Bridging the Divide

A New Methodology for Semi-Automatic Programming of Heterogeneous Parallel Machines

Vladimir Janjic Kevin Hammond
School of Computer Science,
University of St Andrews, St Andrews, UK
{vj32,kh8}@st-andrews.ac.uk

Mehdi Goli John McCall
School of Computer Science,
Robert Gordon University, Aberdeen, UK
{m.goli, j.mccall}@rgu.ac.uk

Kamran Idrees Colin Glass Mhd. Amer Wafai
High Performance Computing Centre
Stuttgart (HLRS)
Stuttgart, Germany
{idrees, glass, wafai}@hlrs.de

Abstract

This paper presents a new programming methodology for introducing and tuning parallelism for heterogeneous shared-memory systems (comprising a mixture of CPUs and GPUs), using a combination of algorithmic skeletons (such as farms and pipelines), Monte-Carlo tree search for deriving mappings of tasks to available hardware resources, and refactoring tool support for applying the patterns and mappings in an easy and effective way. Using our approach, we demonstrate easily obtainable, significant and scalable speedups on a number of case studies showing speedups of up to 41 over the sequential code on a 24-core machine with one GPU. We also demonstrate that the mappings the MCTS algorithm suggest are comparable to the best possible speedups that can be obtained.

Keywords Heterogeneous Parallel Computing; Monte-Carlo Tree Search; Optimisations

1. Introduction

Heterogeneous multicore systems are increasingly common. Programming such systems remains difficult, however, since common programming techniques, such as OpenCL or CUDA+OpenMP, are very low level and require the programmer to make non-trivial scheduling and data-transfer decisions. Moreover, applications generally have many sources of parallelism: deciding which of the possible parallel structures should be exploited is especially challenging on heterogenous architectures. In this paper, we introduce a new methodology for programming heterogeneous parallel systems that: *i)* automatically discovers which parallel structure to exploit; *ii)* computes a near-optimal mapping of work onto the various heterogeneous processing elements; and, *iii)* provides a semi-automatic way of introducing the chosen parallel structure into the original program, and instantiating this with the derived mapping information. Our methodology is based on a combination of algorithmic skeletons [12] for defining the parallel structure, a method of finding a mapping for tasks on heterogeneous architectures[1], and refactoring tool support for user-guided introduction of the skeletons and mapping decisions.

We show the generality of our methodology by using realistic use-cases from three different domains (image processing, heuristic optimisation and molecular dynamics), programmed using the FastFlow [4] skeleton library for C++, which uses OpenCL and CUDA for GPU computations. The principles and techniques that we describe are not limited to C++ or FastFlow, however, but are *completely generic*, and can easily be transferred to other languages and paradigms. The paper makes the following research contributions:

1. we introduce a new methodology for building heterogeneous parallel programs semi-automatically, based on refactoring and algorithmic skeletons;
2. we introduce a mechanism for discovering efficient mappings of parallel application threads to heterogeneous CPU and GPU hardware, based on Monte-Carlo Tree Search simulations; and,
3. we show that, using our methodology, it is possible to derive a parallel structure and the corresponding mapping information, achieving performance that can be within 5% of the best-obtained manual parallelisation.

2. Background

2.1 Skeletons

In this paper we take a pattern-based approach, in which the parallel application is developed by composing and/or nesting *algorithmic skeletons*. An *algorithmic skeleton* [12] is an abstract computational entity that models some common pattern of computation. A skeleton is typically implemented as a higher-order function that abstracts over low-level details such as thread creation, communication, synchronisation, load balancing, etc. We consider two categories of skeletons: *sequential* skeletons, abstracting the structure of a purely sequential computation with no added parallelism; and, *parallel* skeletons, which implement specific parallel patterns. In our skeleton definitions, we assume that all of the input tasks are independent. We consider two *sequential* skeletons:

[1] Here we use Monte Carlo Tree Search (MCTS) [8].

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

- The *Compose ($\circ$)* skeleton represents sequential function composition applied to a sequence of inputs, where $f1 \circ f2$ denotes a sequential composition of two functions, $f1$ and $f2$.
- The *Order (;)* skeleton represents the execution of two functions on a sequence of inputs, where the execution of the first function needs to be completed for all input values before the execution of the second one can start. $f; g$, therefore, requires synchronisation between f and g.

We also consider two widely-used parallel skeletons:

- The *Pipeline ($\|$)* skeleton applies the composition of the functions $f_1, f_2, \ldots, f_n$, *in parallel* to a sequence of independent inputs $x_1, x_2, \ldots, x_m$, where the output of f_i is the input to f_{i+1}. Parallelism arises from the fact that $f_i(x_j)$ can be computed in parallel with $f_{i+1}(f_i(x_{j-1}))$. In the implementation that we consider, a separate thread is assigned to each *pipeline stage* (function f_i). We denote the pipeline skeleton by $(f_1 \parallel f_2 \parallel \cdots \parallel f_n)(x)$.
- A *Farm (Δ)* skeleton, $\Delta(nwCPU, nwGPU, f, x)$, represents the application of a single function, f, to the sequence of independent inputs, $x_1, x_2, x_3, \ldots, x_n$, in parallel. In the farm implementation that we consider, a specific number of *worker threads* is created, and the inputs are assigned to these worker threads in a round-robin fashion. Here *nwCPU/nwGPU* are, respectively, the number of worker threads executed on CPUs/GPUs.

We also allow nested skeletons. It is, therefore, possible to, for example, nest a pipeline inside a farm, $\Delta(nwCPU, nwGPU, f1 \parallel f2, x)$. A *skeletal configuration* abstracts over the skeleton parameters (e.g. the number and type of workers in a farm), thus focusing only on the nesting structure of the skeletons. In a skeletal configuration, we denote $\Delta(nwCPU, nwGPU, f, x)$ simply by $\Delta(f)$, and $(f_1 \parallel f_2 \parallel \cdots \parallel f_n)(x)$ by $f_1 \parallel f_2 \cdots \parallel f_n$. For example, the skeletal configuration $\Delta(f) \parallel (g \circ \Delta(h))$ denotes a pipeline of two stages, i) a farm whose worker function is f, and ii) a sequential composition of function g with a farm whose worker function is h.

