Project no. 288570

PARAPHRASE

Strategic Research Partnership (STREP)
PARALLEL PATTERNS FOR ADAPTIVE HETEROGENEOUS MULTICORE SYSTEMS

Static Mapping Implementation
D3.2

Due date of deliverable: M24

Start date of project: October 1st, 2011

Type: Deliverable
WP number: WP3
Task number: T3.2

Responsible institution: UNIPI
Editor and editor’s address: M. Danelutto, UNIPI

Version 0.1 (Revision)

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
<td></td>
</tr>
<tr>
<td>PP Restricted to other programme participants (including the Commission Services)</td>
<td>√</td>
</tr>
<tr>
<td>RE Restricted to a group specified by the consortium (including the Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium (including the Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary

This deliverable describes a new technique for static mapping, i.e., assigning software components to heterogeneous hardware resources using a Monte-Carlo Tree Search. We introduce a new methodology for programming heterogeneous parallel systems, building on the work in WP4 on refactoring tools using static mappings derived using MCTS to gain near-optimal mappings. We also provide a number of use-cases, including an industrial use-case from HLRS, demonstrating that we are able to gain, with our methodology, near-optimal mappings. Finally we introduce PEI, a Performance Enhancement Infrastructure which is a collection of mechanisms and policies used for improving the performance of skeleton based frameworks for use with the static mapping.

Positioning of Deliverable D3.2

The positioning of this deliverable (D3.2) with respect to other deliverables is shown in Figure . In particular, the work presented in D3.2 is based upon the work in D3.1 (Hardware/Software Virtualisation Interfaces Report). In addition to this, we also build upon the work in WP2: D2.4 (Heterogeneous Implementation of Parallel Patterns and D2.1 (Homogeneous Implementation of Initial Generic Patterns). Finally we integrate work from WP6, D6.5 First Report on Experimental Evaluation.
Figure 1: Positioning deliverable D3.2 (with respect to the other WP3 deliverables and to the main relevant deliverables of the other WPs)
Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executive Summary</td>
<td>1</td>
</tr>
<tr>
<td>Positioning of Deliverable D3.2</td>
<td>1</td>
</tr>
<tr>
<td>Introduction</td>
<td>5</td>
</tr>
<tr>
<td>1.1 What is new in this deliverable?</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Skeletons</td>
<td>6</td>
</tr>
<tr>
<td>A Heterogeneous Parallel Programming Methodology</td>
<td>7</td>
</tr>
<tr>
<td>MCTS-based Static Mapping</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Monte Carlo Tree Search (MCTS) Based Model for Deriving Static Mappings</td>
<td>11</td>
</tr>
<tr>
<td>3.1.1 Converting the Skeleton Tree to the Process Graph</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2 A Monte Carlo Tree Search from the Process Graph</td>
<td>13</td>
</tr>
<tr>
<td>3.1.3 Selection Strategy</td>
<td>13</td>
</tr>
<tr>
<td>3.1.4 Simulation of Static Mapping</td>
<td>14</td>
</tr>
<tr>
<td>3.1.5 Reward Function</td>
<td>14</td>
</tr>
<tr>
<td>3.1.6 Back-propagation, Termination Condition and Final Move Selection</td>
<td>15</td>
</tr>
<tr>
<td>Case Studies</td>
<td>16</td>
</tr>
<tr>
<td>4.1 Image Convolution</td>
<td>16</td>
</tr>
<tr>
<td>4.1.1 Configurations and Cost-Model Filtering</td>
<td>17</td>
</tr>
<tr>
<td>4.1.2 Optimal Static Mappings Determined by MCTS</td>
<td>17</td>
</tr>
<tr>
<td>4.1.3 Evaluation of Skeleton Configurations</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Ant Colony Optimisation</td>
<td>20</td>
</tr>
<tr>
<td>4.2.1 Configurations and Cost-Model Filtering</td>
<td>20</td>
</tr>
<tr>
<td>4.2.2 Optimal Static Mapping Determined by MCTS</td>
<td>21</td>
</tr>
<tr>
<td>4.2.3 Evaluation of Skeleton Configurations</td>
<td>22</td>
</tr>
<tr>
<td>4.3 Molecular Dynamics</td>
<td>22</td>
</tr>
<tr>
<td>4.3.1 Configurations and Cost-Model Filtering</td>
<td>22</td>
</tr>
<tr>
<td>4.3.2 Optimal Static Mapping Using MCTS</td>
<td>23</td>
</tr>
<tr>
<td>4.3.3 Evaluation of Skeleton Configurations</td>
<td>23</td>
</tr>
</tbody>
</table>
1. Introduction

