Mapping Parallel Programs to Heterogeneous CPU/GPU Architectures using a Monte Carlo Tree Search

Mehdi Goli
and John McCall
School of Computing
Robert Gordon University
Scotland, UK
Email: {m.goli, j.mccall}@rgu.ac.uk

Christopher Brown,
Vladimir Janjic
and Kevin Hammond
St Andrews University,
Scotland, UK
Email: {cmb21, vj32, kh}@st-andrews.ac.uk

Keywords—Montecarlo Tree Search; Heuristic Algorithm; Static Mapping; Parallel Programming; Heterogeneous Architecture;

Abstract—The single core processor, which has dominated for over 30 years, is now obsolete with recent trends increasing towards parallel systems, demanding a huge shift in programming techniques and practices. Moreover, we are rapidly moving towards an age where almost all programming will be targeting parallel systems. Parallel hardware is rapidly evolving, with large heterogeneous systems, typically comprising a mixture of CPUs and GPUs, becoming the mainstream. Additionally, with this increasing heterogeneity comes increasing complexity: not only does the programmer have to worry about where and how to express the parallelism, they must also express an efficient mapping of resources to the available system. This generally requires in-depth expert knowledge that most application programmers do not have. In this paper we describe a new technique that derives, automatically, optimal mappings for an application onto a heterogeneous architecture, using a Monte Carlo Tree Search algorithm. Our technique exploits high-level design patterns, targeting a set of well-specified parallel skeletons. We demonstrate that our MCTS on a convolution example obtained speedups that are within 5% of the speedups achieved by a hand-tuned version of the same application.

I. INTRODUCTION

Ongoing rapid progress in technologies such as GPGPU and CPU/GPU clusters and emerging platforms such as CUDA and OpenCL are widening applicability of high performance computing. Data and stream-based parallelism in particular can now be brought to bear on programming tasks in many engineering and manufacturing operations. However, the process of synchronisation and communication among components in a given parallel architecture, essentially to obtain maximal performance gains, is still a challenging issue and requires expert parallel programmers. In this paper we take an approach that exploits high-level parallel patterns, known as Algorithmic Skeletons, which are a high-level abstraction for parallellising applications [1]. They have long been considered as a viable approach to encapsulating coordination, communication and synchronisation among components. [2]

Stream-based parallelism can be regarded as a generic form of pipeline in which the parallel program operates repeatedly on a stream of data items, e.g. an audio or video stream. Typically a parallel task on each execution step reads one or more data items from one or more input streams, processes the input data and passes the processed data to one or more output streams. [3]

The problem of mapping a stream-based parallel program to a given parallel processing architecture can generally be divided into two stages:

1) The application is organised into a skeleton tree, describing the high-level structure of the parallel application. This includes identifying the computational components of the application and the dependencies between them. A computational component is a side-effect-free computational entity, usually corresponding to a unit of work (or a function) in the source program.

2) Taking into account a specific parallel architecture on which the application is to be executed on, the number of instances of each component is decided, and also a mapping of these instances to the parallel architecture is derived.

Separating these two tasks enables us to abstract away from the underlying hardware in the first phase, which increases the portability of parallel applications. Once the common skeleton tree is introduced, the mapping can be retuned for different specific target architectures. The main focus of this paper is to develop an algorithm that automatically derive mappings that optimise performance for a given skeleton tree to a given heterogeneous architecture.

In a homogeneous architecture, each component in the skeleton tree is of the same type (CPU-type). An instance of a component can be executed as a thread in a multi-core system. On a heterogeneous architecture, comprising a mixture of CPUs and GPUs, it is possible for a component to have a CPU and/or a GPU implementation. The optimal performance of the application depends on selecting the appropriate amount and type of each component, depending on the computational cost of components and the underlying hardware. The problem of mapping a parallel program to a heterogeneous architecture is, therefore, considerably more complex than for a homogeneous
architecture.

In this paper, we investigate the applicability of the Monte Carlo Tree Search (MCTS) approach for deriving optimal mappings of applications for heterogeneous CPU/GPU architectures. We are restricting ourselves to static mappings, where all of the decisions about mapping components to the hardware are taken before the application execution. Monte Carlo Tree Search [4] (MCTS) is an approach originating from AI Games to efficiently select moves in large game trees where a full evaluation of the tree is computationally intractable. MCTS uses Monte Carlo random walks to evaluate states only at the endpoints of each walk. Nodes of the MCTS decision tree in our case are partial mappings of components to the hardware, with leaves of the complete tree representing complete mappings, where all components are mapped. The random walk is biased by the parameters of the static mapping. We use simulations to evaluate complete mappings at the end of the walk.

