An improved simulated annealing heuristic for static partitioning of task graphs onto heterogeneous architectures

Aravind Vasudevan
School of Computer Science and Statistics
Trinity College Dublin
Email: vasudeva@scss.tcd.ie

Avinash Malik
University of Auckland
Email: avinash.malik@auckland.ac.nz

David Gregg
School of Computer Science and Statistics
Trinity College Dublin
Email: David.gregg@scss.tcd.ie

***Abstract*—We present a simulated annealing based partitioning technique for mapping task graphs, onto heterogeneous processing architectures. Task partitioning onto homogeneous architectures to minimize the makespan of a task graph, is a known NP-hard problem. Heterogeneity greatly complicates the aforementioned partitioning problem, thus making heuristic solutions essential. A number of heuristic approaches have been proposed, some using simulated annealing. We propose a simulated annealing method with a novel NEXT_STATE function to enable exploration of different regions of the global search space when the annealing temperature is high and making the search more local as the temperature drops. The novelty of our approach is two fold: (1) we go a step further than the existing scientific literature, considering heterogeneity at levels of task parallelism, data parallelism and communication. (2) We present a novel algorithm that uses simulated annealing to find better partitions in the presence of heterogeneous architectures, data parallel execution units, and significant data communication costs. We conduct a statistical analysis of the performance of the proposed method, which shows that our approach clearly outperforms the existing simulated annealing method.**

I. Introduction and related work

Hardware manufacturers in their eagerness to overcome the limits of silicon have introduced heterogeneous execution architectures. There is evidence [1] of software/compiler developers catching up by inventing techniques for automatic parallelisation of programs onto this heterogeneous hardware. Task graph partitioning and scheduling, onto a heterogeneous architecture, to reduce the overall application latency is a well studied problem [2], [3]. Different approaches have been proposed, some that give optimal solutions for two-processor systems [4], while others that are heuristics for generic hardware topologies [5]. Task graph partitioning and scheduling being NP-hard [6], [7] in the general case, requires one to search for good heuristics that can efficiently utilize the underlying execution architecture (unless P=NP). There are two facets that any heuristic task-partitioning technique needs to consider: (1) correct and efficient modelling of the application to extract the parallelism from the underlying execution architecture and (2) a quick algorithm that can provide close to optimal solutions.

Usually, the algorithm designers concentrate on the second facet; providing an efficient heuristic algorithm (normally some kind of meta-heuristic optimization technique [2]), with the first one taking a back-seat. This observation is based on the fact that none of the proposed heuristic techniques, targeting partitioning and scheduling of task-graphs on heterogeneous hardware explicitly specify the different *types* of potential parallelism available in the underlying heterogeneous hardware. For example, if we consider *Graphical processing units* (GPUs) in conjunction with the standard *Central processing units* (CPUs), we can identify at least three forms of parallelism; (1) multi-core parallelism suitable for graphs composed of independent tasks, (2) short vector parallelism available on CPUs and (3) very large vector parallelism available on GPUs. None of the work to date [3], [8], [9], [2] explicitly tries to extract these different types of parallelism together. Sanyal et al. [9] and Braun et al. [2] consider heterogeneous architectures with different cores running at different speeds. However, they do not target vector parallelism.

In this paper we use a simulated-annealing approach to partition and schedule applications modelled as task-graphs onto heterogeneous architectures addressing the above mentioned gaps in the current literature on task-graph partitioning on heterogeneous hardware.

Our **key contributions** in this paper are:

- Mechanisms to exploit data parallelism within tasks and map the parallelism to processing elements with matching vector capacity.
- A simulated annealing approach to partition the task level parallelism and data level parallelism across heterogeneous multi-core architectures.
- We consider communication costs between heterogeneous units and our technique also allocates the so called *data-stores* indicating placement of variables on the correct processor memory.
- We present a novel adaptation of the METIS graph partitioner tool to the problem of partitioning task graphs onto heterogeneous architectures.
- An experimental evaluation of our approach, comparing it with three established approaches: k-way partition, heterogeneous bin packing, and the existing simulated annealing-based approach.
- Experimental results showing that our approach performs

well in practice, and in general produces superior results compared to the existing approaches.

The rest of the paper is arranged as follows. Section II, formalizes the problem statement. Section III, first describes the established *Simulated Annealing* approach and then explains our contributions and improvements to this approach. Next, we perform experiments to quantitatively validate our SA approach in Section IV and discuss the related work and position our work in comparison to the currently available techniques in Sections IV-C2 and IV-D respectively. Finally, we conclude in Section V.

II. Notations and formalization of the problem statement

The overall application partitioning problem onto a heterogeneous execution architecture can be formulated in terms of a graph partitioning problem. An application can be represented as a *task graph* as discussed in Section II-A. We can represent the underlying parallel execution framework as a *resource graph*. The nodes in this graph are processing units and the communication links are represented by the edges, which is discussed in Section II-B. Given these two graphs, how does one partition the former and map it onto the latter? To answer this question, we extend the execution model presented by Sanyal et al. [9], [10] to accommodate the three kinds of parallelisms mentioned in Section I.

