Heterogeneous Multiconstraint Application Partitioner (HMAP)

Servesh Muralidharan†, Aravind Vasudevan†, Avinash Malik§ and David Gregg†
†*Lero and School of Computer Science and Statistics, Trinity College Dublin, Dublin 2, Ireland*
Email: muralis@scss.tcd.ie, vasudeva@scss.tcd.ie, David.Gregg.cs.tcd.ie
§*IBM Research - Ireland*
Email: avinmali@ie.ibm.com

***Abstract*—In this article we propose a novel framework – *Heterogeneous Multiconstraint Application Partitioner* (HMAP) for exploiting parallelism on heterogeneous *High performance computing* (HPC) architectures. Given a heterogeneous HPC cluster with varying compute units, communication constraints and topology, HMAP framework can be utilized for partitioning applications exhibiting task and data parallelism resulting in increased performance. The challenge lies in the fact that heterogeneous compute clusters consist of processing elements exhibiting different compute speeds, vector lengths, and communication bandwidths, which all need to be considered when partitioning the application and associated data. We tackle this problem using a staged graph partitioning approach. Experimental evaluation on a variety of different heterogeneous HPC clusters and applications show that our framework can exploit parallelism resulting in more than $3\times$ speedup over current state of the art partitioning technique. HMAP framework finishes within seconds even for architectures with 100's of processing elements, which makes our algorithm suitable for exploring parallelism potential.**

***Keywords*-Graph partitioning, vectorization, data parallelism, heterogeneous architectures, clusters.**

I. Introduction

High performance computing (HPC) clusters increasingly consist of large numbers of heterogeneous processing elements such as CPUs, graphics processing units (GPUs), field programmable gate arrays (FPGAs), low-power processors intended for digital signal processing (DSP), etc. By combining heterogeneous processing units it may be possible to divide the work so that different types of computation in the application are run on different types of units. This can result in significant speed-ups, lower hardware costs and/or reduced power consumption by the HPC system. For example, if a computation contains the right patterns of data parallelism it may run dozens or even hundreds of times faster on a GPU than on a CPU that has similar cost and power consumption. On the other hand, computations with less data parallelism and more complex control flow may run faster on CPUs. Matching the type of computation to the processor can yield significant benefits. Although the potential of heterogeneous computing is great, exploiting that potential is more difficult.

In this paper we consider the streaming [1] model of computation. Streaming is a popular model for programs such as image and signal processing, financial applications, networking, telecommunications, etc. In the streaming model statements (also called filters/actors/tasks or kernels) execute iteratively, processing the incoming tokens of data. Given such a stream application, it is difficult to partition the available parallelism onto the hardware. For example, how does one decide which parallel filters should run on which type of execution unit? Given a system with dozens or hundreds of CPUs, GPUs and other units, how does one divide the work between them? There are several conflicting factors. For example, one wants to allocate filters to the type of execution unit that will execute it most efficiently. On the other hand, one wants to achieve a good load balance by dividing the work evenly across the units.We want to allocate the filters to reduce communication costs while at the same time taking account of all the other factors.

We consider the problem of partitioning stream graphs onto heterogeneous HPC computing systems. This problem has been studied extensively for homogeneous architectures where all processing elements are the same. Although the homogeneous case is NP-hard [2], several heuristic solutions have been found that work well in practice. However, extending these solutions to the heterogeneous case is difficult for two reasons.

In the heterogeneous case some processing elements are more powerful than others, so achieving a good load balance usually involves distributing the work unevenly.

A second reason why it can be difficult to extend algorithms for homogeneous architectures to the heterogeneous hardware relates to the strengths and weaknesses of different types of processors. When considering heterogeneous architectures, it is tempting to think of some processing elements simply being more powerful than others. A GPU is not simply a more powerful CPU. In fact, some types of computation run better on CPUs and some on GPUs. For a partitioning algorithm to work well, it needs to take account of the strengths and weaknesses of different types of processing elements. In this paper we present an approach to partitioning parallel tasks to heterogeneous architectures that addresses both of these concerns.

Our **main contributions** are as follows:

- We present a novel approach to characterizing the type of processing elements based on their level of vector parallelism which, allows us to distinguish the

suitability of different types of units to different filters.
- We provide a novel algorithm for partitioning task and data parallelism to heterogeneous architectures based on hierarchical graph partitioning.

The rest of this paper is organized as follows. Section II formalizes the problem statement and defines the objective function. Next, in Section III, we provide a detailed description of our framework. Section IV gives the quantitative comparisons of our approach against other approaches. Section V describes the related work and positions our approach in comparison to these works. Finally, we conclude in Section VI.

II. Preliminaries

We now present a formal description of the problem along with the notations used.

A. Execution model

Consider the Jacobi example and its filter graph in Figure 1. The Jacobi algorithm is used in fluid dynamics and heat transfer problems. We consider every statement (marked 1 to 4) in this example to be a filter that can be run in a software pipelined [3] manner on a given architecture. An *example* execution trace of the Jacobi example is shown in Table I for some arbitrary value of computation and communication latency of statements.

P0	1_0		1_1		1_2		
P1	2_0	3_0	2_1	3_1	2_2		
P2	$1'_0$		$1'_1$		$1'_2$		
P3			4_0	4_0	4_1	4_1	

Table I: Example execution trace of the Jacobi kernel

In a software pipelined model, the different iterations of the filters are run in parallel, e.g., 1_0 is the 1^{st} iteration of statement **1** in the Jacobi example, while 1_1 is the second iteration and so on and so forth. The **period** of the application is the time period where all filters of the stream graph run simultaneously, shown within the double lined columns in Table I. In such a model, the resource allocation (rather than dependencies) determines the application period, especially without back-edges in the filter graph (as is the case with our model). In Table I, the resource allocation on processing element `P1` and `P3` determines the application period, because it is the maximum of the four allocation latencies.