2.2 Refactoring

Refactoring is the process of changing the structure of a program while preserving its functional semantics in order, for example, to increase code quality, programming productivity and code reuse. The term *refactoring* was first introduced by Opdyke in his PhD thesis in 1992 [21], and the concept goes back at least to the fold/unfold system proposed by Burstall and Darlington in 1977 [10]. Refactoring is a *semi-automatic* approach that is much more general than fully automated parallelisation techniques, which typically only work for a very limited range of cases under limited conditions. Additionally, unlike simple loop parallelisation, refactoring is applicable to a much wider range of possible parallel structures, since the parallelism is introduced in a controlled way via skeletons. In this paper we make use of a recent refactoring extension [6] for Eclipse, that introduces and tunes parallelism in C++ by introducing a nesting of skeletons into the application semi-automatically by user-guidance.

3. A Heterogeneous Parallel Programming Methodology

In this section we introduce a new parallel programming methodology aimed at increasing the programmability of heterogeneous parallel systems. Our parallel programming methodology aims to support both the inexperienced parallel programmer with little knowledge on parallel programming techniques; and also the experienced parallel programmer, who seeks to maximize productivity with the appropriate tool support to automate the process. Our general methodology is shown in Figure 1 and comprises a number of steps, described below.

1. *Identifying initial structure.* The programmer starts with a (possibly parallel) application. The first step is to *identify* an initial skeleton structure in the code corresponding to the skeletons defined in Section 2.1. This skeleton structure is recorded in a text file, which encapsulates the basic sequential structure of the algorithm, together with its basic units of computation (components) and tasks. Components correspond to functions of the source code. We also record what implementations (CPU, GPU or both) exist for which components.

   As a simple example, consider the following piece of code:

   ```
   for (i=0; i<nrImages; i++) {
     image = read_image(imageFiles[i]);
     outImage[i] = process_image(image);
   }
   ```

   The structure of this code is a composition of two functions, read_image and process_image, on a stream of input files, imageFiles. Components are the functions read_image and process_image, and the tasks are applications of these functions to the elements of the array, imageFiles. We might only have a CPU implementation of the read_image function, and both CPU and GPU implementations (kernels) of the process_image function. Using the notation from Section 2.1, we can denote this by $r \circ p$, where r is read_image function, p is process_image function, and $\circ$ is the sequential composition.
2. *Profiling*. After we have identified the skeleton structure of the application and its components, we do time profiling of the components. That is, we run each available version (CPU or GPU) of each component on a sample of input tasks in order to determine the average time it takes for each component to process one input task. This timing information is used in subsequent steps of our methdology. This step is carried out manually by the programmer, and its time complexity depends on the runtime of components for the sampled inputs.

   In our working example, using profiling we can obtain information that running CPU version of read_image on one image takes $0.2ms$, running CPU version of process_image on one image takes $6.6ms$ and running GPU version of process_image on one image takes $0.08ms$.
3. *Enumerating Skeleton Configurations*. Given the text file with the identified skeleton from the original application, produced in step 1, all possible equivalent skeleton configurations are automatically generated (up to a given depth of nesting) resulting in a number of different possible parallelisations. Given an initial configuration, each composition ($\circ$) can be transformed into a parallel pipeline ($\|$) and a farm skeleton (Δ) can be introduced for any skeleton configuration. Similarly the inverse of these transformations can also be applied; for example, we can transform a parallel pipeline into a sequential composition, or eliminate a farm skeleton altogether. This step is computationally very cheap and fully automatic.

   In our example, in the step 1 we identified the initial structure to be $r \circ p$; therefore, the possible skeleton configurations are $r \circ p$, $\Delta(r \circ p)$, $\Delta(r) \circ p$, $r \circ \Delta(p)$, $\Delta(r) \circ \Delta(p)$, $r \parallel p$, $\Delta(r) \parallel p$ etc.
4. *Filtering Using Cost Model*. Using profiling information obtained in step 2, the skeleton configurations are *filtered* using a cost model, if desired, to restrict the number of possibilities that need to be considered. This allows us to eliminate possible par-