This deliverable describes a static mapping technique, as specified under WP3. The work here builds upon the idea of software components (described in D3.1). In Chapter 2, we first describe a new methodology for programming heterogeneous parallel systems, based on refactoring (D4.2) and a new Monte-Carlo Tree Search technique for deriving static mappings (described in Chapter 3). We then demonstrate our methodology and static mapping technique on a number of use-cases, taken from different domains, including image convolution, ant-colony optimisation and a use-case taken from HLRS (the molecular dynamics BasicN2 use-case). Finally, we introduce PEI, a Performance Enhancement Infrastructure which is a collection of mechanisms and policies used for improving the performance of skeleton based frameworks for use with the static mapping.

Our programming methodology and PEI are demonstrated in the context of C++ and FastFlow skeleton library, but the mechanisms are completely generic and can therefore be easily adapted to other languages/skeleton frameworks. We are currently in initial stages of adapting the methodology to Erlang and Skel library, and making Skel VIP-compatible.

For the purposes of this deliverable, we define static mapping to be an assignment of the number of workers in each farm skeleton in a given skeleton configuration. The static mapping also includes the type of each worker in the farm (whether it is a CPU or a GPU worker).

1.1 What is new in this deliverable?

- We introduce a new heterogeneous parallel programming methodology, supporting the near-optimal derivation of static mappings of skeleton components to available hardware resources. Our methodology builds on top of the C++ refactoring work reported in D4.2 from WP4.

- We introduce a new technique for deriving near-optimal static mappings using a Monte-Carlo Tree Search technique.

- We demonstrate our methodology on a number of use-cases, including the use-case from the industrial partner, HLRS, the BasicN2, as reported in WP6 (D6.5).
We describe a novel generic infrastructure (Performance Enhancement Infrastructure) for improving the performance of skeleton based frameworks.

1.2 Skeletons

In this deliverable, we focus on two categories of skeleton: sequential skeletons, which abstract the structure of a purely sequential computation with no added parallelism; and, parallel skeletons, which implement specific parallel patterns. The skeletons are described in more detail in D2.1, but here we introduce notation for each skeleton that we use in the rest of this deliverable. We assume that all of the input tasks for skeletons are independent. We consider the following sequential skeleton:

- The Compose ($\circ$) skeleton represents sequential function composition applied to a sequence of inputs, where $f_1 \circ f_2$ denotes a sequential composition of two functions, $f_1$ and $f_2$.

- 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 barrier synchronisation between $f$ and $g$.

We also consider two widely-used parallel skeletons:

- The Pipeline ($\parallel$) skeleton applies the composition of the functions $f_1, \ldots, f_n$, in parallel to a sequence of independent inputs $x_1, \ldots, x_m$, 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 ($\Delta$) skeleton, $\Delta(nwCPU, nwGPU, f, x)$, represents the application of a single function, $f$, to the sequence of independent inputs, $x_1, \ldots, x_n$, 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: $\Delta(nwCPU, nwGPU, f_1 \parallel f_2, x)$. 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. A Heterogeneous Parallel Programming Methodology

The derivation of near-optimal static mappings is part of a broader parallel programming methodology [3] that we have developed, aimed at increasing programmability of heterogeneous parallel systems. Our methodology aims to support both the inexperienced parallel programmer with little knowledge of parallel programming techniques as well as the experienced parallel programmer, who seeks to maximize productivity with appropriate tool support to automate the process.

Our general methodology is shown in Figure 2.1 and comprises a number of steps, described below.