B. Task and Data parallelism

1) Task parallelism: Task parallel filters are the branches connected to a split node. These filters can be run in parallel provided enough resources are available. For example, in Figure 1(b) statements **1** and **2** are task parallel filters connected to the split node **start**. The execution trace shown in Table I exploits this task parallelism by excuting statements **1** and **2** on processors `P0` and `P1` concurrently. This type of concurrent execution of different task parallel filters is ususally termed as *Multiple Instruction Multiple Data* (MIMD) parallelism.

2) Data Parallelism: Data parallelism is the ability to exploit the parallelism hidden in stateless filters. As shown in Figure 1(b), statement **1** is a stateless filter. Two copies of this stateless filter are run concurrently on processors `P0` and `P1` thereby reducing the overall period as shown in Table I. The best allocation for a stateless filter is a vector processor with the ability to execute large number of concurrent copies – ususally termed *Single Instruction Mutiple Data* (SIMD) parallelism.

C. Notations

We refer to our application graph, as a *Stream Graph* defined formally as a weighted directed graph: $G_t(V_t, E_t)$, where V_t is the set of all filters in the stream graph and E_t represents the communication buffers between these filters. The system resources are represented by a weighted undirected graph $G_r(V_r, E_r)$ where V_r represents a set of processing elements (PEs), which can have different processing capabilities and E_r represents the communication links between these PEs with differing communication bandwidths.

D. Problem Definition

Given a graph $G_t(V_t, E_t)$, each vertex in the filter graph, $t_i \in V_t$ has a set of associated requirements represented by T_j^i where $j = 0...n_{t-1}$ with n_{t-1} being the number of requirements. These requirements represent the computational requirements of the filter. Namely, T_0^i represents the scalar requirements, while T_1^i represents the vector requirements.

The communication edges are represented with $e^c \in E_t$, which denotes the data(in bytes), the filter requires for processing. Each resource node $r_i \in V_r$ has a number of computational capabilities, represented by R_j^i where $j = 0...n_{r-1}$. For each resource node, capability: R_0^i represents the frequency of the PE or how many scalar instructions the PE can perform in one second (the *Million Instructions Per Second* (MIPS) count). Capability : R_1^i denotes the maximum number of parallel vector operations it can perform (the vector length). Each edge $e \in E_r$ has a weight which represents the bandwidth between two PEs r_i and r_j which is denoted by E^c.

The problem at hand is to effectively partition the stream graph G_t onto given resource graph G_r. This problem is known to be NP-Hard [2].

$$((T_1^i/R_1^j \times T_0^i)/R_0^j) + (e^c/E^{c'}) | c = (t_i, t_k), t_k \neq t_i, \forall t_k \in V_t, c' = (r_j, r_l), r_l \neq r_j, \forall r_l \in V_r \tag{1}$$

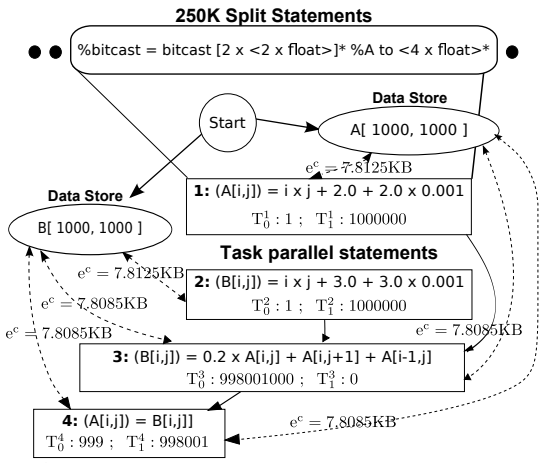
$$\begin{gathered} L_s^{\mathcal{P}} = Lcomp_s^{\mathcal{P}} + Lcomm_s^{\mathcal{P}} \\ Lcomp_s^{\mathcal{P}} = \textstyle\sum_{\forall t_i \in V_t} ((T_1^i/R_1^s \times T_0^i)/R_0^s) \\ Lcomm_s^{\mathcal{P}} = \textstyle\sum_{\forall t_i \in V_t} e^c/E^{c'} \\ s.t., d = (t_i, t_k), t_k \neq t_i, \forall t_k \in V_t \wedge\ c' = (r_s, r_l), r_l \neq r_s, \\ \forall r_l \in V_r \wedge\ \text{d is routed on c}' \end{gathered} \tag{2}$$

$$\mathbb{P} = max(L_s^{\mathcal{P}}), \forall r_s \in V_r \tag{3}$$

```
//Task and data-parallel
for (int i=0;i<999; ++i){
 for (int j=0;j<999; ++j){
  1: A[i][j] = (i*j+2.0+2.0/1000)
  2: B[i][j] = (i*j+3.0+3.0/1000)
 }
}
for (int k=0;k<1000;++k){
 for (int i=1; i<998; ++i)
  for (int j=1; j<998; ++j)
   3: B[i][j] = 0.2*(A[i][j]+A[i][j-1]
              +A[i][j+1]+A[i-1][j])

 //Data-parallel
 for (int i=1; i<999; ++i)
  for (int j=1; j<999; ++j)
   4: A[i][j] = B[i][j]
}
```