A. Formalizing the task graph

A task graph is a weighted directed graph $G_t(V_t, E_t)$, such that each vertex V_t is a program statement in the application, and $E_t \subseteq (V_t \times V_t)$, shows the communication edges between the vertices. Each vertex V_t is decorated with n weights ($n = 2$ in the case of the experiments conducted): $w_0^t : i \rightarrow \mathbb{N}, \forall i \in V_t$ and $w_1^t : i \rightarrow \mathbb{N}, \forall i \in V_t$. Weight w_0^t is a function on some vertex $i \in V_t$, that maps the amount of work to be carried out at node i (represented as Units of Work (UoW)) to an integer value. Similarly, w_1^t is a function on some vertex $i \in V_t$ that maps the vector length that is required by the node i again to an integer value.

Each edge is decorated by a weight $w^e : e \rightarrow \mathbb{N}, \forall e \in E_t$, where w^e represents the number of bytes that need to be transferred from the location of the *data-store* (i) to the utilization node (j). Data-stores are special nodes, one per data variable, that indicate where the data resides. These special nodes (that have their weights set to zero; thereby making them dummy nodes) are inserted into the task graph wherever reader and writer tasks require access to the corresponding data. The introduction of these nodes however, are done ex post facto. Hence the original dependence amongst tasks in the application graph are preserved by inserting *out* and *in* edges to read and write tasks respectively. The edges that existed between tasks in the graph before the introduction of these special nodes are preserved. The allocation of data-stores on a heterogeneous resource graph plays an important role, since we consider CPU-GPU architectures that do not have shared memory. This implies there is a significant memory latency associated with data transfer between these compute units. Hence the placement of the data-store nodes plays a vital role in minimizing data transfer on these expensive communication links.

B. Formalizing the resource graph

The system resources are represented by a weighted undirected graph $G_r(V_r, C_r)$. Where V_r represents a processing element in the underlying resource graph, while the edge $C_r \subseteq (V_r \times V_r)$, represents the communication links. Each vertex is decorated with weights $W_0^r : i \rightarrow \mathbb{N}$ and $W_1^r : i \rightarrow \mathbb{N}, \forall i \in V_r$. W_0^r maps the amount of work that this resource is capable of doing represented by *Units of Work* (UoW) to an integer value and W_1^r maps the vector capacity of that vertex to an integer domain. Every communication link is weighted with the bandwidth capacity denoted by $W^c : c \rightarrow \mathbb{N}, \forall c \in C_r$. For convenience we use the notation $w_0^t(i)$ for function application in the rest of the paper, same for all other functions.

C. Formalizing the objective function

As stated earlier, we extend the objective function from Sanyal et al.'s work [9], [10] to accommodate the vector parallelism that is now exposed because of the multi-constraint representation of the task and resource graph we employ. Given some application node $i \in V_t$ mapped to some resource $j \in V_r$. The latency for that node is computed as

$$\begin{gathered} Latency(i) = \\ ((w_1^t(i)/W_1^r(j) \times w_0^t(i))/W_0^r(j)) + \sum(w^e/W^c) \\ e = (i,k), k \in succ(i) \end{gathered} \tag{1}$$

where $succ(i)$ represents the successors of the task i; c represents the communication link in the underlying hardware topology which connects PE j to the PE onto which the task k is mapped onto. In this formulation for some given task graph node i, we first calculate the number of vectorized instructions that need to be performed (by diving the required vector length with the vector capacity of the resource node). This gives us the total number of vector instructions that would be performed on the resource node j. Next, we multiply the number of vector instructions to be performed by the UoW required, this in turn gives us the total amount of work to be performed by that task graph node. Finally, we find the computation cost by dividing this total UoW value with the UoW of the resource vertex. For communication on the other hand, we calculate the cost, by dividing the number of required bits to be transferred to the successors of the task by the bandwidth of the resource(s).

Given the task graph and the resource-graph, let ζ be all possible mappings of the application on the resource-graph. We extend the formulation from eq. 1 to find how heavily loaded each PE is. In order to do this, we simply add the latencies of all the tasks scheduled on this PE. This necessitates a sequential execution of all the tasks assigned to this PE. Although this formulation doesn't take the dependencies of tasks that are mapped onto other PEs into account, it still

forms a lower bound for the makespan. Accounting for the completion times of all the parents and data transfer time from their PEs to this PE increases the makespan, thereby making our estimate a tight lower bound. In our formalization and our experiments we are only concerned about the *mapping* and not the *schedule*. Once an effective mapping has been found, a schedule can easily be derived from it using a simple list scheduler.