1. 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 1.2. 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 and tasks. We also record what implementations (CPU, GPU or both) exist for which unit of computation. As a simple example, consider the following piece of code:

```c
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`. Basic units of computation are the functions `read_image` and `process_image` (cont.), and the tasks are applications of these functions to the elements of the array, `imageFiles`. We may only have a CPU implementation of the `read_image` function, but both CPU and GPU implementations (kernels) of the `process_image` function. Using the notation from Section 1.2, we can denote this by \( r \circ p \), where \( r \) is `read_image` function, \( p \) is `process_image` function, and \( \circ \) is sequential composition.

2. Given the text file with the identified skeleton from the original application,
all possible equivalent skeleton configurations are automatically generated
(up to a given depth of nesting) resulting in a number of different possible
parallelisations. Equivalent skeleton configurations may be generated using
a number of standard transformations. Given an initial configuration, each
composition (◦) can be transformed into a parallel pipeline (∥) and a farm
skeleton (Δ) can be introduced for any skeleton configuration. Similarly
inverses of these transformations can also be applied, that is the equivalences
$S_1 ∥ S_2 ≡ S_1 ◦ S_2$ and $Δ(S_1) ≡ S_1$ hold true for any pattern (or pattern
composition) $S_1$ and $S_2$. For example, we may transform a parallel pipeline
into a sequential composition, or eliminate a farm skeleton altogether.

In Figure 2.1, we refer to step 2 as enumerating the skeleton configurations.
For example, in the previous step we introduced the initial structure to be
$r ◦ p$; therefore, the possible skeleton configurations are $r ◦ p$, $Δ(r ◦ p)$,
$Δ(r) ◦ p$, $r ◦ Δ(p)$, $Δ(r) ◦ Δ(p)$, $r ∥ p$, $Δ(r) ∥ p$ etc.

3. Using profiling information, these configurations are filtered using a cost
model, if desired, to restrict the number of possibilities that need to be con-
sidered. This allows us to eliminate possible parallelisations with little or no
potential speedup at an early stage of development. In Chapter 4, 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.

In our example, the cost model may predict that $Δ(r) ∥ Δ(p)$, $Δ(r) ∥ p$ and
\Delta(r) \circ \Delta(p) \text{ are the best candidates from all possible skeleton configurations.}

4. 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. For the purposes of this chapter, a static mapping is an assignment of a 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 the next section, we describe a model, based on Monte Carlo Tree Search (MCTS) technique, that can be used to derive static mappings for a given skeleton configuration.

In our example, this step may 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 \(\Delta(p)\).

5. The programmer then chooses one of the parallelisations together with its static mapping.

6. The programmer then refactors the original application from Step 1, introducing the desired skeleton configuration from Step 5 using the C++ refactoring tool from D4.2. The refactoring tool performs all the required program transformations and condition checking automatically, introducing the skeleton structure and the parameters from Step 4. Considering the example code from Step 1 and the skeleton configuration, \(\Delta(r) \parallel \Delta(p)\), the refactoring tool may produce the following:

```cpp
ff_farm<> readFarm;
...
for(int i = 0 ; i< nworker1; i++)
    readFarm.push_back(&read_image)
for(int i = 0 ; i< gpu_nworker2; i++)
    readFarm.push_back(&read_image_gpu)
ff_farm<> processFarm;
...
for(int i = 0 ; i< cpu_nworker2; i++)
    processFarm.push_back(&read_image_cpu)
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. 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.
3. Static Mapping

In this chapter, we introduce a new technique for deriving near-optimal static mappings based on a Monte-Carlo Tree Search. We aim to take advantage of both CPUs and GPUs for component placement, in order to achieve the best speedup of a program. We will therefore restrict the notion of static mapping solely to decisions about the number and type (CPU/GPU) of instances of each component that should be created during the program execution.

3.1 Monte Carlo Tree Search (MCTS) Based Model for Deriving Static Mappings

In this section, we describe one possible approach for deriving static mappings of a given skeleton configuration. In addition to information about the structure of the nesting of skeletons, we also rely on information the sequential components used, i.e. what versions (CPU, GPU or both) of the component are available and what is the estimated runtime for each version for one input on a given hardware. Our approach produces as output the number and type of each component that should be instantiated. For more details, see [4].