II. RELATED WORK

A. Mapping process

The static mapping problem is not a new challenge and there is an extensive body of work on mapping task, data and pipeline parallelism to parallel architectures providing static partitioning [5], [6], [7], using runtime scheduling [8], heuristic based mappings [9], analytical models [10], [11], or ILP solvers [12]. Each of these can improve system performance. There are some heuristic based approaches which automate the process of mapping to multi-core architectures for specific frameworks, such as the learning approach used for partitioning streaming in the StreamIt framework [13] or the runtime adaptation approach used in the FlexStream framework [14]. Also, a series of works has aimed to optimise the use of resources on multi-core embedded platforms linking both design-time optimisation and simulation with run-time optimisation using lightweight heuristics [15], [16], [17]. In [18], the use of platform simulators was considered to identify Pareto-optimal design configurations of parallel applications (incorporating code versions, resource mappings, constraints and costs).

Despite the amount of work done in the homogeneous environment, to our best knowledge there is little work done for mapping to heterogeneous(CPU/GPU) architectures. Most of the work on GPUs are primarily focused on application performance tuning [19] rather than orchestration. In [20] a method is provided to orchestrate the execution of heterogeneous StreamIt program presented on a multi-core platform equipped with an accelerator. They use integer linear programming (ILP) formulations to perform partitioning over the combination of CPU/GPU. Formulating ILP models, however, requires expert knowledge of the architecture and finding a solution under certain constraints for the ILP formula can be time-consuming.

Our aim in this paper is automatic orchestration of CPU/GPU components for FastFlow applications by using MCTS. Using MCTS usually requires less knowledge about the system. Therefore, minimum information is needed to run the result on the system. Moreover, applying mapping in a static manner has a much lower overhead for deployment.

B. Monte Carlo Tree Search

Monte Carlo Tree Search has classically been applied to challenging game playing, for example the Go and Bandit problem [21], [22], [23], [24].

Recently MCTS has been applied to planning and scheduling problems. In [25] a MonteCarlo search algorithm was used to produce management problems which can be regarded as single-agents selecting a sequence of actions with side effects, leading to high quantities of one or more goal products. The algorithm achieves a successful solution faster than an existing method. In [26] a Monte Carlo Random Walk planning algorithm is used in deterministic classical planning achieving results comparable to state of the art algorithms. In [27] a version of UCT was successfully applied to continuous stochastic problems with continuous action space.

In this paper we establish the applicability of MCTS to the seamless orchestration of hybrid(CPU, GPU) components over a heterogeneous environment. Although we work on the FastFlow framework here, the technique can be applied to a different framework by adapting the evaluation method.

III. BACKGROUND

A. Skeletons

An algorithmic skeleton is an abstract computational entity that models some common pattern of parallelism. A skeleton is typically implemented as a high-level function that takes care of the parallel aspects of a computation (e.g., the creation of parallel threads, communication and synchronisation between these threads, load balancing etc.), and where the user provides a sequential code specific to his concrete problem.

In this paper, we restrict ourselves to two fundamental, heterogeneous skeletons, that are popular and widely used:

- The Pipeline skeleton models the application of a composition of functions \( f_1, f_2, \ldots, f_n \) to a sequence of independent inputs \( x = x_1, x_2, \ldots, x_m \), where the output of \( f_i \) is the input to \( f_{i+1} \). The parallelism arises from the fact that, for example, \( f_1(x_{n-1}) \) can be executed in parallel with \( f_2(f_1(x_{n-1})) \). We will denote the pipeline skeleton by \( \text{Pipe}(f_1, f_2, \ldots, f_n, x) \).

- A Farm skeleton models the application of a function, \( f \), to a sequence of independent inputs, \( x = x_1, x_2, x_3, \ldots, x_m \). We will denote the farm skeleton by \( \text{Farm}(nw, f, x) \), where \( nw \) is the number of worker threads.