Let $\zeta_{\mathcal{M}}$ be the mapping under consideration and $\zeta_{\mathcal{M}}(i)$ represent the PE to which the task i has been mapped to. Let s be the processor under scrutiny, then we define the load for such a processor as:

$$\begin{gathered} Load_{\zeta_{\mathcal{M}}}(s) = \\ \sum_{\forall i \mid \zeta_{\mathcal{M}}(i)=s}(((w_1^t(i)/W_1^r(s) \times w_0^t(i))/W_0^r(s)) \\ + \sum w^e/W^c) \\ \text{s.t., } s \in V_r \wedge \{\forall i \in V_i \wedge \zeta_{\mathcal{M}}(i) = s\} \\ \text{and } e = (i,k) : \{k \in succ(i)\} \wedge \\ \{c = (s,l) : \zeta_{\mathcal{M}}(k) = l, \forall l \in V_r\} \end{gathered} \tag{2}$$

Finally, we define the objective function as the most heavily loaded PE according to eq. 2. More formally, the complete objective function can be defined as:

$$Objective^{\zeta_{\mathcal{M}}} = max_{s \in V_t}(Load_s^{\zeta_{\mathcal{M}}}) \tag{3}$$

The goal of our framework is to minimize the total objective function value as described in Equation (3).

asdasd asd asd asd asd asd

III. Simulated annealing

Simulated Annealing [11] is an adaptation of the Metropolis-Hastings algorithm for solving the problem of locating a good approximation of the global optimum of a given function, $\mathcal{F} : \mathbb{R} \rightarrow \mathbb{R}$, which has a large search space. In the context of the problem at hand, we have a 2-dimensional search space, where one axis represents the task ID and the other represents the resource ID. Each point in this 2-D space represents a $\{$*task,resource*$\}$ pair which implies this task is mapped onto this resource. We define a *state* to be a collection of $|V_t|$ points such that each task is mapped to exactly one resource. The total number of possible *states* in this discrete space is $|V_r|^{|V_t|}$ which is exponential. The large number of states make exhaustive enumeration to find optimal solutions, infeasible. Please note that we will use the term *resource* and *processing element (PE)* interchangeably. The same applies for *task graphs* and *application graphs*.

SA is a heuristic algorithm that explores the search space by inspecting one valid state at each iteration. Each of these inspected states are evaluated by an *objective function* which tells us how *good* or *bad* this state is. The *goodness* in an SA algorithm is problem dependent and in our case it is given by the metric defined in Equation 3, Section II-C. The algorithm progresses by inspecting a candidate state at each iteration and it either accepts it as its current state or discards the state and *moves* on to another state. We define a move as the generation of the next candidate state and this progress is governed by a global time-varying parameter called the *temperature* which changes based on an *annealing schedule*.

The algorithm always accepts a move to a better solution, i.e. whenever a new state which has a *better* objective function value than the current state, the SA algorithm accepts it. When this value is worse however, the SA algorithm accepts this move with a certain *acceptance probability*, that changes with the current temperature. When the temperature is high, the algorithm accepts moves to a worse solution with a higher probability; as the temperature reduces over time, this probability decreases as well.

Input: Initial Mapping ζ_0 and Starting and Final Temperatures $\mathcal{T}_0$, $\mathcal{T}_f$
Output: Best Mapping ζ_{best}

```
ζ_current ← ζ_0 ;
C_current ← OBJECTIVE_FUNCTION(ζ_0); //calculate initial objective function value;
ζ_best ← ζ_current;
C_best ← C_current;
R ← 0;
for i ← 0 to ∞ do
    T_current ← NEXT_TEMPERATURE(T_0, i);
    ζ_new ← NEXT_STATE(ζ_current, T);
    C_new ← OBJECTIVE_FUNCTION(ζ_new);
    ΔC ← C_new − C_current;
    r ← RAND();
    p ← ACCEPTANCE_PROBABILITY(ΔC, T_current);
    if ΔC < 0 or r < p then
        if C_new < C_best then
            ζ_best ← ζ_new; C_best ← C_new; ζ_current ← ζ_new;
            C_current ← C_new;
            R ← 0;
        end
    end
    else
        if T_current ≤ T_f then
            R ← R + 1;
            if R ≥ R_max then
                break
            end
        end
    end
end
return ζ_best
```

Algorithm 1: The Conventional Simulated Annealing Algorithm

A. Conventional Simulated Annealing : From the mapping problem standpoint

The algorithm employed by Orsila et al. [8] is given in Algorithm 1. This algorithm takes as input an initial (random) mapping (ζ_0), the starting temperature $\mathcal{T}_0$ and the final temperature $\mathcal{T}_f$, all of which are set by the user. ζ_{best} holds the best mapping found after the algorithm halts. $\zeta_{current}$ and $\mathcal{T}_{current}$ are the current mapping and the current temperature, respectively. The *NEXT_STATE* function moves a random task to a random processing element (PE). For further information about this algorithm, we encourage the readers to read Sections II.B and III from Orsila et al.'s paper [8].