Given a skeleton configuration, our approach proceeds in two phases:

- Conversion of the skeleton configuration to its associated process graph
- Application of Monte Carlo Tree Search to generate a near-optimal mapping of the process graph to the target architecture

3.1.1 Converting the Skeleton Tree to the Process Graph

The skeleton configurations we use here are a nested composition of farms and pipelines. Figure 3.1 shows a skeleton structure of a 3-stage pipeline. Each stage is a farm and the second stage contains CPU and GPU components. In terms of the notation introduced in Sec. 1.2, this skeleton tree structure may be expressed as

\[ \Delta(n_A, 0, A) \parallel \Delta(n_{BG}, m_{BG}, BG) \parallel \Delta(n_C, 0, C) \]

The skeleton configuration is first converted into a first order logic formula from which a process graph can be created, using the following scheme:

\[ \Delta(n_A, 0, A) \parallel \Delta(n_{BG}, m_{BG}, BG) \parallel \Delta(n_C, 0, C) \]
\( \Delta(f) := f_{CPU} \lor f_{GPU} \)

\( f_1 \parallel f_2 \parallel \cdots \parallel f_m := \land_{i=1}^{m} f_i \)

\( f := g_1 \parallel g_2 \parallel \cdots \parallel g_m | \Delta(g)| g \)

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

As an example, for the skeleton configuration described in Figure 3.1, the formula is as follows. \( FOL(TS) = (A \land (B \lor G) \land C) \).

To create a process graph from the formula, each preposition represents a component, each operator \( \land \) represents a queue connecting the components of the systems together and the operator \( \lor \) represents different components of the system streaming data from the same queue.

Figure 3.2 shows the process graph generated from \( FOL(TS) \). 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. For each task queue, the queue level is the number of tasks waiting to be processed by the next pipeline 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, finding the optimal number of instances of each component is key to creating the balance of queues and achieving the best throughput of a system, where throughput is defined as the number of tasks in the last task queue in time interval $T$. This is the job of our mechanism for deriving static mappings.

3.1.2 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 certain amount of remaining resources to one or more components, limiting the resources available to yet unallocated components. A leaf of the tree corresponds to a complete static mapping, where zero or more resources are allocated to each component. At this point the mapping can be evaluated by simulation. We apply Monte Carlo Tree Search (MCTS) technique 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:

$$\{ A(w_i, k) \mid w_i \in M, k \in \mathbb{N}, k \leq L \},$$

where $A \in \{ADD, REMOVE\}$; $M$ 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.

3.1.3 Selection Strategy

MCTS operates by repeatedly visiting and evaluating possible mapping decision values (e.g. number of workers for a particular component) for each decision in sequence. When a decision value is visited, subsequent decisions are made at random to create a complete mapping. This mapping is then evaluated by simulation, which we describe in the following section. Each simulation returns a reward for the mapping, which is back-propagates to the decision value that was visited. Particular policies for back-propagation are explained below. Rewards accumulate on particular decisions values, building an expectation of the quality of that particular value. Once several evaluations have been made for each decision value, MCTS is ready to select a particular value for that decision, based on the expected quality of all accumulated values. The selection strategy that we use is **Upper Confidence bounds applied to Trees** (UCT), a well-established approach in the MCTS literature. The formula for UCT is:

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

where $n$ is the number of times the current decision 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, $\overline{X}_j$ is the average reward value given to child node, $j$. 

13
3.1.4 Simulation of Static Mapping

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. We have determined empirically on test programs that our simulator is accurate enough to enable us to compare different static mappings.

3.1.5 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:

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

where $N$ is the total number of components in the system; $U(w_i)$ is the utilisation of the component, $w_i$; and, $U_{mean}$ 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.

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

Here, $L$ is the total number of queues in the system, $T_{Q_i}$ is the throughput of the $Q_i$, and $T_{mean}$ 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, $P_{GPU} = \sqrt{\frac{N_{UG}}{N_{TG}}}$, where $N_{UG}$ is the number of GPUs in the system that have not been used by the static mapping, and $N_{TG}$ is the total number of available GPUs in the system.
2. CPU penalty, \( P_{CPU} = \sqrt{\frac{N_{UC}}{N_{TC}}} \), where \( N_{UC} \) is the number of CPU cores in the system that have not been used by the static mapping, and \( N_{TC} \) is the total number of available CPU cores in the system.

The main effect of \( P_{GPU} \) and \( P_{CPU} \) 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 a dedicated private machine, we do not use \( P_{GPU} \) and \( P_{CPU} \) 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 \( Q(v) = T - (SD_U + SD_Q + P_{CPU} + P_{GPU}) \).

