language support for Morton-order matrices*. In Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming, PPoPP '01, pages 24–33, New York, NY, USA, 2001. ACM.

[Wolfram 03] Stephen Wolfram. The mathematica book. Wolfram Media/Cambridge University Press, fifth edition edition, 2003.

[Wulf 95] Wm. A. Wulf & Sally A. McKee. *Hitting the Memory Wall: Implications of the Obvious*. SIGARCH Comput. Archit. News, vol. 23, no. 1, pages 20–24, March 1995.

[Yotov 05] Kamen Yotov, Keshav Pingali & Paul Stodghill. *Automatic Measurement of Memory Hierarchy Parameters*. In Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '05, pages 181–192, New York, NY, USA, 2005. ACM.

[Yotov 07] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels & Fred Gustavson. *An experimental comparison of cache-oblivious and cache-conscious programs*. In Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, SPAA '07, pages 93–104, New York, NY, USA, 2007. ACM.

[Yzelman 11] A.N. Yzelman & Rob H. Bisseling. *Two-dimensional cache-oblivious sparse matrix-vector multiplication*. Parallel Computing, vol. 37, no. 12, pages 806 – 819, 2011. ¡ce:title¿6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA10)¡/ce:title¿.

[Ziv 77] J. Ziv & A. Lempel. *A universal algorithm for sequential data compression*. Information Theory, IEEE Transactions on, vol. 23, no. 3, pages 337–343, 1977.

Glossary

Algorithm	A step by step procedure for performing a calculation or carrying out a task.
BLAS	Basic Linear Algebra Subprogrammes - A software library of routines for performing standard Linear Algebra operations.
Cache	The cache is part of the Central Processing Unit (CPU) which is used to reduce the average time to access data from the main memory. It is a smaller, faster memory which stores copies of data from recently used main memory locations.
Cache Oblivious	An algorithm designed to take advantage of CPU caches without needing to know the size of the cache.
COO	Compressed Coordinate matrix storage format (Section 2.3.4).
CSR	Compressed Sparse Row matrix storage format (Section 2.3.5).
CSC	Compressed Sparse Column matrix storage format (Section 2.3.5).
Cycle	A permutation of (a subset) of the elements - The complete chain of elements that are rearranged one after the other.
Cycle-Chasing	The processes during in-place transpose of moving elements from location to location one after another in the matrix, permuting the elements.
Dense	A matrix where all values are stored in memory - There are very few zero values in a dense matrix.

Diagonal	The line of elements from top left of the matrix to bottom right. Diagonal elements have the same row and column index, $i = j$.
DRAM	Dynamic Random-Access Memory - Main memory.
Element	The individual items/values/entries in a matrix.
Fill-In	In high order routines, when a row or column of a sparse matrix is added to another - zero elements may become non-zero - space must be made in the compact structure for these new elements.
In-Place	When an algorithm alters data while keeping it in the original array/data-structure.
Jump	During the cycle-chasing transpose, the algorithm moves an element from one location in the matrix arrays to it's new row in a different location in the matrix arrays. The algorithm will then need to move the existing element at that new location to yet another location, and so on until the cycle completes. We use the term "Jump" to describe how the algorithm moves from location to location during the cycle-chasing.
LAPACK	Linear Algebra PACKage - A software library of high order Linear Algebra routines, similar to BLAS.
Linear Algebra	The branch of mathematics that deals with the theory of systems of linear equations, matrices, vector spaces, determinants, and linear transformations.
Matrix	A rectangular array of numbers arranged in rows and columns - A mathematical representation of Linear Algebra Equations.
MIMD	Multiple Instruction Multiple Data - A type of parallel architecture, multiple processors can carry out different instructions on different data at the same time.

Page Table	The mapping between virtual addresses in a program and physical addresses on the machine.
PAPI	Performance Application Programming Interface - A library for monitoring hardware counters used to measure cache/TLB/etc. performance.
Permutation	Rearrange elements in a particular pattern (permute).
Out-of-Place	When an algorithm alters data by copying it to a completely new array/data-structure of the same size.
SIMD	Single instruction Multiple Data - A type of parallel architecture where multiple processors carry out the same instruction on different data values at the same time.
Sparse	A matrix with a high proportion of zero elements stored in a condensed format. Many matrices have a high proportion of elements which have a value of zero, often 99% of the elements or more. A sparse matrix is a matrix with a high proportion of zero elements which is stored in a compact format (such as CSC/CSR) in memory in order to avoid explicitly storing the zero values. This compact format reduces the memory required to store the matrix and can reduce the number of arithmetic operations that need to be performed during Linear Algebra algorithms.
TLB	Translation Lookaside Buffer - A Cache of the Page Table.
Transpose	A linear algebra operation that reflects a matrix through its main top left to bottom right diagonal - swapping rows with columns - element $A_{i,j}$ is swapped with $A_{j,i}$.
Triangular Solve	Solving the unknowns in the vector x in an equation such as $Ax = b$ where A is lower (L) or upper (U) triangular - An much simpler process than when A is a full matrix.