B. Our Improved SA Approach

Simulated annealing is a generic framework that is characterized by the definition of few of its parameters and functions. In this section, we discuss our adaptation of Orsila et al.'s

Application	Vector strip size									
	10		20		30		40		50	
	$\|V_t\|$	$\|E_t\|$	$\|V_t\|$	$\|E_t\|$	$\|V_t\|$	$\|E_t\|$	$\|V_t\|$	$\|E_t\|$	$\|V_t\|$	$\|E_t\|$
Binomial option pricing	82	206	102	306	122	406	142	506	162	606
Convolution	79	143	89	173	99	203	109	233	119	263
Gram Schmidt	228	443	838	1653	1848	3663	3258	6473	5068	10083
Gauss-Seidel	227	531	837	2041	1847	4551	3257	8061	5067	12571
Jacobi	48	130	78	240	108	350	138	460	168	570

TABLE I: The task graph setup

work [8] to suit the heterogeneity in our multi constraint representation of task and resource graphs. We also discuss the rationale behind the generation of next candidate states based on the temperature parameter.

Orsila et al.'s paper [8] primarily dealt with parametrizing the Simulated Annealing algorithm. The authors demonstrated that their heuristic for setting the initial and final temperatures in the annealing schedule was sufficient for the algorithm to produce a good quality result. However, the authors do not mention how the *NEXT_STATE* function is computed, i.e. how the SA method finds the next candidate state. Our *NEXT_STATE* function is one of the most important contributions of this paper.

In conventional SA, temperature is used only in the acceptance probability function to accept worse states as a way of escaping local minimums, which we have retained in our implementation. Additionally, we incorporate temperature in the generation of the next candidate state (the next probable *move* of the SA algorithm). When the temperature is high, we allow a higher proportion of the elements in this mapping of the form $t_k \rightarrow r_i$ for some $t_k \in V_t$ and $r_i \in V_r$ to change. We denote this proportion by,

$$\mathcal{T}_{scaled} = \frac{\mathcal{T}_{current}}{\mathcal{T}_0 - \mathcal{T}_f} \tag{4}$$

where $\mathcal{T}_{current}$ is the current temperature, $\mathcal{T}_0$ is the initial temperature and $\mathcal{T}_f$ is the final temperature.

$|V_t| * \mathcal{T}_{scaled}$ gives us an estimate of how many tasks we can migrate in order to generate the next candidate state. As is evident from eq. 4, the scaled temperature factor allows a lot of tasks to migrate to different processing elements when the temperature is high. As the temperature decreases, we restrict the motion of these tasks, meaning we allow only fewer tasks to migrate to different processing elements and enforce a condition on the rest of the tasks to stay at the processing element they are currently on. Note that during every iteration, the tasks that are allowed to move are selected at random while the number of tasks that have to be moved is given by $|V_t| * \mathcal{T}_{scaled}$.

This is in contrast to the conventional simulated annealing algorithm where only the acceptance probability is affected by temperature. Although the acceptance probability decreases with temperature, the annealing schedule generates candidate states which are distributed randomly throughout the search space. In our method, as the temperature drops, the next candidate state is generated closer to the current best solution. This enables us to fine-tune the current best solution in order to only move tasks that give us better objective function values. This idea of letting temperature influence the generation of the next state opens up a range of optimizations that can be incorporated into the *NEXT_STATE* function. This is something that we plan to explore in our future work.

We have also changed the starting point, ζ_0 of the annealing process. In the conventional algorithm a random starting point is chosen by mapping each task to a random PE. We maintain the randomness in the starting solution, but we mandate all tasks to be mapped onto a single random PE. This puts tightly coupled tasks, i.e. tasks that communicate heavily, onto the same PE and moving them onto different PEs would only incur more communication costs and would thus lead to a worse objective function value.

We use the value for the initial and final temperature for the annealing schedule same as Orsila et al. [8]. To accommodate the multiple-constraint representation model with theirs, we calculate the fastest and slowest processors by multiplying each processor's capabilities (the PE's UoW capability and vector width) and sorting them in non-descending order.

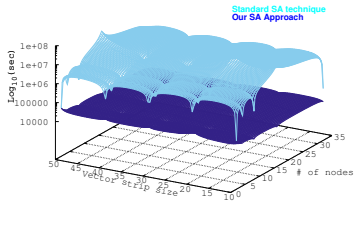
IV. Experiments and Results

Although several list-scheduling heuristics for the problem exist [12], [5], [13], [14], we focus our comparison of the proposed solution against the conventional SA approach in [8]. Furthermore, we also compare our technique with two state of the art heuristic algorithms for allocation of task graphs onto heterogeneous architectures: one based on K-way partitioning [15] and other based on heterogeneous bin-packing [16]. Statistical analysis is performed on a large set of randomly generated set of heterogeneous execution architectures (resource graphs) and real-world applications (task graphs).

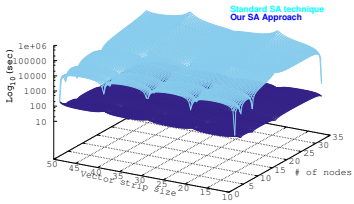
We show the speedup obtained using our improved heuristics against the conventional heuristics for SA as prescribed by [8]. We also compare the results we obtain against the K-way graph partitioning algorithm [15] and heterogeneous bin packing heuristic [16].