### 3.1.6 Back-propagation, Termination Condition and Final Move Selection

We have considered two back-propagation policies: 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 terminates 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.
4. Case Studies

In this chapter, we demonstrate the methodology described in Chapter 2 on three realistic benchmark applications. For each of the applications, we illustrate the different stages of its parallelisation as follows:

1. starting from a sequential version, we show a number of different possible skeleton configurations;

2. if there are many different possible skeleton configurations, we pre-filter these configurations using a cost model described in [2] 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.

4.1 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:

\[
\text{output}_{\text{pixel}}(i, j) = \sum_m \sum_n \text{input}_{\text{pixel}}(i - n, j - m) \times \text{filter}_{\text{weight}}(n, m)
\]  

(4.1)

### 4.1.1 Configurations and Cost-Model Filtering

Figure 4.1 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 1.2, 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 4.1, the bold results are the three best configurations we have selected for further processing using the MCTS model.

### 4.1.2 Optimal Static Mappings Determined by MCTS

Figure 4.2 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.
Figure 4.2: 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.

### 4.1.3 Evaluation of Skeleton Configurations

All experiments are on a stream of 25 4096*4096 images. Figure 4.3 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 4.3 shows, this mapping gives an actual speedup of 39.14. This may be compared to the known 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 4.4 we show the speedups for $\Delta(r) \parallel \Delta(p)$ skeleton configuration. The $x$ axis shows the number of CPU workers for $\Delta(r)$, 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, MCTS predicts the optimal speedup for 6 CPU workers for $\Delta(r)$ and $(0, 3)$ CPU and GPU workers for $\Delta(p)$. Figure 4.4 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 4.5 shows the speedups for the skeleton configuration, $\Delta(r \parallel p)$. The best speedups for this configuration were obtained when the number of CPU and GPU workers are equal for $\Delta(r \parallel p)$. As Figure 4.5 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 4.2).
Figure 4.3: Speedup graph for the Image Convolution configuration $\Delta(r) \parallel p$, where $p$ is executed on a GPU.

Figure 4.4: Speedup figures for the Image Convolution configuration $\Delta(r) \parallel \Delta(p)$, with 0 CPU and 1 GPU used by $n$ workers for $\Delta(p)$. 
4.2 Ant Colony Optimisation

Ant Colony Optimisation (ACO) [5] 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, \( \tau \), 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.

4.2.1 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
4.2.2 Optimal Static Mapping Determined by MCTS

Figure 4.6 shows the output of MCTS for the $\Delta(s); p; u$ configuration for the ACO example.

<table>
<thead>
<tr>
<th>Mapping (C,G)</th>
<th>$\Delta(s); p; u$</th>
</tr>
</thead>
</table>

Figure 4.6: 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 4.7: Speedup graph for the ACO configuration $\Delta(s); p; u$

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.

21
4.2.3 Evaluation of Skeleton Configurations

Figure 4.7 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 4.6) 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.

4.3 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 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.

4.3.1 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
Figure 4.8: MCTS predicted optimal mapping for Molecular Dynamics example with $\Delta(r \circ h)$ configuration. $(C, G)$ denotes the number of CPU and GPU workers for a farm.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>$\Delta(r \circ h)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(CPU, GPU)$</td>
<td>$(22, 1)$</td>
</tr>
</tbody>
</table>

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.

### 4.3.2 Optimal Static Mapping Using MCTS

Figure 4.8 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.

### 4.3.3 Evaluation of Skeleton Configurations

Figure 4.9 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 4.8 illustrates, the predicted mapping is $(22, 1)$ (i.e., 22 CPU workers and 1 GPU worker). From Figure 4.9, 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 4.9: Speedup graph for the Molecular Dynamics configuration $\Delta (r \circ h)$.
5. Performance Enhancement Infrastructure

In this part of the deliverable we describe the Performance Enhancement Infrastructure (PEI), a collection of mechanisms and policies used for improving the performance of skeleton-based frameworks through static mapping. We also provide guidelines on how to use the tools provided in PEI. PEI is used for profiling the application in order to gain information needed for the methodology described in Chapter 2 and for the static mapping framework described in Section 3.