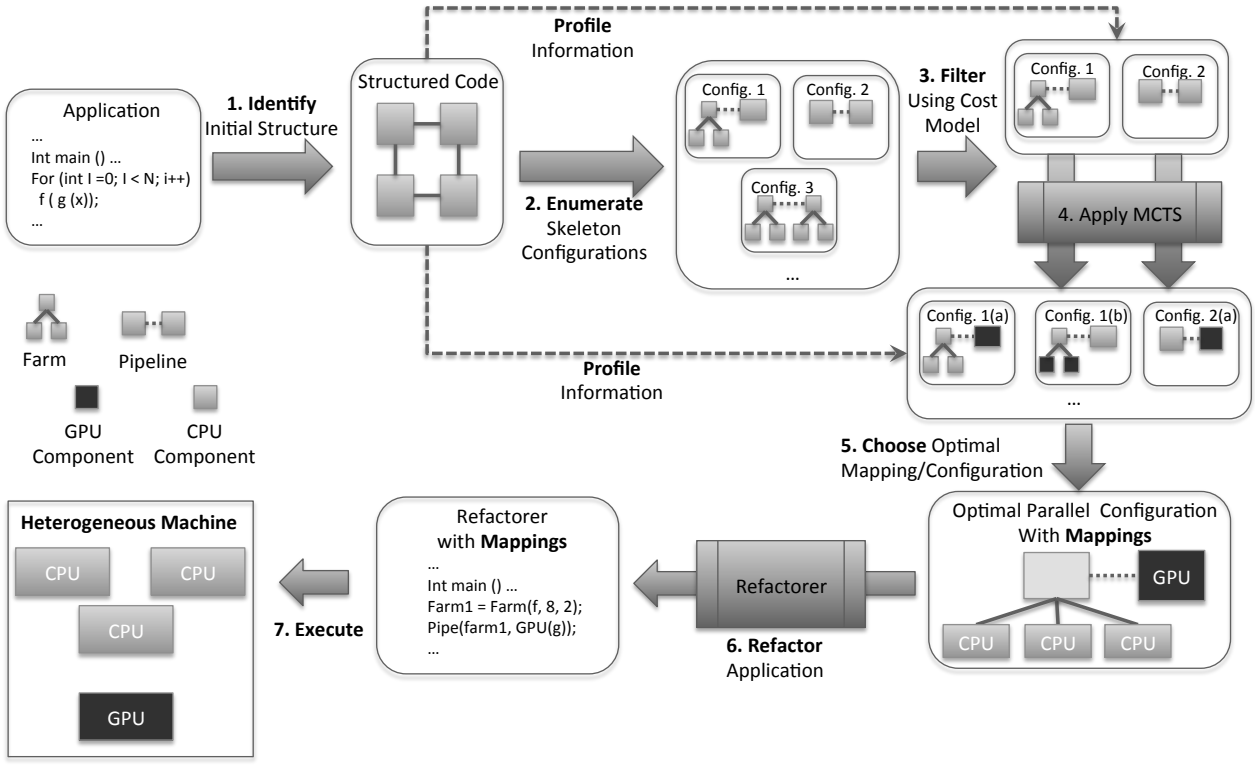

Figure 1. Overview of our methodology

allelisations with little or no potential speedup at an early stage of development. In Section 5, we use a simple high-level cost model to predict the best possible run times for each configuration on a given hardware. At this stage, exact timing information is not needed, as only very poor potential speedups lead to exclusion. Since we use relatively simple cost models, this step is computationally very cheap, and also fully automatic.

In our example, the cost model may predict that $\Delta(r) \parallel \Delta(p)$, $\Delta(r) \parallel p$ and $\Delta(r) \circ \Delta(p)$ are the best candidates from all possible factorisations.

5. *Ranking the Configurations and Deriving Mappings*. The remaining configurations are then analysed in more detail, deriving optimal (or near-optimal) static mappings for each of them, together with the estimated runtime. A static mapping is an assignment of number of workers for each farm skeleton in a skeleton configuration, together with the type of each worker and each pipeline stage (the type can be CPU or GPU). Possible types of a farm worker/pipeline stage depend on the type of implementation that we have for that kind of worker/pipeline stage. This phase, therefore, outputs for each configuration, all the missing skeleton parameters. It also gives the ranking of the configurations in terms of their expected performance. In this paper, we present one possible model for deriving static mappings for a given skeleton configuration, based on the Monte Carlo Tree Search (MCTS) algorithm [8]. This step is fully automatic, and is also computationally the most expensive part of the methodology. Exactly how much time it takes to rank the configurations and derive mappings depends mostly on the method used for estimating of the application runtime with a particular static mapping. If full simulation is used, the cost is very high, whereas if some analytical model is used (e.g. some more precise cost model than in step 4), the cost can be very low.

   In our example, this step can tell us that the best parallelisation on a given machine (e.g. comprising of 24 CPU cores and 1 GPU) is $\Delta(r) \parallel \Delta(p)$, where 15 CPU workers are used for $\Delta(r)$ and 9 CPU and 1 GPU workers are used for $\Delta(p)$.

6. *Refacotring the Application*. The programmer then *chooses* one of the parallelisations together with its static mapping. and refactors the original application from Step 1, introducing the desired skeleton configuration from Step 5 using the *refactoring* tool. The refactoring tool performs all the required program transformations and condition checking automatically, introducing the skeleton structure and the parameters from Step 4. This part is semi-automatic and computationally cheap.

   Considering the example code from Step 1 and the skeleton configuration, $\Delta(r) \parallel \Delta(p)$, the refactoring tool may produce the following:

```
ff_farm<> readFarm;
...
for(int i = 0 ; i< nworker1; i++)
    readFarm.push_back(&read_image)
for(int i = 0 ; i< gpu_nworker1; i++)
    readFarm.push_back(&read_image_gpu)
ff_farm<> processFarm;
...
for(int i = 0 ; i< nworker2; i++)
```

```
    processFarm.push_back(&read_image)
ff_pipeline pipe;
pipe.add_stage(&readFarm);
pipe.add_stage(&processFarm);
....
```

where the refactoring tool introduces FastFlow farm and pipeline skeletons (ff_farm and ff_pipeline) including the number of CPU and GPU workers for the farm skeletons, readFarm and processFarm. These worker parameters are taken directly from the output of Stage 4.

7. *Executing the Application*. The refactored program can then be executed on the available heterogeneous hardware, and the process can be repeated if necessary. For example, if the programmer decides to port the application to a different architecture, or if after executing the program, the programmer discovers that an alternative configuration given at Step 5 would be better suited.

4. Deriving Mappings using Monte Carlo Tree Search

In this section, we describe a model that we use to derive, for a given skeletal configuration, a good static mapping of its components to the available hardware. A static mapping in our case corresponds to a particular choice of values for the parameters of skeletons, i.e. the number of workers in each farm, the type (CPU or GPU) of each worker in each farm and each stage of each pipeline. The quality of a mapping is derived from a specific evaluation function Q, being a combination of the runtime and the resource utilisation.