B. FastFlow

In this paper, we demonstrate the applicability of our approach using FastFlow as an example skeletal framework for C++. However, the technique described in this paper is not confined only to FastFlow, but is completely generic, allowing it to be applied to other languages and framework.

FastFlow [28] is a skeleton-based parallel programming framework for multi-core platforms, implemented in C++. FastFlow allows programmers to focus on application-specific computational components by abstracting over complex coordination and communication layers. It supports skeletons for heterogeneous CPU/GPU architectures [29].
C. Monte Carlo Tree Search

MCTS uses four main steps. (i) in the recursive selection step nodes are optimally selected until the most urgent expandable node is reached; A node is expandable if it represents a nonterminal state and has unvisited children; (ii) expansion: create one or more children and select one; (iii) evaluate a node by sampling: create child nodes at random until a leaf is reached and a result obtained; and (iv) finally, in back-propagation: update each node in the current evaluation path with the results. [30]. At any time, a node contains an estimated value (from simulation results) and the number of times it has been visited.

IV. THE PROPOSED APPROACH

We assume that we are given a parallel program, together with its associated skeleton tree structure. A skeleton tree structure provides a high-level view of the skeletons used in a program, together with information about their nesting and the sequential components used. We are also given whether is a CPU or GPU component and the estimated runtime of that component for one input on a given hardware. Figure 1 shows an example skeleton tree structure, with three components (A, B, G, D). Components A, B and C are CPU components, whereas component G is a GPU component. Components B and G correspond respectively to CPU and GPU implementations of the same function.

We define a static mapping of a parallel program to a given hardware as an assignment of the amount of appropriate resources (CPUs or GPUs) to each component in the program skeleton tree structure.

The problem of finding an optimal static mapping for the given program is then divided into two phases:

- Converting the program’s skeleton tree structure to its associated process graph
- Apply Monte Carlo Tree Search to generate an optimal mapping of the process graph to the target architecture

A. Converting the Skeleton Tree to the Process Graph

The skeleton tree is a nested combination of farms and pipelines which is created by the user. Figure 1 shows the skeleton tree of a 3-stage pipeline. Each stage is a farm and the second stage contains CPU and GPU components.

The skeleton tree is converted into a first order logic formula from which a process graph can be created. In case the computational resources available for a particular factorisation are not sufficient, the process graph is reshaped by removing leaves for which the formula remains correct after their removal. Any skeleton tree can be converted to a first order logic formula, using the following scheme:

\[
\begin{align*}
F_{\text{farm}}(A_1, \ldots, A_n) & := \bigvee_{i=1}^{n} A_i \\
F_{\text{pipe}}(A_1, \ldots, A_m) & := \bigwedge_{i=1}^{m} A_i \\
A_i & := F_{\text{pipe}}(A'_1, \ldots, A'_m)|F_{\text{farm}}(A'_1, \ldots, A'_n)|f \\
\end{align*}
\]

where \( f \in \{\text{set of components}\} \).

As an example, for the skeleton tree presented in Figure 1, the formula is as follows. \( F_{\text{OIL}}(TS) = (A \land (B \lor G) \land C) \).

To create a process graph from the formula, each proposition represents a component, each operator \( \land \) represents a queue connecting the components of the systems together and the operator \( \lor \) is used to represent different components of the system streaming data from the same queue. Figure 2 shows the process graph generated from \( F_{\text{OIL}}(TS) \). The process graph has three task queues for the top-level pipeline. Queues \( Q_0 \) and \( Q_1 \) connect the subsequent pipeline stages, and the queue \( Q_2 \) accumulates the output of the whole pipeline. The task queues are the main factor for finding the optimal solution. For each task queue, the queue level is the number of tasks waiting to be processed by the next stage. In an ideal case the queue level for all task queues is zero, because as soon as a task enters the task queue, it will be processed by the next stage. However, this does not occur in reality, especially when we have heterogeneous components running on heterogeneous resources. In this case the number of resources allocated to each component is key to creating that queue level balance.

The more the queue levels of task queues in the system are similar and close to zero, the better the provided mapping. The queue level along with the throughput of the system, which is measured as the number of tasks in the last task queue in time interval \( T \), are the main evaluation factors for a mapping. Therefore, at each step the MCTS approach, selects a mapping solution, simulates the execution of the problem for the selected mapping and rewards it by analysing the simulation based on the above main factors. The result of the reward will be backpropagated based on the applied backpropagation policy. In the following we explain the proposed MCTS approach in more detail.