A. K-way graph partitioning

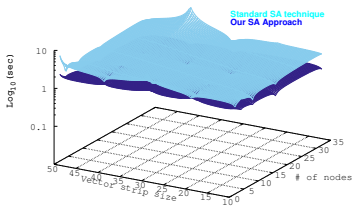
Graph partitioning plays an important role in the multiprocessor and VLIW scheduling and partitioning algorithms [17]. K-way graph partitioning is an important algorithm, which partitions a given graph into K or less parts, resulting in load balanced allocations. K-way partitioning mixed with min-edge cuts can form a good tool to partition



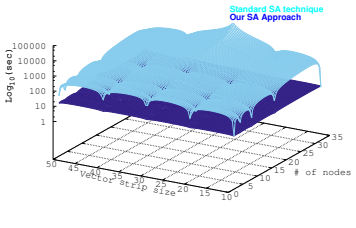
(a) Binomial Option Pricing



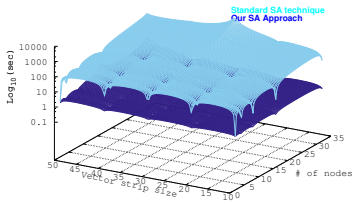
(b) 2 Dimensional Convolution



(c) Gram Schmidt linear-algebra kernel



(d) 2 Dimensional Seidel stencil computation



(e) 2 Dimensional Jacobi stencil computation

Fig. 1: Comparison of $Objective^{\zeta,\mathcal{M}}$ from eq. 3 : Our SA approach vs standard SA

a task-graph onto a multi-processor system, resulting in equal utilization of PEs and reduced communication costs. We have utilized the METIS [18] graph partitioning tool to perform a K-way graph partitioning onto heterogeneous multi-processor architectures for comparison purposes.

METIS is a graph partitioner, which implements K-way partitioning with min-edge cut as the primary objective. The weights on the graph nodes are represented as constraints. Each graph node can have multiple-node weights representing different criteria. The edges between nodes can be weighted themselves, but as opposed to nodes, edges can only be decorated with a single weight. Moreover, in METIS one can use a concept called *tp-weights*, which gives preference to different node constraints when performing load-balancing during K-way graph partitioning [18].

The gist of our K-way task partitioning approach onto a heterogeneous multi-core architecture is as follows:

- Our resource graph is first described as a simple graph in METIS. In this description each of the two capabilities W_0^i and W_1^i are described as two constraints for each node in the graph. The communication bandwidth is described as edge weights in METIS.
- Once we have the resource graph in the METIS format we calculate the tp-weights. There are two tp-weights generated, one for each of the resource node capabilities. For each processor the UoW tp-weight is calculated by the formula: $W_0^i / \sum_{\forall i \in V_r}(W_0^i)$. Similarly, we can easily calculate the tp-weight metric for each PEs vector length capability as: $W_1^i / \sum_{\forall i \in V_r}(W_1^i)$. These fractions give a relative approximation of the capabilities of each PE compared to the other. For example, given two nodes with $W_0^0 = 100$ and $W_0^1 = 50$, then the first node has a tp-weight of, 0.667, while the second has a tp-weight of, 0.333.
- The nodes in our task graph are described as graph nodes in the METIS format. The two requirements (R_0^i and R_1^i) for each task node are described as two constraints in the METIS node format. The edge weights in our task graph are described as edge weights in the METIS graph format.
- Once we have the task graph described in the METIS format along with the tp-weight metric for each PE in the resource graph. We ask for a $|V_r|$ partition from Metis, giving the tp-weight metric for each constraint of the task graph.
- The resultant partition is then used to calculate the objective in Equation (3).

B. Heterogeneous bin packing heuristic

Heuristic bin packing solutions have given good results in the general case [19]. Comparing with the heterogeneous bin packing heuristic [16] allows us to gauge the effectiveness of our algorithm against a standard technique.

Let $\mathcal{I}$ be the items to be accommodated into the bins and let $\mathcal{K}$ be the set of bins available. From the standpoint of the mapping problem, $\mathcal{I}$ refers to the set of task graph nodes ($|V_t|$) and $\mathcal{K}$ refers to the nodes in the resource graph ($|V_r|$). Similar to the Knapsack problem [20], by which A-BFD (*Adaptive Best First Decreasing*) is inspired, each element $i \in \mathcal{I}$, $\mathcal{K}$ has two constraints on them represented by c_i (cost), which translates to the PE capability W_0^i and V_i (volume), which translates to the capability W_1^i, respectively. *A-BFD* proceeds to sort $\mathcal{I}$ according to non-increasing order of their volume and sorts $\mathcal{K}$ according to non-increasing order of the ratio c_i/V_i. Then, it proceeds to allocate items from $\mathcal{I}$ into best bins $b \in \mathcal{S}$.