Our model accepts as an input a skeletal configuration and the timings for its components (derived from profiling both for CPU and GPU versions, if the GPU version of a component is available). As an output, it produces a candidate static mapping and the corresponding estimated runtime of the skeletal configuration. Since considering all possible static mappings for a given skeletal configuration may be computationally intractable, an optimisation method is used. Here, we use the Monte Carlo Tree Search (MCTS) approach, well known for generating and evaluating large game trees in Game theory. In our case, the nodes of the generated tree correspond to partial mappings (with some of the skeleton parameters chosen) and the leaves of the tree correspond to complete mappings. The children of a node represent different possible assignments of a yet unassigned skeleton parameter.

The MCTS approach starts from a tree that consists only of a single root node (i.e. a static mapping where no parameters are chosen). It proceeds by repeating the following three steps:

1. *Expansion step* – A node (corresponding to a partial static mapping) is selected, and one of its children is added to the tree. This is equivalent to assigning a value to one previously unassigned parameter;
2. *Selection step* – Starting from the newly added node, a complete static mapping is generated by randomly assigning the remaining unassigned parameters. The resulting static mapping is evaluated based on the evaluation function, Q, yielding the valuation v;
3. *Propagation step* – The valuation, v, is propagated back to the node added in step 1.

Steps 2 and 3 are repeated until a reliable statistical evaluation of step 1 is attained. Then, step 1 is repeated, adding a new value to the partial mapping. Finally, the overall best complete mapping (a leaf of the tree) is selected.

The evaluation function that we use to evaluate how good static mappings are is based on the estimation of the runtime for that static mapping that we obtain using simulations, and the utilisation of the system. The function is

$$Q(M) = S(M) - (\delta_U(M) + \delta_Q(M)),$$

where $S(M)$ is the estimated throughput of the whole system (i.e. the number of tasks per unit of time that get processed), $\delta_U(M)$ is the standard deviation of the utilisation of components (where the utilisation of a component is the ratio between the time the component spends executing tasks and the total execution time of the application) and $\delta_Q(M)$ is the standard deviation of the utilisation of the connecting queues between the components (where the utilisation of a queue is the ratio between the time where at least one task was in the queue and the total execution time) in the skeleton. Using this function, $Q(M)$, rather than using just the throughput, $S(M)$, as an evaluation function, discourages the allocating of more resources to the skeleton configuration, if it only results in marginally improved runtime, which may be important in settings where resources are paid for (e.g. clouds).

5. Case Studies

In this section we demonstrate our methodology on three realistic case studies. For each application, we show different steps of its parallelisation:

1. starting from a sequential version, we show a number of different possible skeleton configurations;
2. if the number of skeleton configurations is large, we pre-filter these configurations using a cost model described in [7] to eliminate weak configurations (i.e. those that would only give small speedups);
3. we apply MCTS to the remaining configurations to discover the estimated optimal static mappings for each of them, and to find out which configuration (with its corresponding static mapping) delivers the best speedup;
4. finally, we evaluate the static mappings for each skeleton configuration resulting from Step 3, in order to verify the accuracy of the result returned by MCTS.

We consider applications that belong to different domains, showing the generality of our parallelisation methodology. The applications we consider are Image Convolution, Ant Colony Optimisation and Molecular Dynamics. The evaluations of the skeleton configurations in Step 4 are performed on a machine comprising 2x2.4Ghz 12-core AMD Opteron 6176 CPUs, coupled with an NVidia Tesla C2050 graphic card with 448 CUDA cores running at 1.16GHz, running CentOS Linux and g++ 4.1.2. The speedups reported in the figures are averages over 5 independent runs.

5.1 Image Convolution

Image convolution is a technique widely used in image processing applications such as blurring, smoothing or edge detection. The basic structure of the convolution algorithm is a two-stage function composition, $r \circ p$. r is a stage that reads in an image from a file and p is a stage that processes the image by applying a filter. This convolution process is typically applied to a stream of input images, where the output is also a stream. For each pixel of the input image, the filtering stage consists of computing the scalar product of the filter and the window surrounding the pixel:

$$output\_pixel(i,j) = \sum_m \sum_n input\_pixel(i-n, j-m) \times filter\_weight(n,m) \quad (1)$$

Configuration	Est. runtime
$r \circ p$	5.60
$r \parallel p$	3.88
$\mathbf{\Delta(r) \parallel p}$	**1.60**
$r \parallel \Delta(p)$	4.00
$\mathbf{\Delta(r) \parallel \Delta(p)}$	**0.40**
$\mathbf{\Delta(r \parallel p)}$	**0.56**
$\Delta(r \circ p)$	5.60
$\Delta(r) \circ \Delta(p)$	2.00
$\Delta(r) \circ p$	2.00
$r \circ \Delta(p)$	5.60

Figure 2. Skeleton configurations and their cost-predicted runtimes for the Image Convolution

	$\Delta(r) \parallel \Delta(p)$	$\Delta(r) \parallel p$	$\Delta(r \parallel p)$
Mapping (C,G)	$(6,0) \parallel (0,3)$	$(4,0) \parallel (0,1)$	$(5,5)$

Figure 3. MCTS predicted optimal mappings for three configurations of the Image Convolution example. (C, G) denotes the number of CPU and GPU workers for a farm.

5.1.1 Configurations and Cost-Model Filtering

Figure 2 shows all possible skeleton configurations for the image convolution, up to a nesting depth of two. The first column shows the skeleton configuration, using the notation introduced in 2.1, and the second column shows the cost-estimated minimal runtime for that configuration. The minimal runtime is taken over all possible combinations of workers in each skeleton farm. Using profiling, we obtained sequential timings for functions r and p on one $4096 * 4096$ image, where $T(r_{CPU}) = 0.2ms$, $T(p_{CPU}) = 6.6ms$, $T(p_{GPU}) = 0.08s$. In Figure 2, the bold results are the three best configurations we have selected for further processing using the MCTS model.