B. A Monte Carlo Tree Search from the Process Graph

We search a decision tree whose nodes are mapping decisions. A single decision corresponds to allocating an amount of remaining resources to one or more components. At each decision point, some resources will be allocated, limiting the total resources available to the yet unallocated components. A leaf of the tree corresponds to a complete static mapping,
where all components have been allocated an amount of resources. At this point the mapping can be evaluated by simulation, and so we apply MCTS to search for an optimal path through this decision tree.

At each step, a set of resources is allocated to components of a single farm. We can represent the set of possible decisions as:

$$\{A(w_i, k) \mid w_i \in M, k \in \{0, 1, \ldots, L\}\},$$

where $$A \in \{\text{ADD, REMOVE}\}; M$$ is the set of components in the farm (containing one element if there is only a CPU or a GPU component, or two if there are both a CPU and a GPU component); $$k$$ is the amount of resources to be allocated to a component, $$w_i$$; and, $$L$$ is the maximum amount of resources that can be allocated to any component.

### C. Selection Strategy

The selection strategy that we use is Upper Confidence bounds applied to Trees (UCT) [31], [22]. The formula for UCT is:

$$UCT = \bar{X}_j + 2C_P \sqrt{\frac{2 \ln n}{n_j}}$$

where $$n$$ is the number of times the current node has been visited; $$n_j$$ is the number of times the child, $$j$$, has been visited; $$C_P > 0$$ is a constant value; and, $$\bar{X}_j$$ is the average reward value given to child node, $$j$$.

### D. Simulation of Static Mapping

There are two ways to evaluate the performance of a selected path in the decision tree: mathematical abstraction of the system (such as a cost model) and simulation. Here we use simulation because the accuracy of a cost model depends on large number of parameters that may be difficult to obtain for certain skeletons and hardware systems. Our approach that is intended to be approximate, however, the precision gained using simulations is good enough for comparing different static mappings and allows us to apply our technique.

We have developed a simulator for FastFlow that mimics the behaviour of a given FastFlow application running on the target architecture. The simulator outputs the metrics (queue Level, component utilisation, throughput) that we use to evaluate the static mappings.

### E. Reward Function

Once a static mapping has been simulated, a reward for the mapping is calculated. The reward function is based on the throughput of the system, denoted by $$T$$. There are two balancing factors related to the overall utilisation of the system:

1) We define utilisation of a component, denoted $$U(w)$$, to be the utilisation of the resources allocated to the component, $$w$$. We denote by $$SD_U$$ the standard deviation from the mean utilisation of all components in the system:

$$SD_U = \sqrt{\frac{\sum_i(U(w_i) - U_{\text{mean}})^2}{N}},$$

where $$N$$ is the total number of components in the system; $$U(w_i)$$ is the utilisation of the component, $$w_i$$; and, $$U_{\text{mean}}$$ is the average utilisation of all components in the system. Using $$SD_U$$ as a reward function discourages the allocation of additional resources to a component in the case where this results in only a minor gain in the overall program speedup (and in reduced utilisation).

2) We define throughput factor as $$SD_Q$$, the standard deviation of the mean queue throughput, defined as follows.

$$SD_Q = \sqrt{\frac{\sum_i(T_i - T_{\text{mean}})^2}{L}}$$

Here, $$L$$ is the total number of queues in the system, $$T_i$$ is the throughput of the $$Q_i$$, and $$T_{\text{mean}}$$ is the average value of the throughput of all queues in the system. Adjusting the reward for this factor discourages the allocation of additional resource to the components of a queue when they are no longer bottlenecks on that queue.

In the case where a program is executed on an environment where the resources are not necessarily free-of-charge (e.g. cloud infrastructure), we may add two penalty factors:

1) GPU penalty, $$P_{\text{GPU}} = \sqrt{\frac{N_{TG}}{NTG}}$$, where $$N_{TG}$$ is the number of GPUs in the system that have not been used by the static mapping, and $$NTG$$ is the total number of available GPUs in the system.

2) CPU penalty, $$P_{\text{CPU}} = \sqrt{\frac{N_{TC}}{NTC}}$$, where $$N_{TC}$$ is the number of CPU cores in the system that have not been used by the static mapping, and $$NTC$$ is the total number of available CPU cores in the system.