5.1 The Performance Enhancement Infrastructure

The novelty of our approach lies in analysing, visualising, and optimising the coordination mechanism of the Heterogeneous (CPU/GPU) skeleton-based frameworks, transparently from the user applications. Figure 5.1 shows an architectural overview of PEI. Virtual Interaction Protocol (VIP) enables interaction between PEI and skeleton-based frameworks, offering portability and generality and allowing us in principle to instrument arbitrary skeleton framework running on arbitrary CPU/GPU heterogeneous platform. The Dynamic Skeleton Runtime Interface (DSRI) is an interface to external tools, that is also able to communicate via VIP. The next two sections describe VIP and DSRI in more details.

5.1.1 VIP: Virtualisation Interaction Protocol

VIP protocol provides a standard for interaction between all parts of an instrumented skeleton application. The protocol covers four different types of information:

1. **Structural Information** that captures the high-level structure of parallel patterns.

2. **Behavioural Information** related to performance parameters of the components of skeletons.

3. **Instructional Information** that provides an interface to set or modify control parameters.
4. **Architectural Information** about the underlying hardware platform.

### 5.1.2 DSRI: Dynamic Skeleton Runtime Interface

DSRI uses VIP for communication and serves as a bridge between performance enhancement tools (described later) and an application being executed. It provides access to structural and behavioural information retrieved from the components and also forwards instructions from external controllers to these components. It can also communicate with remote tools via proxy, provided that the tools support VIP. DSRI consists of **sensors**, which gather information from a running application and **actuators**, which are control parameters of components [?].

**Actuators**  Actuators are tuneable parameters that affect the non-functional behaviour and configuration of the application, including mapping of the components to resources (CPUs and GPUs), component execution states (online or offline) and the load balancing policy for distributing tasks to components. The actuators can be tuned statically at program initiation or dynamically at runtime.

**Sensors**  Sensors provide runtime information from the application including i) *structural metadata* that describes the skeleton configuration and certain features of components and ii) *profiling information* related to service time of components and their utilisation, throughput and status.

DSRI offers two different types of information monitoring, which incur different performance overheads:
1. **Comprehensive Monitoring**, where information is gathered about every individual task executed by a component. This monitoring introduces some performance overhead and is intended for initial data gathering or after phase change events that alter the runtime characteristics of a program.

2. **Sparse Monitoring**: where information is gathered for only a certain fraction of tasks. This intended to be the default mode.

### 5.1.3 Performance Enhancement Tools

Performance Enhancement Tools (PET) provide a set of mechanisms for controlling the application behaviour and improving its performance. These mechanisms monitor the runtime behaviour of components (in terms of their service time) and compare it to the expected behaviour (derived from the performance models associated to the skeleton structure). In the following, we describe the available controlling mechanisms.

**Task distribution** When both CPU and GPU components of the skeleton configuration are instantiated, it is very important to appropriately distribute the tasks to these components, such that both CPU and GPU components contribute to computation of the final result and they terminate as close to each other as possible. This reduces overheads caused by components waiting for other components to complete. In PET, we use *divisible load theory (DLT)* to distribute tasks to components. Our mechanism distributes tasks to components proportionally to the service times of components.

**Heuristic Mapping** A PET called Mapper is provided to optimise the placement of *service* components, in order to maximise the floating point resources for each worker component. Mapper applies this strategy whenever there are sufficient cores available. However, given that an application can exploit arbitrary pattern nesting parallel pattern, there are cases where the number of computing components surpasses the number of cores. In these cases, Mapper tries to allocate the CPU/GPU component into available devices where all processors receive a fair amount of workload based on components effective service time.

**Efficient Idling** The key objective here is to obtain maximum utilisation of the resources for any CPU slot allocated to a component. The optimised usage of the allocated slot of each component depends on the availability of a task on the component queue. If the task is not available the component must wait for it. This waiting can be either a busy wait loop or a thread sleeping until the data becomes available. In the former case, especially when the number of threads running the application is less than or equal to the number of available cores, the performance of the application does not drop. However, when the number of threads running the application is more than the number of cores, the sleeping mechanism can dedicate
this time slot to those threads which have received their tasks but are waiting for resources to execute them. Moreover, the energy consumption of thread sleeping is higher and less optimised than busy waiting, which is also detrimental to optimal resource utilisation.

Therefore, our strategy is to provide an extra information layer on top of the operating system scheduler by putting a thread which is executing a component to sleep, whenever the component input/output queue is empty/full. Applying this strategy on frameworks supporting busy waiting can be made on demand and is useful when the number of threads is bigger than the number of available cores and also preventing from waisting energy by optimal utilisation of the resources are of high priority.