5.1.2 Optimal Static Mappings Determined by MCTS

Figure 3 shows the output of MCTS for the three best skeleton configurations for image convolution. The figure shows, for each farm in each configuration, the estimated optimal number of CPU and GPU workers, denoted by a pair (C, G) where C is the number of CPU workers and G is the number of GPU workers.

5.1.3 Evaluation of Skeleton Configurations

All experiments are on a stream of 25 4096*4096 images. Figure 4 shows the actual speedups obtained for $\Delta(r) \parallel p$ skeleton configuration. For this configuration, the first stage of the pipeline is a farm of workers executing r (for which only a CPU implementation exists), and the second stage is a single worker executing p. Since p is much faster when executed on a GPU, we only consider mappings where the second pipeline stage is mapped to one GPU worker. The figure shows the speedups with a different number of CPU workers in the farm of the first pipeline stage. MCTS predicted the best speedup when 4 CPU workers are used for this stage. As Figure 4 shows, this mapping gives an actual speedup of 39.14. Compared to the best speedup of 39.43 when 8 CPU workers are used in the first pipeline stage. The speedup obtained with the predicted mapping is within 1% of the best speedup obtainable. The difference in speedup is 0.29, however, the mapping with maximum speedup also uses more resources, resulting in lower hardware utilisation.

In Figure 5 we show the speedups for $\Delta(r) \parallel \Delta(p)$ skeleton configuration. The x axis shows the number of CPU workers for

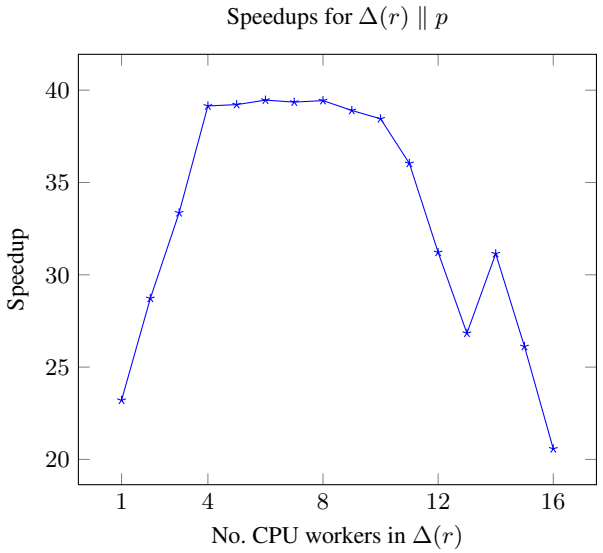

Figure 4. Speedup graph for the Image Convolution configuration $\Delta(r) \parallel p$, where p is executed on a GPU.

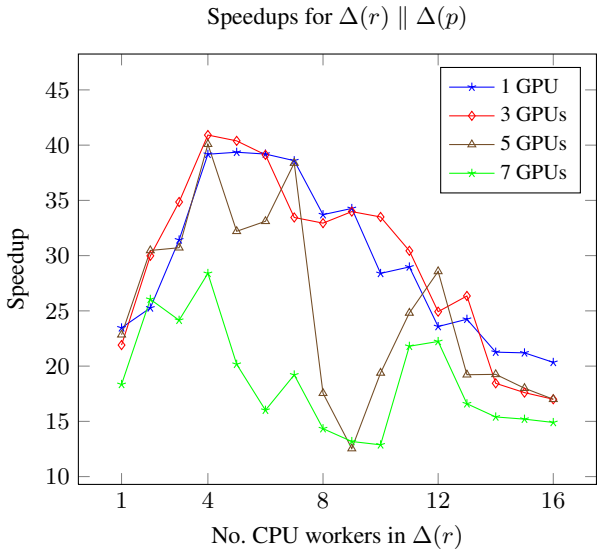

Figure 5. Speedup figures for the Image Convolution configuration $\Delta(r) \parallel \Delta(p)$, with 0 CPU and a different number of GPU workers for $\Delta(p)$.

$\Delta(r)$, whereas each line on the graph corresponds to a fixed number of GPU workers in $\Delta(p)$, with the number of CPU workers in $\Delta(p)$ being 0; this corresponds to the best speedups obtained for this configuration. For this configuration, the MCTS predicts the optimal speedup for 6 CPU workers for $\Delta(r)$ and $(0,3)$ CPU and GPU workers for $\Delta(p)$. Figure 5 shows a speedup of 39.12 for this mapping. The best overall speedup is 40.91, for 4 CPU workers in $\Delta(r)$ and $(0,3)$ CPU and GPU workers for $\Delta(p)$. Therefore, the speedup obtained using the MCTS predicted mapping is within 4% of the best speedup obtained.

Finally, Figure 6 shows the speedups for the skeleton configuration, $\Delta(r \parallel p)$. The best speedups for this configuration were

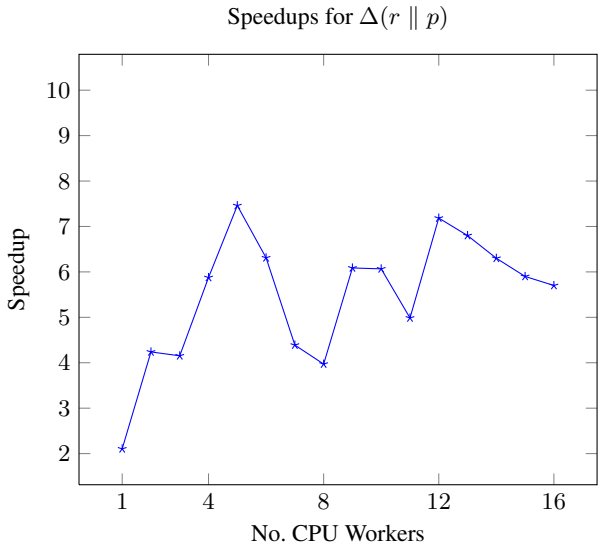

Figure 6. Speedup figures for the Image Convolution configuration $\Delta(r \parallel p)$, where the number of GPU workers and the number of CPU workers for $\Delta(r \parallel p)$ are equal.

obtained when the number of CPU and GPU workers are equal for $\Delta(r \parallel p)$. As Figure 6 demonstrates, the best speedup obtained for this configuration is 7.45 for $(5, 5)$ CPU and GPU workers for $\Delta(r \parallel p)$, confirming the prediction given by MCTS (Figure 3).