The main effect of $$P_{\text{GPU}}$$ and $$P_{\text{CPU}}$$ is to force the system to use all of the available resources, by introducing a penalty for unused resources. If we add these two factors to the definition of the reward function, then in the case where more than one mapping achieves the same throughput and utilisation, the one that uses the smallest amount of resources is chosen. Conversely, if we omit these two factors, the mapping that uses the largest amount of resources is chosen.

Since our assumption is that the system on which programs are executed is dedicated private machine, we do not use $$P_{\text{GPU}}$$ and $$P_{\text{CPU}}$$ in the reward function, which is therefore

$$Q(v) = T - (SD_U + SD_Q),$$

for a selected path, $$v$$, of the decision tree. If we were to use the two penalty factors, the reward function would become

$$Q(v) = T - (SD_U + SD_Q + P_{\text{CPU}} + P_{\text{GPU}}).$$

### F. Back-propagation, Termination Condition and Final Move Selection

We have considered two back-propagation policies [30]: the Max policy, where the maximal reward of all children is propagated to the parent, and the Average policy, where the average reward of all children is propagated to the parent.

The MCTS algorithm finishes if no new moves have been made for $$K$$ iterations. The final move selection is based on the robust-max child policy. To select the final path, in each step, the robust-max child policy tries to select the child with both the highest visit count and the highest value. If there is
no robust-max child at any step, more simulations are run until a robust-max child is obtained [30].

V. CASE STUDY: IMAGE CONVOLUTION IN FASTFLOW

In order to evaluate our approach, we have used an image convolution [32] as our example parallel program. Image convolution is widely used in image processing applications, e.g., for blurring, smoothing and edge detection. Our version of image convolution consists of reading a stream of images into the memory and applying the convolution function (with a given filter) to each of these images. Applying the convolution function to an input image consists of calculating, for each pixel of the input image, a scalar product of the “window” surrounding that pixel with the filter weights, and storing the result in the output image at the same position:

\[
\text{out}(i, j) = \sum_{m} \sum_{n} \text{in}(i-n, j-m) \times \text{filter}(n, m),
\]

where \(\text{out}(i, j)\) is the pixel of the output image at position \((i, j)\), \(\text{in}(i, j)\) is the pixel of the input image at position \((i, j)\), \(m\) and \(n\) are the dimensions of the filter, and \(\text{filter}(i, j)\) is the pixel of the filter at position \((i, j)\).

We emphasise that our intention here is to demonstrate the applicability of our MCTS approach to deriving optimal static mapping for computationally-demanding real-world FastFlow applications, not to develop a new version of the CPU/GPU convolution algorithm. The application that we use is implemented in the GPU-extended version of FastFlow [29], where it is shown that the hand-tuned version of the application achieves a 45 speedup compared to the sequential version for large-scale inputs. Here, as a proof of concept, we demonstrate that our MCTS approach can generate a static mapping that delivers a comparable speedup, therefore opening up the prospect for automating tuning for non-experts.

We consider two different skeleton tree structures, or factorisations, for the image convolution application [29]. In both factorisations there are two kinds of components: \(r\), which reads a single image into memory, and \(p\) which applies the convolution function to a single image. \(r\) only has a CPU version, whereas \(p\) has both CPU and GPU versions. Therefore, there are two different \(p\) components, which we will denote by \(p_{CPU}\) and \(p_{GPU}\). The two factorisations that we consider are:

1) \(\text{Pipe} (\text{Farm}(r), \text{Seq}(p_{GPU}))\) and
2) \(\text{Pipe} (\text{Farm}(r), \text{Farm}(p_{CPU}, p_{GPU}))\).

Table I contains information about the hardware we used in the evaluation of the MCTS algorithm. Table II shows the two instances of the problem that we consider. We use the Factorisation 1 for large-scale problems where we cannot run more than one instance of a component on a single GPU. Since in our system we have only one GPU, it means that we can have at most one instance of \(p_{GPU}\). We use Factorisation 2 for small-scale problems where we can run more than one instance of a component on a single GPU.