(a) Example 2-dimensional Jacobi application



(b) The filter graph for the Jacobi example [1]

Figure 1: Jacobi example and its filter-graph

Given some filter $t_i \in V_t$ partitioned onto some resource $r_j \in V_r$, the computation latency for that node is computed by Equation (1). In this formulation for some filter t_i being partitioned onto some resource r_j, we first calculate the number of vectorized instructions that can be executed in parallel (by dividing the required vector length by the vector capacity of r_j represented by R_1^j). We then multiply this by the number of iterations in the loop to get the total number of instructions to be performed by that filter-graph node. For example, for statement **4** in Figure 1(b), $T_0^4 = 999$ and $T_1^4 = 998001$, respectively. This in-turn indicates that statement **4** requires 998001 way vector parallelism (also termed data-parallelism) and each of these vector instructions are carried out 999 times. If we assume a processor (most likely a GPU) is able to provide 998001 way vector-parallelism, then the result of Equation (1) is: $\frac{998001}{998001} \times 999$.

Once we have this number, we calculate latency of execution of this filter-graph node t_i on this resource r_j by dividing it with the MIPS value of the resource denoted by R_0^j. Calculation of the communication latency requires dividing the number of bits to be transferred by the bandwidth of the shortest path.

Given the filter-graph and the resource-graph, let $\mathcal{P}$ be some partition of the application on the resource-graph. For a particular allocation onto some resource node $r_s \in V_r$, we define its computation and communication latency as in Equation (2). Finally, the complete application period is defined in Equation (3) and is the map of all latencies calculated using Equation (3) as shown in Table I.

The objective of our framework is to find a partition $\mathcal{P}$ that minimizes the total application period as in Equation (3).

III. HMAP FRAMEWORK

A common approach to solving the homogeneous case is to *partition* the graph across the processing elements [4]–[6]. However, we have found that such heuristic partitioning approaches do not work well for heterogeneous architectures. Instead, we propose a novel approach where the architecture is hierarchically partitioned into sub-clusters. Our heuristic algorithm does this in such a way that the sub-cluster at the same level of the architecture hierarchy has approximately the same computing capability and has relatively local communication. This allows us to use existing approaches that work well for the homogeneous case to partition *heterogeneous* architectures across the sub-clusters at each level of the hierarchy. We show that this is an effective approach if the sub-clusters are well balanced at each level of the hierarchy.

Our heuristic HMAP consists of two important concepts that need description. First, we describe how the topology clusters are formed from the resource graphs accounting for communication and heterogeneity of the topology. Secondly, given a stream graph with data parallel filters, task parallel filters, and communication extracted as shown in Figure 1(b) how the partitioning is performed.

A. Clustering the resource graph

The resource graph represents the cluster of compute nodes on which the filter graph will be executed. A sample resource graph is shown in Figure 2 at level 0. The resource graph that is shown is heterogeneous in both computation and communication. The properties of the resource graph are described below:

- **Compute nodes:** The compute nodes (PEs) are assumed to belong under two categories of processing units, mainly CPUs and GPUs. Following the current trend, the CPUs have a larger MIPS count. The MIPS

[1] Ellipses represent data stores. Rectangles represent filter nodes. Rounded rectangle represents data parallel nodes. The dots represent other data parallel nodes not shown in the figure. Dashed arrows represent communication between data stores and execution statements. Solid arrows represent dependence edges. Task parallel statements: 1, 2 and data parallel statements: 1, 2, and 4 are marked for convenience.

capacity of a PE is denoted by the first constraint $R_0^i, \forall i \in V_r$. The GPU nodes have a lower MIPS count, but have a large vector length, denoted by the second constraint $R_1^i, \forall i \in V_r$. For example, the fifth PE (`E0`, Figure 2) at level 0 is a CPU, since it has a small vector count and a large MIPS count, the first PE (`A0`) on the other hand is a GPU, since the capabilities are reversed.

- **Communication links:** The resource graph shown in Figure 2 at level 0, follows a 2D mesh topology. In this topology the PEs are connected in a grid with individual communication links between them. The bandwidth of these links is non uniform. The different bandwidths on the communication links is represented by the constraint E^C. Our framework can handle any kind of topology, the 2D mesh shown in Figure 2, is just an example topology.

1) Clustering the topology: The main idea behind our partitioning approach is to first hierarchically cluster the nodes in the heterogeneous architecture provided by the designer, thereby forming clusters. The application is then partitioned in stages (levels) onto the resulting hierarchy. The intuition behind this approach is two fold:

- *Heterogeneous K-way partitioning:* The process of partitioning an application onto a given architecture is equivalent to a heterogeneous K-way partitioning problem. Hierarchically clustering a heterogeneous topology such that the resulting hierarchy consists of clustered PEs with balanced compute capabilities can reduce the heterogeneous K-way partitioning problem to a homogeneous one.
- *Considering communication links:* The communication can be considered into the equation, while building the hierarchical cluster using the min-cut technique. Thus, a min-cut load-balancing of the PEs in a topology intuitively means: we are clustering together PEs, which have large bandwidth together into a single cluster, while making an attempt to load balance the two capabilities: MIPS and vector lengths.