5.2 Ant Colony Optimisation

Ant Colony Optimisation (ACO) [9] is a heuristic for solving NP-complete optimisation problems, inspired by the behaviour of real ants. In this paper, we apply ACO to the Single Machine Total Weighted Tardiness Problem (SMTWTP) optimisation problem, where we are given n jobs and each job, i, is characterised by its processing time, p_i, deadline, d_i, and weight, w_i. The goal is to schedule the execution of jobs in a way that achieves minimal total weighted *tardiness*, where the tardiness of a job is defined by $T_I = \max\{0, C_i - d_i\}$ (with C_i being the completion time of the job, i) and the total tardiness of the schedule is defined as $\sum w_i T_i$. The ACO solution to the SMTWTP problem consists of a number of iterations, where in each iteration each ant independently computes a schedule, and is biased by a *pheromone trail*. The pheromone trail is stronger along previously successful routes and is defined by a matrix, τ, where $\tau[i, j]$ is the preference of assigning job j to the i-th place in the schedule. After all ants compute their solution, the best solution is chosen as the 'running best'; the pheromone trail is updated accordingly, and the next iteration is started.

5.2.1 Configurations and Cost-Model Filtering

The basic structure of one iteration of the algorithm is $s; p; u$, where s is the phase which finds the solutions for all ants, p the phase which picks up the best solution and u the phase where the pheromone trail is updated, taking into account the current best solution. Sequential ordering of the phases prevents introducing a pipeline between any two stages. Also, the phase p cannot be parallelised using a farm, so we are left with introducing a farm for s and/or u. Cost-model filtering, however, showed that introducing the farm for u is not viable, so we will consider only the configuration where a farm is introduced for s, giving a skeleton configuration, $\Delta(s); p; u$. For s, we have both CPU and GPU implementations.

	$\Delta(s); p; u$
Mapping (C,G)	$(9, 5)$

Figure 7. MCTS predicted optimal mappings for the $\Delta(s); p; u$ configuration for the ACO example. (C, G) denotes the number of CPU and GPU workers for a farm.

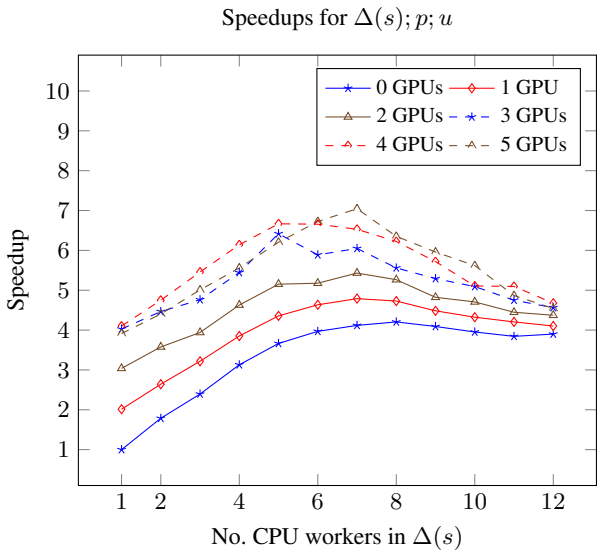

Figure 8. Speedup graph for the ACO configuration $\Delta(s); p; u$

5.2.2 Optimal Static Mapping Determined by MCTS

Figure 7 shows the output of MCTS for the $\Delta(s); p; u$ configuration for the ACO example.

5.2.3 Evaluation of Skeleton Configurations

Figure 8 shows speedups for the $\Delta(s); p; u$ configuration. Each line shows the speedups with a fixed number of GPU workers and varying number of CPU workers for $\Delta(s)$. From the figure, we can observe that the best speedup of 7.04 is obtained with $(7, 5)$ CPU and GPU workers. The MCTS model predicted the best speedups for $(9, 5)$ CPU and GPU workers, and for this mapping we obtained the speedup of 5.95. Therefore, the mapping returned by the MCTS model (shown in Figure 7) gives the speedup that is within 15% of the best obtained. In the figure, we have omitted the speedups when more than 12 CPU workers are used for $\Delta(s)$, as (due to the NUMA architecture and the fact that our version of ACO is very data-intensive) these speedups are smaller than when fewer CPU workers are used.

5.3 Molecular Dynamics

Molecular Dynamics (MD) simulation computes a system of N particles on the atomic level [1]. Once the system is initialised, the interactions between the molecules are evaluated explicitly, allowing for the numerical integration of Newton's equations of motion. The molecular trajectories in time yield the thermodynamic properties of the system.

The molecular simulation code used here (CMD) is designed for basic research into HPC MD. In the BasicN2 variant investigated in this paper, all intermolecular distances are evaluated in order to identify interaction partners. However, a special flavour

	$\Delta(r \circ h)$
Mapping (CPU, GPU)	$(22, 1)$

Figure 9. MCTS predicted optimal mapping for Molecular Dymamics example with $\Delta(r \circ h)$ configuration. (C, G) denotes the number of CPU and GPU workers for a farm.

of BasicN2 is used, where the domain is decomposed into subdomains of approximately 1000 molecules in order to counter the prohibitive scaling of neighbour search (otherwise O(N^2)). These subdomains are distributed among FastFlow CPU and GPU workers. As inferred from profiling data, the force calculation routine dominates the simulation time and is therefore parallelised. The force calculation itself is decomposed into two kernels, intra-domain and inter-domain (with the use of halos) interactions.