Application	Max objective value		Min objective value		Better (%)
	K-way	Our SA	K-way	Our SA	Our SA vs K-way
Bin. option pricing	79947.2	53404.8	11218.1	10975.5	94
Convolution	52.85	124.309	19.64	19.64	12
Gram Schmidt	54.13	2.96	1.33	0.91	64
Gauss-Seidel	542.44	32.99	9.67	9.67	68
Jacobi	16	14.69	0.95	0.89	32

(a) Statistics comparing our SA approach and K-way graph partitioning. Our SA algorithm was run for 10 minutes per simulation run

Application	Max objective value		Min objective value		Better (%)
	HBP	Our SA	HBP	Our SA	Our SA vs HBP
Convolution	215741	124.309	98.45	19.64	92
Gram Schmidt	3169.72	2.96	2947.65	0.91	92
Jacobi	4653.66	14.69	1.69	0.89	92

(b) Statistics comparing our SA approach and heterogeneous bin packing. Our SA algorithm was run for 10 minutes per simulation run

Application	Max objective value		Min objective value		Better (%)
	Std. SA	Our SA	Std. SA	Our SA	Our SA vs Std. SA
Bin. option pricing	219230	53404.8	25967.5	10975.5	95
Convolution	220525	124.31	50.34	19.64	76
Gram Schmidt	10.04	2.96	0.96	0.91	92
Gauss-Seidel	14607.8	32.99	9.07	9.67	72
Jacobi	3504.44	14.69	0.95	0.89	88

(c) Statistics comparing our SA approach and the standard SA. Both algorithms were run for 10 minutes per simulation run

TABLE II: Statistical comparisons of $Objective^{\zeta_{\mathcal{M}}}$ from equation (3) for different benchmarks

A **best** bin, i.e., the bin with maximum free space, is defined as the bin volume minus the sum of volumes of the items loaded into it.

The post pass in *A-BFD* chooses every bin that has at least one item allocated to it and tries to find an empty bin, that has a higher or equal volume than the allocated volume on the chosen bin but also has a lower cost. If it finds such an empty bin, then it transfers all the items allocated to the chosen bin to the newly found empty bin which is cheaper. One of the main advantages of *A-BFD* is that it is very fast with a best case complexity of $O(N_{\mathcal{I}})$ without the post pass, where $N_{\mathcal{I}}$ is the number of items (number of tasks $|V_t|$ in the application graph G_t). Including the post pass, the best case complexity becomes $O(N_{\mathcal{I}}+N_{\mathcal{K}})$ where $N_{\mathcal{K}}$ is the number of bins (number of PEs $|V_R|$ in the resource graph G_r).

C. The experimental setup

We set up the resource graphs and the task graphs for performing the experiments as follows.

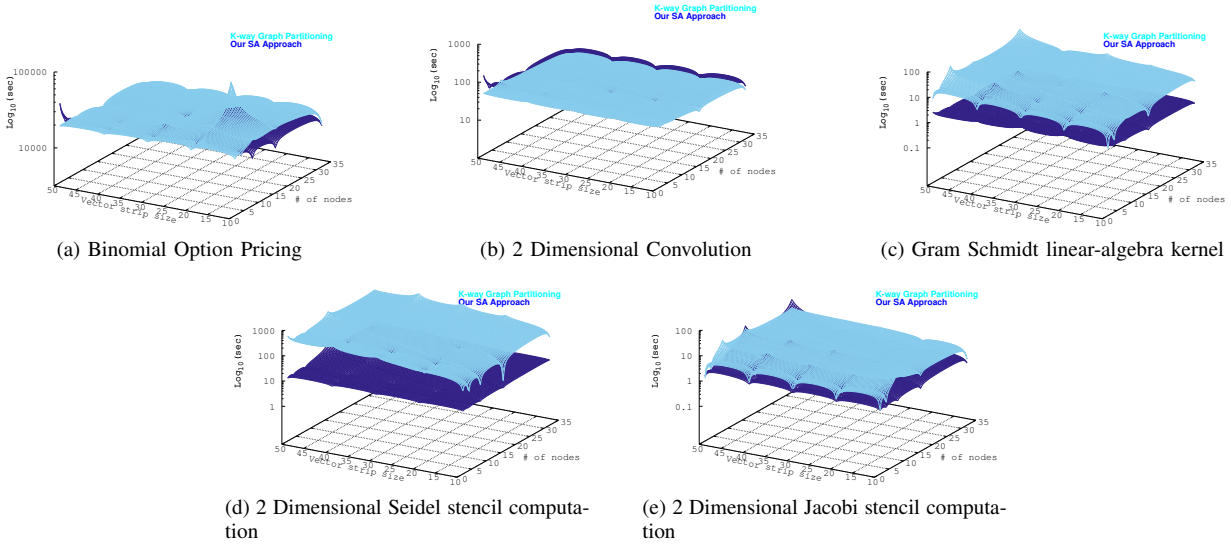
1) The resource graph setup: The experimental setup consists of the following:

1) A multi-core system with $|V_r|$ nodes. A node could be just a multi-core CPU or a multi-core CPU with a GPU attached to it. $|V_r|$ varies in a normal distribution from 2 to 32.
2) The bandwidth is selected from a set $\boldsymbol{B} = \{B_1, B_2, ..., B_{|\boldsymbol{B}|}\}$. Every communication link weight (W^c) is selected from the set $\boldsymbol{B}$. The elements of the set $\boldsymbol{B}$ varies in the normal distribution: 1 GB/s to 10 GB/s, representative of the multi-core connection networks in today's machines.
3) A set of N_G GPUs where N_G is at most $|V_r|$. The GPUs are connected in the network at pre-determined locations, chosen randomly in the normal distribution of 25% to 75% of $|V_r|$.
4) A set $\boldsymbol{V} = \{V_1, V_2, V_3, ...V_{|\boldsymbol{V}|}\}$ where V_i is a power of 2. Every GPU in this experiment has a vector length of V_i where V_i is sampled randomly from the set $\boldsymbol{V}$. The elements of set $\boldsymbol{V}$ are chosen from a normal distribution ranging from 2 to 32.
5) A set $\boldsymbol{M} = \{M_1, M_2, M_3, ...M_{|\boldsymbol{M}|}\}$ where M_i is a power of 2. Every $C_i \in \boldsymbol{C}$ and GPU in this experiment has a UoW value 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 2^7 to 2^{20}.

For given values of $|V_r|$, N_G, $\boldsymbol{V}$, $\boldsymbol{C}$, and $\boldsymbol{M}$ 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 $\boldsymbol{V}$ and $\boldsymbol{M}$ randomly to determine its vector length V_i and UoW 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 $\boldsymbol{V}$ and $\boldsymbol{M}$.
- Use our framework to extract data and task parallelism that is best utilizable by the heterogeneity created by parameters in items 1, 2, and 3 above. Determine the execution time $Objective^{\zeta_{\mathcal{M}}}$ using eq. 3.

An experiment, $\boldsymbol{E}$ ($|V_r|$, N_G, $\boldsymbol{V}$, $\boldsymbol{C}$, $\boldsymbol{M}$), consists of conducting enough of the above trials so that width of the 95% confidence interval on the average value of $Objective^{\zeta_{\mathcal{M}}}$ is less than 10% of the average value. This results in a variable number of trials with different experimental setups. Note that



(a) Binomial Option Pricing

(b) 2 Dimensional Convolution

(c) Gram Schmidt linear-algebra kernel

(d) 2 Dimensional Seidel stencil computation

(e) 2 Dimensional Jacobi stencil computation

Fig. 2: Comparison of $Objective^{\zeta_M}$ from eq. 3 : Our SA approach Vs K-way graph partitioning

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) The task graph setup: We chose 5 applications: binomial option pricing (a financial derivatives application), 2-dimensional convolution (for image processing), Gram Schmidt linear algebra kernel, 2-dimensional Gauss-Seidel stencil computations, and finally our motivating example itself the 2-dimensional Jacobi stencil computation.

The compiler first collapses all the nested loops and vectorizes the resultant statement to the largest possible vector length, if it is able to do so. Once vectorized to a large vector of some length N (N can be of length 1 million in some cases) we strip this vector, breaking it into multiple nodes requiring $N/(\text{vector strip size})$ each. The resultant nodes are then allocated onto the heterogeneous processor. We take this approach to increase the exploitation of parallelism; especially taking into consideration the GPUs that provide the ability to perform large vector operations at once.

For the 5 aforementioned applications we vary the vector strip from 10 to 50, which results in graphs varying from around 50 to 5000 nodes and with 23 to 12,000 edges. A detailed description of the applications and their features is shown in Table I. The vector requirement of applications in our benchmark suite varies from 1000 to 1 Million elements. The UoW count varies from around 1 to 0.3 billion. The edge weights depicting the amount of data transfer on the other hand varies from 3000 bytes to almost 4.8 Mega byte.

D. Experimental results

The comparison of our simulated annealing approach with other techniques are described in the next sub sections. In all the upcoming comparisons we set the value of $q = 0.75$(which controls how fast the temperature drops) in our SA and standard SA techniques. This value of q is a good compromise between a slow running SA heuristic vs objective function value, since a bigger value of q, results in larger explored state space. For the aforementioned experimental setup the standard SA and our SA techniques were run for 10 minutes.

1) Comparison with K-way partitioning: The K-way graph partitioning algorithm uses the METIS [21] graph partitioner to solve the problem of mapping task graphs onto hardware topologies. We discuss the adaptation of the METIS graph partitioner to suit the multi constraint representation model in Section IV.A of our previous work [15]. The major statistics comparing the two approaches is provided in Table IIa. In Table IIa, The **Max objective** and the **Min objective** columns, provide the maximum and minimum application latencies for each of the applications. The last column gives the % of instances our SA technique performed better than K-way graph partitioning, in the experiments conducted. The METIS project has been under development for the past ten years and is extremely well suited for performing min-cuts of large graphs and minimizing cross partition communication. This is clearly reflected in the results of two data intensive benchmarks, namely Convolution (better in 88% of the experiments) and 2D Jacobi Stencil computation (better in 68% of the experiments).