The hierarchical cluster built for the synthetic topology at level 0 of Figure 2, is shown in the levels 1-3. The stages used to build the hierarchy are as follows:

- *Effective computation during clustering:* Given a topology graph with $|V_r|$ resources, we cluster the PEs in levels, whereby the height of the cluster is $log_2|V_r|$. For example, consider the PEs at level 0 in Figure 2. Clustering from level 0 to level 1 results in 4 PEs at level 1, where the two capabilities, R_0^k, R_1^k for each cluster $k = \{i, j\}, \exists i \in V_r \wedge \exists j \in V_r$ is computed as: $R_0^k = R_0^i + R_0^j$, and $R_1^k = R_1^k + R_1^j$. Without loss of generality we assume this for any $k = \{i, j, .., n\}, \forall n \in V_r$. This process is continued until we reach the top-level with just 1 cluster. We end up with a load-balanced hierarchy, with each level showing a larger amount of homogeneity, and a smaller number of clustered PEs. The reason for a level based clustering, instead of clustering all nodes into a single 2 node partition, is that when partitioning the application on the resulting top-down cluster, we have fine grained details within each of the cluster.
- *Effective communication during clustering:* When clustering nodes, the effective communication between two such clusters is hard to determine. This is because the clustering itself is a virtual representation of the actual nodes. Moreover, there might be multiple unique paths between two nodes across different clusters. Choosing a suitable path is a routing problem and its beyond the scope of this paper. In this paper we take a pessimistic approach to calculating the effective communication bandwidth between clustered nodes, because this gives us the lower bound on communication. We assume that in the worst case scenario the link(between any two nodes) with the least bandwidth would act as the bottleneck. These unique links are determined for all the nodes in the cluster. Then to get the *effective bandwidth* between these clusters we aggregate these bandwidths. The algorithm to calculate the effective communication between clusters is outlined below:

  1) We use the all-pair Floyd-Warshall [7] algorithm to calculate the shortest path, in terms of communication latency of data-transfer, for every communication link in the topology. From this we calculate the bandwidth of these links by inverting its communication latency. This creates a list, which contains all of the best case bandwidths between any two nodes.
  2) For any two clusters, we choose a source node in one cluster and determine all the paths to all the nodes in the connecting cluster. From this we choose the link with the least bandwidth.
  3) Next, we repeat the above step for the other source nodes and end up with the paths, which would act as a bottleneck for any given unique source and destination pair.
  4) Addition of these resultant bandwidths is the effective bandwidth between the two clusters.

The purpose of calculating this effective bandwidth is that during clustering of the nodes at each level we would like to balance the nodes not only based on their compute capabilities but also their communication potential. This would give us more balanced clusters with both computation and communication taken into account. For the topology, at level 0, in Figure 2, we first calculate the best possible paths between every pair of nodes $(i, j) \in (V_r \times V_r)$. Now let us consider the example of the clustered nodes `B1` and `C1`. Node `B1` is a cluster of the set of children nodes $\{C0, H0, F0\}$ and `C1` is the cluster of the node set

$\{B0, A0\}$, respectively. Being an undirected graph with loss of generality we can consider `B1` to be the source and `C1` to be the destination. Hence, the worst path, in terms of communication latency of data-transfer, following the all-pair Floyd-Warshall algorithm, between nodes $C0$ and $B0$ within the clustered nodes is given by the maximum communication latency path amongst the memoized best edges: $max((C0, B0),\ (H0, B0),\ (F0, B0))$. Similar computation is carried out for all link pairs in the clustered node with $A0$ as the destination node to find the minimum communication latency path. Finally, the reciprocal of these two latencies and its addition gives us the effective bandwidth between the two clusters. This effective bandwidth represents the communication link between the two clustered nodes.

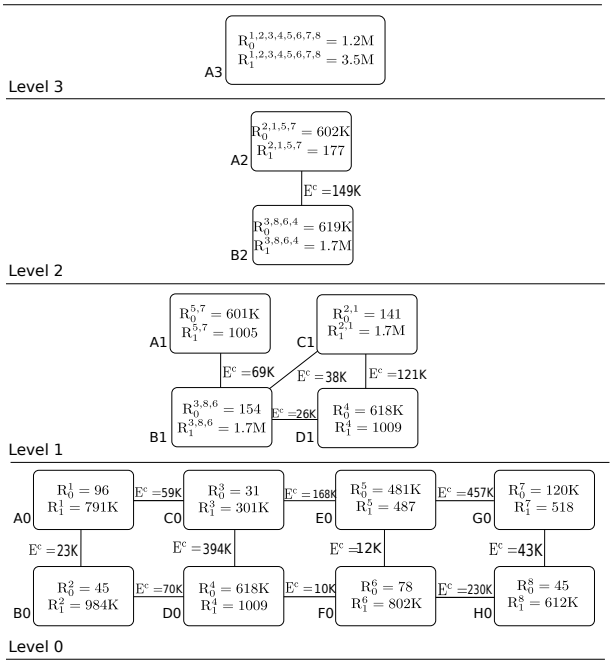


Figure 2: Clustering of a resource graph

B. Filter graph partitioning

The filter partitioning on the resulting hierarchical cluster takes a top-down approach. We start with the filter graph extracted from the application (Figure 1). We then recursively partition the filter graph (using K-way partitioning) on the hierarchical cluster of the resource-graph. For the example cluster in Figure 2, we start by partitioning the filter graph in Figure 1, first into two (K=2) partitions considering two equally weighted compute nodes at level 2. Once a partition is obtained for this level, we move onto the next level (level 1), whereby all the filter graph nodes allocated onto node `A2` are further partitioned onto the nodes `A1` and `C1` (again K=2), which are coupled into the cluster `A2`. This process is continued recursively for all clusters until the final level (level 0). During each partition we make sure that the filter node requirements are closely matched to the resource node capabilities.