5.1.4 Instrumenting FastFlow with PEI

While the proposed approach is intended to be generally applicable to different skeleton frameworks, we apply the performance enhancement infrastructure specifically to the FastFlow framework.

**Instrumenting FastFlow** FastFlow is designed around a uniform object oriented class hierarchy with components (which may be a pipeline stages or a farm worker) and skeleton constructs (pipelines, farms and maps) implemented as subclasses of the ff_node base class.

1. DSRI: We have introduced a VIP-compliant DSRI manager node in FastFlow that connects with the root of a skeleton tree and retrieves or passes structured data. This DSRI node executes in a separate thread and uses non-blocking locks during information retrieval to maintain the lock-free semantics of FastFlow - a central aspect of the architecture.

2. **ff_node** Instrumentation: This introduces methods to allow all subclasses to interact with the DSRI node and expose sensor and actuator interfaces.

3. Instrumented Coordination Patterns: The pipeline, farm and map patterns are updated to expose their own sensor and actuator interfaces as well as those of their child components. This enables recursive extraction and forwarding of sensor states and actuator controls from the root of the skeleton tree by the DSRI manager node in a single operation using a corresponding tree data structure.

4. Instrumented Structural Subcomponents: **ff_loadbalancer** and **ff_gatherer** are specialised structural components that perform work distribution, load balancing and recombination in FastFlow farms. These subcomponents may be subclassed in user applications.
FastFlow incorporates tracing functions that collect output performance information at program termination. These performance counters are output as unstructured textual information. By exposing these performance counters, previously output in the form of unstructured textual information, as sensors, the DSRI node retains comprehensive information about the executing program.

The available actuators include CPU/GPU component mappings and load balancer policies that previously required source code intervention to modify.

5.2 The PEI tool HOWTO (Rel 1.0)

Installation  The code can be downloaded and/or tested from two different sources:

- the ParaPhrase project web site, under the Deliverables tab, or through the direct url
  
  http://paraphrase-ict.eu/Deliverables/deliverable-3.2/deliverable-3.2-prototype/

- by logging into the project host platform xookik (IP address: 194.66.84.87) with the account name paraphrase. Login is available through a password which can be requested by e-mailing m.goli@rgu.ac.uk. The account provides a pre-configured environment in which all of the code is installed and ready to be used.

PEI uses the JSON spirit library, which is provided both in the tarball from the ParaPhrase web site and in the project host platform account.

Compilation Flags  The application has to be compiled with the following flags:

-DTRACE_FASTFLOW -L <jason directory> -ljson_spirit

Usage  To use PEI, programmers should invoke the following method only once for any component in the skeleton tree, before calling the run function.

- PROFILE(&comp);
  This activates VIP instrumentation on that subtree for both mapping and, if applicable, load-balancing as default. This function can accept 2 more optional parameter that control the level of automation.
  PROFILE(&comp, boolean);
  This boolean value determines whether or not Mapper is used.
  PROFILE(&comp, boolean, boolean2);
  The second boolean determines whether or not DLT is used.

---

1 the library provided is the one available from http://goo.gl/G9tHF
After execution, profiling information in JSON format will be output to the locations

```
profiling/<executable-name>/actuator.json
```

and

```
profiling/<executable-name>/sensor.json.
```

Profile information can be visualised with the JSON Online Viewer (available from [http://jsonviewer.stack.hu/](http://jsonviewer.stack.hu/)).

**Sample usage** The following is a simple FastFlow farm example with 4 workers instrumented with PEI. As stated in the example, the user needs to add line 14 in order to use PEI. On running the code for the first time, the sensor file will be generated containing the behavioural information (profiled data) and the structural information of the application. On running the code for a second time, the DSRI manager uses the available PETs to process the Sensor information and build the Actuator file containing specific information for the application. It then passes the information to the root of the application for further tuning.

```c
int main() {
...
ff_farm<adaptive_loadbalancer> *farm = new ff_farm<adaptive_loadbalancer>;
Emitter E(streamlen);
farm->add_emitter(&E);
for(int i=0;i<nworkers;++i) w.push_back(new Worker);
farm->add_workers(w);
Collector C;
farm->add_collector(&C);
PROFILE(farm);
if (farm->run_and_wait_end()<=0) {
    error ("running farm\n");
    return -1;
}
return 0;
}
```