5.3.1 Configurations and Cost-Model Filtering

r denotes intra-domain interactions, and h denotes inter-domain.

In CMD, the two units of computation r and h need to be applied to the set of input elements (molecules). Both are compute intensive and can be farmed ($\Delta(r)$ and $\Delta(h)$). There are three possible skeleton structures that can be configured:

1. r and h can be executed sequentially and farmed, $\Delta(r \circ h)$
2. r and h can be executed concurrently (different threads working on same input set of elements in both routines), $\Delta(r; h)$.
3. r and h can form a pipeline, where once r for ith element is computed, then h on same ith element can be computed. This makes a nested skeleton with pipeline of two farms, $\Delta(r) \parallel \Delta(h)$.

The best configuration as determined by the cost-predicted runtime is $\Delta(r \circ h)$. Therefore we have selected this configuration for further processing using MCTS. The key parameters here are: i) how much work to offload onto the GPU (GPU workers), as the CPU and the GPU can work on the farm concurrently; and, ii) how many CPU workers should be utilised.

5.3.2 Optimal Static Mapping Using MCTS

Figure 9 shows the output of the MCTS model applied to the best skeleton configuration. The figure shows the estimated optimal number of CPU and GPU workers for the $\Delta(r \circ h)$ configuration.

5.3.3 Evaluation of Skeleton Configurations

Figure 10 shows the speedups for a domain of 1000 molecules for the $\Delta(r \circ h)$ skeleton configuration. In the figure, the x axis corresponds to the number of CPU workers, and each line in the graph corresponds to a fixed number of GPU workers. In the figure, the best obtained speedup for this configuration is 23.43 for 22 CPU workers and 4 GPU workers. As Figure 9 illustrates, the predicted mapping is $(22, 1)$ (i.e., 22 CPU workers and 1 GPU worker). From Figure 10, we can see that the $(22, 1)$ mapping gives us a speedup of 20.65. The accuracy of the MCTS prediction for this configuration is therefore within 12% of the best possible speedup obtained.

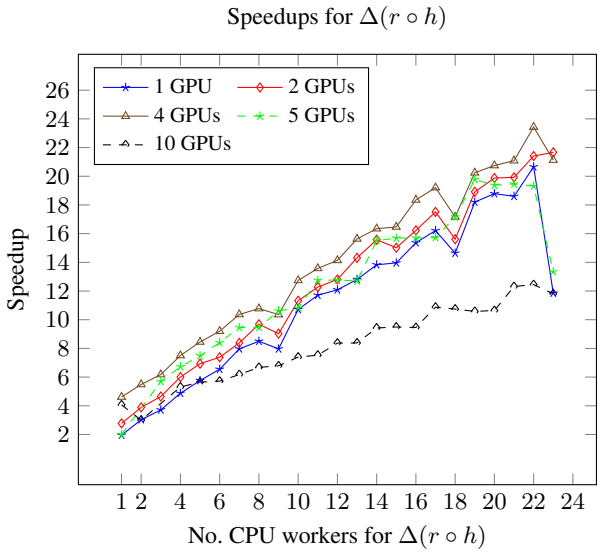

Figure 10. Speedup graph for the Molecular Dynamics configuration $\Delta(r \parallel p)$.

6. Related Work

Since the 1990s, the skeletons research community has been working on high-level languages and methods for parallel programming [12]. A rich set of skeleton rewriting rules has been proposed in [5, 24]. When using skeleton rewriting transformations, a set of functionally equivalent programs exploiting different kinds of parallelism is obtained. Cost models and evaluation methodologies have also been proposed that can be used to determine the best of a set of equivalent parallel programs [3, 24]. The methodology presented in this paper extends and builds on this and similar work by providing refactoring tool-support supplemented by a programming methodology that aims to make structured parallelism more accessible to a wider audience. There has so far been only a limited amount of work on refactoring for parallelism [17]. In [7], we introduced a parallel refactoring methodology for Erlang programs, demonstrating a refactoring tool that introduces and tunes parallelism for Skeletons in Erlang. In [6] we presented a refactoring methodology, demonstrating a refactoring tool support approach for introducing and tuning parallelism for C++ programs. However, unlike the methodology proposed in this paper, both of these methodologies did not support heterogeneous architectures, or provide support for deriving mapping information. The static mapping problem is by no means a new challenge and there is an extensive body of work on mapping task, data and pipeline parallelism to parallel architectures providing static partitioning [19, 23, 25], using runtime scheduling [22], heuristic-based mappings [15], analytical models [20]. Each of these can improve the performance of the system. 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 [27] or the runtime adaptation approach used in FlexStream [18] framework. 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. In [16] we introduced a new mapping technique for heterogeneous multicore systems, but unlike the approach here, did not provide a usable programming methodology. Most of the work on GPUs is primarily focused on application performance tuning [2] rather than orchestration. In [26] a method is provided to orchestrate the execution of heterogeneous StreamIt program presented on a multi core platform equipped with an accelerator. Monte Carlo Tree Search have classically been applied to challenging game playing, for example the GO and Bandit problem [14]. In this paper we establish the applicability of MCTS to

the seamless orchestration of heterogeneous components over a hybrid (CPU, GPU) platform.

7. Conclusions and Future Work

In this paper we introduced a new heterogenous parallel programming methodology that employs new refactoring and static mapping technology, and is based on algorithmic skeletons. Our case studies have shown that even for an algorithm with relatively little structure, there are many different potential parallelisations to choose from. The methodology presented here suggests promising candidates (skeletal configurations and corresponding static mappings) which are introduced automatically via the refactoring tools. This allows the programmer to concentrate on the correctness of the application, rather than the parallelisation.

Although our methodology is described in terms of C++ using FastFlow, the approach taken here is, in fact, completely generic. The refactorings and skeletons could easily be carried over to different languages and frameworks, such as Cilk, OpenMP, etc.