Doing it in this top down manner has two important consequences.

- *Increase in the time complexity:* As stated previously, the partitioning problem is equivalent to the K-way partitioning problem. The multi-level K-way partitioning results in the worst cast time complexity of $O(|E_t| \times log_2|V_r|)$ [8]. Our algorithm gives a worst case complexity of $log_2|V_r| \times O(|E_t| \times log_2|V_r|)$ when using the multi-level K-way partitioning. But, in the average case we ask for a balanced 2-way partition at all levels, which results in a complexity of $log_2|V_r| \times O(E_t)$ in the average case.
- *Refined partitioning:* By dividing the partitioning on to several levels we can achieve much better load balancing by considering only fewer nodes to partition than if we were to do it directly on to the resource graph.

IV. Experiments and results

A. Our implementation

We use the Metis [9] graph partitioning library in the implementation of our partitioning algorithm. We are not tied to Metis and any other graph partitioner such as Zoltan [10] or Scotch [11] can be used for implementing our algorithm.

We now describe how we used Metis to implement the graph partitioning. The resource graph is represented in the Metis graph format. We represent the PEs' capabilities as constraints of the nodes and the links' bandwidths as communication weight on the edges. We then construct our clustered structure (Figure 2) by asking for a 2-way partition at each level of the $log_2|V_r|$ height. Metis partitions the graph by load balancing the constraints and performing a minimum edge cut. In partitioning the filter graph, we need to distribute the constraints on to the available partitions such that capability and requirement is balanced. Metis offers the ability to load balance multiple constraints on to different partitions based on the metric 'tp-weight'. This metric basically represents the ratio between a given type of constraint across the different partitions. We calculate the ratios between the capabilities of different partitions and represent them as this metric in order to load balance on to the available partitions.

B. The experimental set-up

The experimental set-up consists of the resource graph generation and the filter graph generation. Herein, we describe the two set-ups.

1) The resource graph set-up: The experimental set up consists of the following.

1) An interconnection network with $|V_r|$ nodes. $|V_r|$ varies from 64 to 4096 PEs. A node can be just a multi-core CPU or a multi-core CPU with an attached GPU.

2) A set of N_G GPUs where N_G is at most $|V_r|$. The GPUs are connected in the network at locations, chosen randomly in the normal distribution of 25% to 75% of $|V_r|$.
3) A set $\boldsymbol{G} = \{G_1, G_2, G_3, ...G_{|\boldsymbol{G}|}\}$ Every GPU in this experiment has a vector length of G_i where G_i is sampled randomly from the set $\boldsymbol{G}$. The elements of set $\boldsymbol{G}$ are chosen from a normal distribution ranging from: 10000 to 100000.
4) A set $\boldsymbol{C} = \{C_1, C_2, C_3, ...C_{|\boldsymbol{C}|}\}$ Every CPU in this experiment has C_i cores where C_i is sampled randomly from the set $\boldsymbol{C}$.
5) A set $\boldsymbol{M} = \{M_1, M_2, M_3, ...M_{|\boldsymbol{M}|}\}$ Every $C_i \in \boldsymbol{C}$ and GPU in this experiment has a MIPS count of M_i where M_i is sampled randomly from the set $\boldsymbol{M}$. The elements of set $\boldsymbol{M}$ are chosen from a normal distribution ranging from: 1000 to 100000.
6) A set $\boldsymbol{B} = \{B_1, B_2, B_3, ...B_{|B|}\}$ Every $|E_r|$ edge in this experiment has a bandwidth of B_i in MB/s where B_i is sampled randomly from a normal distribution ranging from: 100 to 100000

For given values of $|V_r|$, N_G, $\boldsymbol{G}$, $\boldsymbol{C}$, $\boldsymbol{M}$ and $\boldsymbol{B}$ and a given application, let the k-th trial be defined as one execution of the following sequence of steps.

- For each GPU G_i, sample $\boldsymbol{G}$ and $\boldsymbol{M}$ randomly to determine its vector length V_i and MIPS count M_i.
- For each CPU P_i, sample $\boldsymbol{C}$ randomly to determine the number of cores C_i in the processor P_i.
- For each core C_i in the processor P_i sample V_i and M_i randomly from set $\boldsymbol{G}$ and $\boldsymbol{M}$.
- Use our framework to extract data and filter parallelism that is best utilizable by the heterogeneity created by parameters in items 1, 2, and 3 above. Determine the execution time $\mathbb{P}$.

An experiment, $\boldsymbol{E}$ ($|V_r|$, N_G, $\boldsymbol{G}$, $\boldsymbol{C}$ $\boldsymbol{M}$, $\boldsymbol{B}$), consists of conducting enough of the above trials so that width of the 95% confidence interval on the average value of $\mathbb{P}$ is less than 10% of the average value. This results in a variable number of trials with different experimental set-ups. Note that two trials differ from each other only in the seed for the random number generator. This reduces the dependence of our results on a lucky sequence of numbers from the random number generator.