Figure 3 compares the average absolute speedup obtained with a mapping calculated by the MCTS algorithm over 1000 executions with the hand-tuned version for both factorisations, compared with a sequential version. The speedups obtained with the mapping calculated by the MCTS algorithm are within 5% of the speedups of the hand-tuned version. Figure 4 shows the utilisation of the components of the application. We can observe that the utilisation of all components is above 0.6. Considering the fact that each component is assigned to a resource in the system this implies an overall 0.6 utilisation of the system. Figure 5 shows the throughput of the queues in the application. We can observe that the throughput of both queues is the same, which indicates perfect balance of the pipeline.

VI. ANALYSIS OF MCTS ON STATIC MAPPING PROBLEM

In this section we discuss in more detail the choice of MCTS for the static mapping problem.

The proposed approach aims to target FastFlow applications which are computationally heavy and contain a nested combination of farms and pipelines. Such applications usually take hours or even days to run [29], [33]. For example, running the application presented in Section V for an input stream of 8000 images on a machine with 2 GPUs and 24 CPUs takes around 4 hours. Therefore, the effort required to run the MCTS algorithm to obtain an optimal mapping, even for a relatively

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of CPUs</td>
<td>2</td>
</tr>
<tr>
<td>Cores per CPU</td>
<td>6</td>
</tr>
<tr>
<td>CPU Clock</td>
<td>3.07 GHz</td>
</tr>
<tr>
<td>Physical Memory</td>
<td>50 GB</td>
</tr>
<tr>
<td>No. of GPUs</td>
<td>1</td>
</tr>
<tr>
<td>GPU Model</td>
<td>NVIDIA Tesla M2090</td>
</tr>
<tr>
<td>GPU Memory</td>
<td>6 GB</td>
</tr>
<tr>
<td>GPU Cores</td>
<td>256</td>
</tr>
<tr>
<td>CUDA Version</td>
<td>4.0 V0.2.1221</td>
</tr>
<tr>
<td>GPU Driver Version</td>
<td>290.10</td>
</tr>
</tbody>
</table>

TABLE I. HARDWARE SPECIFICATION FOR RUNNING ACTUAL CONVOLUTION PROBLEM

<table>
<thead>
<tr>
<th>Category</th>
<th>Input Image Size</th>
<th>Filter Size</th>
<th>No. Input Images</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Scale</td>
<td>2048*2048</td>
<td>15*15</td>
<td>1000</td>
</tr>
<tr>
<td>Large Scale</td>
<td>8192*8192</td>
<td>48*48</td>
<td>1000</td>
</tr>
</tbody>
</table>

TABLE II. PROBLEM CATEGORIES

![Fig. 3. Speedups obtained with a static mapping calculated by the MCTS algorithm and with a hand-tuned version.](image-url)
simple problem like convolution, is amply justified in terms of the savings in time and energy realised by achieving speedups over such long run times.

The solution space is also sufficiently complex and expensive to justify an advanced search approach. The size of the solution space depends on the number of components of the application and the amount of resources available in the target architecture. For example, consider the skeleton tree structure in Figure 1, presented in Section IV, which can be categorised as a small problem. See Table III for the sizes of the solution space for this skeleton tree structure on various architectures.

The most expensive operation in the MCTS algorithm is the evaluation of the “quality” (i.e. throughput and utilisation) of a solution. See the fourth column of the table for the time it takes to evaluate all of the solutions in each case. Larger-scale architectures can have orders of magnitude more CPUs and GPUs, so for these architectures evaluating all of the possible solution space becomes practically impossible; therefore, we need a method for an intelligent search to determine good solutions.

In order to analyse the solution space we study the fitness-distance correlation (FDC) and autocorrelation (AC - a measure of convergence) for the above example over a platform with 16 cores CPU and 1 GPU. Exhaustive evaluation of the example indicates that the best solution is \{(a, 5), (b, 8), (g, 1), (c, 3)\}, where (x, y) denotes that y resources are allocated to the component x. The FDC and AC values for the solution space are calculated by running 50-step random walks from the global optimum for 100 times (see Figure 6).

Table V, displays the FDC obtained for this problem. This indicates a clear gradient in solution quality as distance increased, showing that it is reasonable to apply heuristic search. [34]. The AC value is comparable with the reported AC on the NK landscape problem [35] with k=50 and N=100 which is quite rugged. This counter-indicates the use of simple heuristics such as hill-climbing which are likely to become trapped in local optima.