We have used the Monte Carlo Tree Search (MCTS) algorithm to predict good mappings of components of a parallel program to processing elements of heterogenous machines, which are within 5% - 15% of the best speedups that are obtainable. However, the method used to determine the best mapping efficiently is not limited to MCTS. Depending on the accuracy of the cost model predictions and on the dimensionality of the problem, other methods such as evolutionary algorithms will be considered in the future. Obviously, in order to attain the best speedup, the programmer can perform an exhaustive search over the parameter space, as we have shown in Section 5 to verify our MCTS predictions. This is compute-time intensive, but worth while if the application is to be executed frequently. Our methodology therefore supports tuning the invested computing time vs. the quality of the results, while the refactoring tool allows for straight-forward exploration of different skeletal configurations. It is our intention, in time, to develop a generic refactoring and mapping methodology capable of using a common set of refactoring rules and skeletons.

In the near future, we expect to extend our methodology to cover a wide range of parallel skeletons including parallel workpools, divide-and-conquer, map-reduce, bulk synchronous parallelism, and other domain-specific parallel patterns, such as parallel orbit enumerations. In addition, we intend to demonstrate the use of our methodology on a further set of case studies, showing greater skeleton nesting and heterogeneity. We also intend to broaden the scope of our methodology to include dynamic remapping. A remaining issue for further investigation is the applicability of this approach over a distributed heterogeneous architecture, where the cost of transferring data over the network is accounted for in the simulation.

Acknowledgements

This work has been supported by the European Union grants IST-2010-248828 “ADVANCE: Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering”, and IST-2011-288570 “ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems”, the BMBF project SkaSim and by the UK’s Engineering and Physical Sciences Research Council grant EP/G055181/1 “HPC-GAP: High Performance Computational Algebra.”

References

[1] Michael P Allen. Introduction to Molecular Dynamics Simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins*, 23:1–28, 2004.

[2] S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing Stream Programs Using Linear State Space Analysis. In *Proc. of the 2005 int. con. on Compilers, architectures and synthesis for embedded systems*, pp 126–136. ACM, 2005.

[3] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting Skeleton Programs: How to Evaluate the Data-Parallel Stream-Parallel Tradeoff. *CMPP*, 44–58. Germ., May 1998.

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating Code on Multi-cores with FastFlow. In *Euro-Par*, pp 170–181, 2011.

[5] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards Parallel Programming by Transformation: The FAN Skeleton Framework. *Parallel Algorithms and Applications*, 16(2–3):87–121, Mar. 2001.

[6] C. Brown, V. Janjic, K. Hammond, H. Schöner, K. Idrees and C. Glass Agricultral Reform: More Efficient Farming Using Advanced Parallel Refactoring Tools. *Submitted to PDP* 2014.

[7] C. Brown, K. Hammond, M. Danelutto, P. Kilpatrick, and A. Elliott. Cost-Directed Refactoring for Parallel Erlang Programs. in *Int. J. of Parallel Processing*. Springer. Sept. 2013.

[8] C. Browne. A Survey of Monte Carlo Tree Search Methods *IEEE Trans. on Computational Intelligence and AI in Games Vol.1 Iss. 2* March. 2012.

[9] M. den Besten, T. Stuetzle, M. Dorigo. Ant Colony Optimization for the Total Weighted Tardiness Problem *PPSN 6*, p611-620, Sept. 2000.

[10] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J. of the ACM*, 24(1):44–67, 1977.

[11] G. Chaslot, S. De Jong, J.-T. Saito, and J. Uiterwijk, Monte-carlo Tree Search in Production Management Problems, in *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, 2006, pp. 91–98.

[12] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[13] A. Couëtoux, J. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard. Continuous Upper Confidence Trees. *Learning and Intelligent Optimization*, pp 433–445, 2011.

[14] R. Coulom. Efficient Selectivity and Backup Operators in Monte-carlo Tree Search. *Computers and Games*, pp 72–83, 2007.

[15] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *ACM SIGOPS Operating Systems Review*, volume 40, pp 151–162. ACM, 2006.

[16] M. Goli, J. McCall, C. Brown, V. Janjic and K. Hammond Using Machine Learning to Derive Mappings for Heterogeneous Parallel Computations. *Proc. of IEEE Congress on Evol. Comp.* Cancun, Mexico. May. 2013.

[17] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. Gonzalez-Velez, P. Kilpatrick, R. Keller, T. Natschlager, and G. Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. FMCO. Feb. 2012.

[18] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In 18th Int. Con. *Parallel Architectures and Compilation Techniques*, pp 214–223. 2009.

[19] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[20] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval. Analytical Modeling of Pipeline Parallelism. In 18th Int. Con. *Parallel Architectures and Compilation Techniques.* , pages 281–290. 2009.

[21] W. Opdyke. Refactoring Object-Oriented Frameworks. PhD Thesis, Dept. of Comp Sci, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992).

[22] K Ramamritham. Dynamic Task Scheduling in Hard Real-time Dist. Systems. in J. of Software, IEEE 1(3), pp 65–75. 1984.

[23] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: Concurrent Programming for Modern Architectures. In *Proc. of the 12th ACM Sym. on Principles and Practice of Parallel Programming*, pages 271–271. 2007.

[24] D. B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, 1995.

[25] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross. Exploiting Task and Data Parallelism on a Multicomputer. *ACM SIGPLAN Notices*, 28(7):13–22, 1993.

[26] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software Pipelined Execution of Stream Programs on GPUs. In Int. Sym. *Code Generation and Optimization. I*, pp 200–209. 2009.

[27] Z. Wang and M. F. O'Boyle. Partitioning Streaming Parallelism for Multi-cores: a Machine Learning Based Approach. In *Proc. of the 19th int. con. on Parallel Architectures and Compilation Techniques*, pp 307–318. ACM, 2010.