2) Random application graph generation: We built a random graph generator to test our partitioning methodology rigorously. The random graph generator takes as input the following parameters:

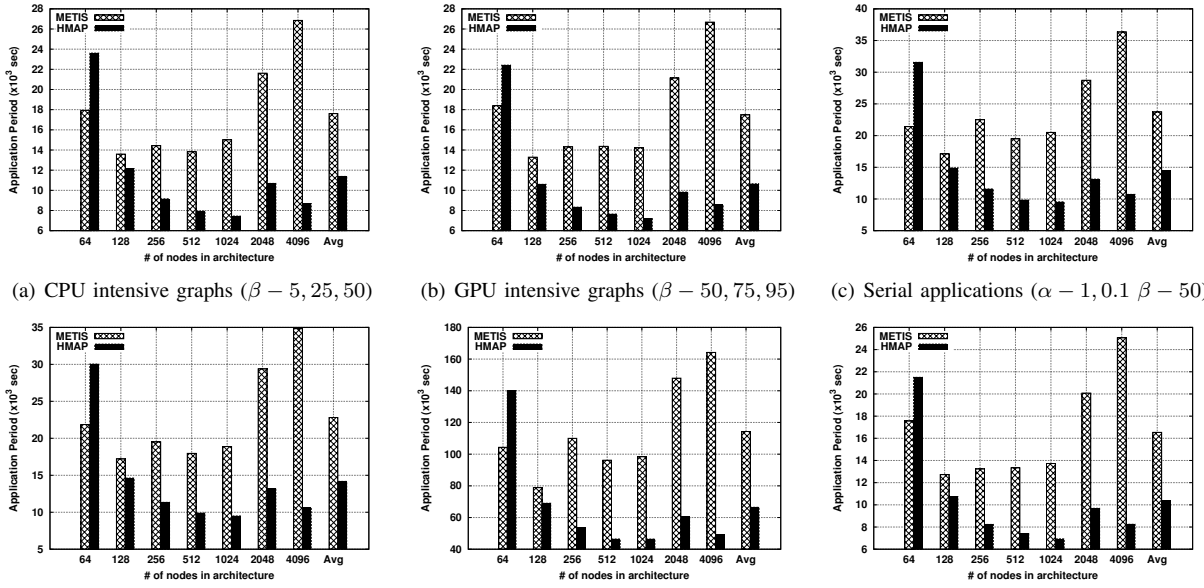
- Number of nodes (n) - Total number of nodes to be present in the filter graph
- Indegree (i) - Average indegree of every vertex
- Outdegree (o) - Average outdegree of every vertex
- Communication to Computation Ratio - CCR (c) - It is the ratio of the average communication cost of an out-edge the average computation cost of the vertex itself. If a application graph's c is low, then it can be called a computation intensive application and if it is greater than 1 it can be called a communication intensive application
- Structure of the graph (α) - We generate the height of the graph based on α as $\frac{\sqrt{n}}{\alpha}$. This implies the width of the graph becomes $\sqrt{n} \times \alpha$. Higher values of α give wider graphs, which means the graph has more inherent task parallelism, while lower values give taller graphs, which means the graph is inherently serial
- Beta (β) - We use this parameter to decide if an actor in the filter graph is CPU intensive or GPU intensive. Smaller values of β makes actors CPU intensive (by making first constraint larger than the second), while larger values make it GPU intensive (by making the second constraint larger than the first). This essentially means that smaller values of β creates filters, which require lot more non-vectorized units, whereas larger values would result in filters with larger vector requirement.
- Skewedness factor (γ) - This parameter dictates how computation is spread across the graph. Smaller values of γ give uniformly distributed values for the constraints of the actors while larger values produces skewed graphs

For our experiments, random graphs are generated by choosing values for the input parameters from the following sets:

- $\chi_n = \{128, 256, 512, 1024, 4096, 8192, 16384\}$
- $\chi_o = \{2, 4, 8\}$
- $\chi_c = \{0.0001, 0.001, 0.01, 0.1, 1\}$
- $\chi_\alpha = \{0.1, 1.0, 10.0\}$
- $\chi_\beta = \{5, 25, 50, 75, 95\}$
- $\chi_\gamma = \{5, 25, 50, 75, 95\}$

We generated one graph per combination for a total of 7875 application graphs. Since the random graph generator has a variety of inputs and these inputs are filled in from a large set of possible values, a diverse set of application graphs are generated with various characteristics.

We have varied the structure of the random application graphs from more CPU intensive to more GPU (vector) intensive, more task-parallel to sequential and a combination of these (see Section IV-B1) and hence, tried to encapsulate all different possibilities. Also instead of choosing a simple application such as Jacobi example as shown in Figure 1, experiments based on such diverse set of application graphs prevents biasing towards a particular partitioning algorithm. This allows us to evaluate how our HMAP heuristic performs for different category of application graphs. The real workload characterization for a certain domain of applications (e.g., scientific computing) is out of the scope of this paper and remains as future research.



(a) CPU intensive graphs ($\beta - 5, 25, 50$) (b) GPU intensive graphs ($\beta - 50, 75, 95$) (c) Serial applications ($\alpha - 1, 0.1\ \beta - 50$)

(d) Task parallel applications ($\alpha - 10, 1\ \beta - 50$) (e) Comm. intensive ($\alpha - 1\ \beta - 5\ c - 1$) (f) Overall

Figure 3: Experimental Results
Each graph shows the application period $\mathbb{P}$ for a given class of application, on architecture consisting of various $|V_r|$ nodes

C. Experimental Results

The experimental set-up consists of a dual socket system consisting of Intel Xeon E5620 CPU running at 2.4Ghz with 24GB DDR3 RAM. The system is running Linux kernel ver 3.0.40-1. The HMAP heuristic was compiled using GCC version 4.7.1 with '-O3' optimization flag.