A. Selection Strategy

In the UCT approach applied here, the factor \(C_P\) can be considered as a greediness factor of the algorithm. Therefore, considering the ruggedness of the solution space, selecting too small a value for \(C_P\) can result in searching smaller regions of the solution space, with a higher risk of becoming trapped in local optima. However, choosing too large a value for \(C_P\) slows or even prevents convergence. In our experience values of \(C_P\) around the average throughput gives good convergence to the optimal solution.

B. Solution Space Searched

As shown in Table IV all of the optimum results were found by exploring 9 – 32% of the solution space. We expect that, on larger problems with deeper trees, these percentages will drop dramatically as the number of unevaluated paths will
increase exponentially. Thus the justification for MCTS grows with search space size.

C. Back-propagation Policy

Looking at the back-propagation policies that we considered, we can observe that the \textit{Average} policy (MCTS-AVG) works better than the \textit{Max} (MCTS-MAX) in our case, due to it being less greedy. Figures 7 and 8 show the run length distribution for our example on two different architectures. We can observe that the MCTS-AVG is more accurate than the MCTS-MAX for larger problems. Figure 8 shows that the MCTS-MAX converges to a local optimum for 20% of runs. When the solution space is small, the probability of the MCTS-MAX converging to a global optimum is increased (Figure 7).

Moreover, the convergence of the MCTS-AVG is more predictable than the MCTS-MAX (as we can see in Figures 7 and 8). For the MCTS-AVG, 4 out of 5 times the convergence is in a certain iteration, while in MCTS-MAX this is less predictable. However, the number of solutions seen by the MCTS-MAX, is less than that for the MCTS-AVG. Since the evaluation of the solutions is expensive, the MCTS-MAX is computationally less expensive than the MCTS-AVG.

Another advantage of the MCTS-AVG over the MCTS-MAX is that the solutions it finds are more robust, i.e., at each decision step, the number of good solutions found in the selected subtree using the MCTS-AVG is larger than using the MCTS-MAX. This is potentially important if we need to apply dynamic remapping during the runtime of an application, in order to address the dynamic runtime factors that can influence the application’s performance. Fine-tuning the solution found using the MCTS-AVG is more likely to produce another good solution than for the MCTS-MAX. MCTS-MAX is less likely than MCTS-AVG to produce robust subtrees, since its evaluation is based solely on the best result of its children.

Finally, comparing the run length distributions using different versions of the MCTS algorithms with those of a simple random search, demonstrates the huge benefit of learning in finding high quality solutions with a non-exhaustive search.

VII. CONCLUSION

In this paper, we presented a novel approach for finding optimal static mappings of parallel applications executed on heterogeneous CPU/GPU architectures. Our approach was demonstrated using a FastFlow skeleton library for C++. The approach uses the Monte Carlo Tree Search algorithm (MCTS), and provides the application with static mappings that are comparable with hand-tuned solutions. We demonstrated that by using our MCTS approach, we were able to easily obtain speedups that are within 5% of the best possible speedups for our example application. The huge advantage of our approach is that it does not require in-depth specialist knowledge of either the parallel application or the underlying parallel architecture. This is an enormous saving in effort, and in most cases is simply unachievable for the average programmer.

We defined the problem of finding the optimal static mapping for a given parallel application, and we have studied the applicability of a MCTS approach to this problem, in terms of its suitability, scalability and solution quality. We have shown that the solution space for the problem is too large for exhaustive search techniques, making it suitable for the MCTS approach. Furthermore, we analysed two different versions of the MCTS algorithm: the version that uses the \textit{average} back-propagation policy (MCTS-AVG) and the version that uses the \textit{maximal} back-propagation policy (MCTS-MAX). We have shown that the MCTS-AVG provides more accurate, more predictable and more robust solutions than the MCTS-MAX.

As a future work, we plan to consider more skeletons (e.g. map skeleton for data parallelism). We also plan to demonstrate the generality of our approach using a wider range of applications and skeletal frameworks, including e.g. SkePU [36] for C++ and CUDA/OpenCL, SkeCL [37] for C++ and OpenCL and consider other programming paradigms, such as Erlang [38]. We also plan to investigate the applicability of our approach to distributed heterogeneous architectures, where the cost of transferring the data over a network has to be taken into account.

REFERENCES