2) Comparison with heterogeneous bin packing heuristic: Comparison of our SA technique with heterogeneous bin packing heuristic is shown in Table IIb. Unlike, K-way graph partitioning, the bin packing heuristic could not partition the binomial option pricing and Gauss-Seidel examples for any vector strip size, because the algorithm terminates if there is no underlying PE that can support the required vector tile size. Our SA heuristic runs a part of the vector on the underlying PE and then iterates in a loop until all vector computations

are finished. For example, if the task graph requires a 1000 vector elements that need to be processed at once, and the largest vector capability in the resource graph is a 100, then, our heuristic will allocate a 100 vector elements onto the underlying hardware and then increase the UoW count by 10.

Heterogeneous bin packing prioritizes volume (vector length) over cost (UoW count; see Section IV-B) thereby performing much worse compared to our approach. It packs a large number of task nodes from the application graph into a single large vector PE, including those task nodes, which have a small vector count (w_1^i), but a large UoW count (w_0^i).

3) Comparison with conventional SA: The table showing the comparison of our approach with the established SA approach of [8] is shown in Table IIc. There is not a single case where our SA approach is worse than the current established technique. The reasons for this are two fold,

- We change the annealing schedule such that the search exploration is global initially, i.e. high temperatures, and it becomes local to the current best solution as temperature drops.
- The greedy phase in our annealing schedule helps us fine tune our best solution

In all the figures and statistics presented, we ran the conventional SA and our improved SA approach with a time limit of 10 minutes, i.e. SA will terminate after 10 minutes irrespective of whether it has already found an acceptable solution. The experiments were performed on a quad-core dual processor system with 24 gigabytes of RAM.

V. Conclusion

In this paper we have described a novel *Simulated Annealing* (SA) heuristic for mapping applications with task and data level parallelism onto heterogeneous execution architectures. We partition and schedule such applications onto heterogeneous architectures with differing compute costs, vector lengths, and communication bandwidths. To our knowledge we are the first to utilize SA for extracting different kinds of parallelism (task and data) directly from compilers onto heterogeneous architectures.

Our SA approach guides the movement of the partition using temperature itself, as opposed to others, where such movement occurs randomly. Moreover, a guided movement allows for faster identification of good solutions, resulting in faster runtime for the SA algorithm, making it scalable to larger applications and larger number of processor cores. Experimental results show that our SA approach outperforms the existing SA technique in 84% of the instances. Moreover, for all these instances it has a better objective function value. We also compared our SA technique with the well established K-way graph partitioning and heterogeneous bin packing heuristics. Our SA approach out performs the former in 54% of the instances and the latter in 92% of the instances. As part of the future work, we intend to run some instances of the applications on partitions suggested by our method to confirm the validity of the results as the objective function is an approximate estimation of the real runtime.

VI. Acknowledgement

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - Irish Software Engineering Research Centre (www.lero.ie) and IRC Enterprise Partnership Scheme in collaboration with IBM Research, Ireland.

References

[1] J. Humphrey and K. Spagnoli, "Scheduling operations for massive heterogeneous clusters," 2013.

[2] 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 *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed computing*, vol. 61, no. 6, pp. 810–837, 2001.

[3] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[4] A. P. E. Coffman Jr and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta Informatica*, vol. 1, no. 3, pp. 200–213, 1972.

[5] H. Topcuoglu, S. Hariri, and M.-y. 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, 2002.

[6] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors*. MIT press, 1989.

[7] J. D. Ullman, "Np-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[8] H. Orsila, T. Kangas, E. Salminen, and T. D. Hamalainen, "Parameterizing simulated annealing for distributing task graphs on multiprocessor socs," in *System-on-Chip, 2006. International Symposium on*. IEEE, 2006, pp. 1–4.

[9] S. Sanyal and S. K. 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*. IEEE, 2005, pp. 64b–64b.

[10] A. Jain, S. Sanyal, S. K. Das, and R. Biswas, "Fastmap: a distributed scheme for mapping large scale applications onto computational grids," in *Challenges of Large Applications in Distributed Environments, 2004. CLADE 2004. Proceedings of the Second International Workshop on*. IEEE, 2004, pp. 118–127.

[11] S. Kirkpatrick, D. G. Jr., and M. P. Vecchi, "Optimization by simmulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[12] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618

[13] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Task scheduling algorithms for heterogeneous processors," in *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*. IEEE, 1999, pp. 3–14.

[14] H. Arabnejad and J. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," 2013.

[15] S. Muralidharan, A. Vasudevan, A. Malik, and D. Gregg, "Heterogeneous multiconstraint application partitioner (hmap)," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*. IEEE, 2013, pp. 999–1007.

[16] M. M. Baldi, T. G. Crainic, G. Perboli, and R. Tadei, "The generalized bin packing problem," *Transportation Research Part E: Logistics and Transportation Review*, vol. 48, no. 6, pp. 1205–1220, 2012.

[17] K. M. G. 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, 1999.

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

[19] E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM Journal on Computing*, vol. 7, no. 1, pp. 1–17, 1978.

[20] S. S. Skiena, "The algorithm design manual. 2008."

[21] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 1998, pp. 1–13.