We ran over 100,000 experiments, using the random application graphs and our HMAP heuristic and K-way partitioning for seven resource architectures. Overall it took us 74 sec on average for the biggest architecture of 4096 nodes to determine a partition of the application graph onto the given heterogeneous architecture, which is 50 sec more than Metis. The results comparing K-way partitioning using Metis [9] and our HMAP framework are shown in Figure 3. Once the partitioning of the application graph for the given architecture is determined, we calculate the application period using Equation (3).

The results are divided into five graphs, each representing a common class (based on section IV-B2) of application graphs. The final bar in each graph represents the average application period for that class of application over the different architectures.

Figures 3(a) and 3(b) represent application graphs that consists of proportionally larger non-vectorized and vectorized requirements, respectively. The application graphs with values of $\beta = 5,\ 25,\ \text{and}\ 50$ are more non-vectorized and require a higher MIPS count. Whereas the application graphs with $\beta = 50,\ 75,\ \text{and}\ 95$ are those that require a higher vector count. On resource architecture of size 64 nodes, Metis results in an application partition with the period being $1.2\times$ faster than the one obtained by our scheme. But, for the rest of the architectures we outperform the partition obtained by standard Metis. For example, in the case of 4096 node architecture, the resultant application period from our HMAP scheme is $3\times$ faster compared to the one obtained via Metis.

In the case of Figures 3(c) and 3(d), which represent more serial and task-parallel application graphs, respectively we notice a similar trend. Metis outperforms the HMAP scheme for smaller architectures but HMAP outperforms the partition obtained via Metis in all other cases.

We attribute such behavior to the following; for smaller architectures, the variation (in terms of MIPS and vector lengths) in the underlying physical architecture can be easily captured in standard Metis using its so called constraints 'tpweights'. For larger architectures with a very large number of nodes as in our HPC case, these variations in the underlying architecture cannot be expressed in Metis in the form of constraints. We alleviate this inexpressibility by clustering the large number of heterogeneous nodes into a smaller number of homogeneous clusters as shown in Figure 2.

Figure 3(e) shows the result of partitioning communication intensive application graphs on the same set of heterogeneous HPC physical architectures. We perform significantly better in comparison to Metis for larger archtiectures. This is due to our clustering approach that ensures topology nodes that communicate with high bandwidths are combined together at each level. Moreover, Metis is unaware of the communication between nodes in the topology as it only performs a min-cut when partitioning the application graphs. Our framework on the other hand, takes into account the communication in the topology and indirectly matches heavy

edges from the filter graph onto high bandwidth edges in the topology graph.

Figure 3(f) shows the average application period based on all the input graphs. HMAP performs better than standard metis on all architectures bigger than 64 nodes with an average speedup of more than $1.5\times$ and for the largest resource architecture of 4096 nodes, HMAP performs $3\times$ better than standard metis.

Finally, HMAP does not perform as well on physical topologies wherein the clustering of the resource graph does not result in clusters with equal number of nodes. For example, consider the 2048 node cluster, which is a 64×32 2D mesh. In such cases, clustering the given topology into virtual clusters (see Figure 2) results in clusters without equal number of nodes. Improving performance, further still, in such topologies remains an open question, which we plan to deal with in the future.

Moreover it is well known that data transfers between RAM and GPU memory are expensive and we would like to extend our heuristic to take into account the additional latency encountered during partitioning the application. In the future, we would also like to build a system capable of generating and executing the application based on the partition provided by our heuristic on a given architecture.

V. Related work

A significant amount of existing research aims to extract parallelism from programs [12]–[14]. The polyhedral optimization model [12] concentrates on automatically extracting data parallelism from loops operating on arrays. The polyhedral optimization community has addressed parallelization for CPUs and GPUs separately, but to our knowledge has not explored the combination of the two.

Carpenter et. al [15] provide a heuristic algorithm for partitioning stream programs onto heterogeneous architectures. However, the heterogeneity of processors is represented purely by their clock speed. No distinction is made between processors with differing amounts of vector parallelism.

The StreamIt [16] community also address the problem of scheduling and partitioning stream programs onto homogeneous parallel hardware, such as the RAW [17] architecture. StreamIt can exploit data parallelism, by replicating stateless filters, but it does not exploit vector parallelism.

Classical algorithms such as critical path scheduling [18] and list scheduling [19] are used for scheduling task parallelism onto homogeneous architectures. The list scheduling techniques targeting heterogeneous architectures such as [20] do not exploit vector SIMD parallelism. Declustering [14], is another technique for partitioning tasks to parallel hardware, which again does not consider vector parallelism.

Cluster based partitioning techniques [21] consider only independent tasks without communication. The proposed heuristics for partitioning data-parallel applications onto clusters [22], [23] do not consider vectorization potential available on the compute clusters and only concentrate on partitioning task parallel processes.

Heuristic optimization techniques such as genetic algorithms (GA) and Simulated annealing [24], [25] and local search methods [26] use a semi-random search of the space of possible partitions filters on to execution resources. The effectiveness of these techniques often depends on choosing good values for parameters to the algorithm. Determining good values for these parameters is a difficult problem that often requires trial and error.

Malik et al. 2012 [27], similar to our technique partitions stream graphs onto heterogeneous architectures consisting of CPUs and GPUs based on a mathematical integer linear programming formulation (ILP) to provide optimal solutions but, does not scale well to large architectures or stream graphs. Hence, ILP is more suitable for small embedded systems, whereas our work is more suited for HPC systems. Sui et al. 2010 [28] focus on extracting amorphous parallelism, which occur in irregular algorithms working on graph data-structures similar to Metis [9]. We on the other hand target streaming applications, and extract data and task-parallelism, which are commonly occurring forms of parallelism on SIMD and MIMD architectures. Thus, our work is orthogonal to that of Sui et al. Same can be stated about Catalyurek et al. 2001 [29].

VI. Conclusion

In this paper we have described a novel staged graph based partitioning heuristic to partition and schedule stream graphs onto heterogeneous execution architectures called HMAP. Our HMAP heuristic is able to exploit both multiple instruction multiple data and single instruction multiple data parallelism from stream graphs by allocating the task and data parallel actors to appropriate computation units – primarily CPUs get allocated the task parallel compute intensive filters, while GPUs get allocated the data parallel filters. HMAP is able to detect the strengths and weaknesses of the varying compute units in the given architecture and perform (if required) an uneven load balance to achieve maximum throughput. Moreover, our HMAP framework also deals with the varying bandwidth in the underlying topology.

We tested HMAP on a statistically significant 100,000 different experiments, by differing the features in the stream graph and the underlying architecture, with success. HMAP outperforms the current state of the art Metis partitioner by providing a stream graph partition that outperforms Metis by around $1.5\times$ on average and by around $3\times$ for large architectures. Finally, partitioning results even for large architectures and stream graphs are obtained within seconds, thereby making our heuristic suitable fo both off-line and on-line partitioning.

VII. Acknowledgment

This work is partly funded by the IRCSET Enterprise Partnership Scheme in collaboration with IBM Research, Ireland.

References

[1] J. Buck and E. Lee, *The Token Flow Model*. Advanced Topics in Dataflow Computing and Multithreading, Wiley IEEE Computer Society, 1995.

[2] V. Sarkar, "Partitioning and scheduling parallel algorithms for execution on multiprocessors," Ph.D. dissertation, Stanford university, 1989.

[3] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software Pipelined Execution of Stream Programs on GPUs," in *CGO*, 2009, pp. 200–209.

[4] A. Aletà, J. M. Codina, J. Sánchez, and A. González, "Graph-partitioning based instruction scheduling for clustered processors," in *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 34. Washington, DC, USA: IEEE Computer Society, 2001, pp. 150–159.

[5] K. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *Computers, IEEE Transactions on*, vol. 48, no. 6, pp. 579 –590, Jun. 1999.

[6] E. Nystrom and A. E. Eichenberger, "Effective cluster assignment for modulo scheduling," in *Proceedings of the 31st annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 31. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 103–114.

[7] S. S. Skiena, *The Algorithms Design Manual*, 2nd ed. Springer-Verlag London, 2008.

[8] G. Karypis, V. Kumar, and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, pp. 96–129, 1998.

[9] G. Karypis and V. Kumar, "Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0," Tech. Rep., 1995.

[10] K. Devine, E. Boman, L. Riesen, U. Catalyurek, and C. Chevalier, "Getting started with zoltan: A short tutorial," in *Proc. of 2009 Dagstuhl Seminar on Combinatorial Scientific Computing*, 2009, also available as Sandia National Labs Tech Report SAND2009-0578C.

[11] C. Chevalier and F. Pellegrini, "PT-Scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, no. 6-8, pp. 318–331, Jul. 2008.

[12] M. Griebl, C. Lengauer, and S. Wetzel, "Code Generation in the Polytope Model," in *In IEEE PACT*. IEEE Computer Society Press, 1998, pp. 106–111.

[13] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGOPS Oper. Syst. Rev.*, vol. 40, pp. 151–162, October 2006.

[14] G. C. Sih and E. A. Lee, "Declustering: A new multiprocessor scheduling technique," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 625–637, June 1993.

[15] P. M. Carpenter, A. Ramirez, and E. Ayguade, "Mapping stream programs onto heterogeneous multiprocessor systems," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 57–66.

[16] W. Thies, "Language and Compiler Support for Stream Programs," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[17] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal, "Baring it all to software: The raw machine," MIT, Cambridge, MA, USA, Tech. Rep., 1997.

[18] W. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *Computers, IEEE Transactions on*, vol. C-24, no. 12, pp. 1235 – 1238, dec. 1975.

[19] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, Dec. 1974.

[20] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 260 –274, Mar. 2002.

[21] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[22] S. Sanyal and S. Das, "Match : Mapping data-parallel tasks on a heterogeneous computing platform using the cross-entropy heuristic," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, april 2005, p. 64b.

[23] S. Kumar, S. K. Das, and R. Biswas, "Graph partitioning for parallel applications in heterogeneous grid environments," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, ser. IPDPS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 167–.

[24] E. S. H. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 2, pp. 113–120, Feb.

[25] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund, "Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments," in *5th Heterogeneous Computing Workshop (HCW'96)*, 1996, pp. 98–117.

[26] M. Wu, W. Shu, and J. Gu, "Local search for dag scheduling and task assignment," in *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*, Aug, pp. 174–180.

[27] A. Malik and D. Gregg, "Executing synchronous data flow graphs on heterogeneous execution architectures using integer linear programming," Trinity College Dublin, Tech. Rep., 2012.

[28] X. Sui, D. Nguyen, M. Burtscher, and K. Pingali, "Parallel graph partitioning on multicore architectures," in *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 246–260.

[29] U. Catalyurek and C. Aykanat, "A hypergraph-partitioning approach for coarse-grain decomposition," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '01. New York, NY, USA: ACM, 2001, pp. 28–28.