After execution, the sensor file produced is:

```json
{
  "Collector": {
    "Assigned_Processor": 0,
    "Data_Size": 0,
    "FF::Node_Subclass": "ff::ff_gatherer",
    "Maximum_SVC_Ticks": 130,
    "Minimum_SVC_Ticks": 21,
    "Pop_Delay_Count": 31728,
    "Pop_Delay_Ticks": 158640000,
    "Push_Delay_Count": 0,
    "Push_Delay_Ticks": 0
  }
}```
"Sampled_SVC_Tick_Distribution":{
},
"Sampling_Rate":0,
"Total_Number_Of_Tasks":8,
"Total_SVC_Ticks":393,
"Type":"FF::GATHERER"},
"Emitter":{
"Assigned_Processor":3,
"Data_Size":0,
"FF::Node_Subclass":"adaptive_loadbalancer",
"Maximum_SVC_Tick":44467,
"Minimum_SVC_Tick":189,
"Pop_Delay_Count":0,
"Pop_Delay_Ticks":0,
"Push_Delay_Count":0,
"Push_Delay_Ticks":0,
"Sampled_SVC_Tick_Distribution":{
},
"Sampling_Rate":0,
"Total_Number_Of_Received_Tasks":9,
"Total_Number_Of_Send_Tasks":8,
"Total_SVC_Ticks":48123,
"Type":"FF::LOADBALANCER"},
"FF::Node_Subclass":"ff::ff_farm<adaptive_loadbalancer, ff:: (cont.)ff_gatherer>",
"Local_ID":-1,
"Type":"FF::FARM",
"Workers":[
{
"Assigned_Processor":2,
"Data_Size":0,
"FF::Node_Subclass":"Worker",
"Local_ID":0,
"Maximum_SVC_Tick":165,
"Minimum_SVC_Tick":26,
"Pop_Delay_Count":22867,
"Pop_Delay_Ticks":22867000,
"Push_Delay_Count":0,
"Push_Delay_Ticks":0,
"Sampled_SVC_Tick_Distribution":{
},
"Sampling_Rate":0,
"Total_Number_Of_Tasks":2,
"Total_SVC_Ticks":191,
"Type":"FF::NODE"},
{
"Assigned_Processor":3,
"Data_Size":0,
"FF::Node_Subclass":"Worker",
"Local_ID":1,
"Maximum_SVC_Tick":144,
"Minimum_SVC_Tick":144,
"Pop_Delay_Count":28300,
"Pop_Delay_Ticks":28300000,
"Push_Delay_Count":0,
"Push_Delay_Ticks":0,
"Sampled_SVC_Tick_Distribution":{
},
"Sampling_Rate":0,
The actuator file produced is:

```json
{
    "Collector":{
        "Assigned_Processor":0,
        "FF::Node_Subclass":"ff::ff_gatherer",
        "Type":"FF::GATHERER"
    },
    "Emitter":{
        "Assigned_Processor":3,
        "FF::Node_Subclass":"adaptive_loadbalancer",
        "Masking_Array":[
            1,
            1,
            1,
            1
        ],
        "Total_Number_Of_Received_Tasks":9
    }
}
```
"Total_Number_Of_Send_Tasks": 8,
"Type": "FF::LOADBALANCER",
"Workload": [
0.17832134376005776,
0.09843381175558185,
0.12304172719443964,
0.60020354728995051
],
"FF::Node_Subclass": "ff::ff_farm<adaptive_loadbalancer, ff::
(cont.)ff_gatherer>",
"Local_ID": -1,
"Type": "FF::FARM",
"Workers": [
{
"Assigned_Processor": 2,
"FF::Node_Subclass": "Worker",
"Local_ID": 0,
"Status": 1,
"Type": "FF::NODE"
},
{
"Assigned_Processor": 3,
"FF::Node_Subclass": "Worker",
"Local_ID": 1,
"Status": 1,
"Type": "FF::NODE"
},
{
"Assigned_Processor": 0,
"FF::Node_Subclass": "Worker",
"Local_ID": 2,
"Status": 1,
"Type": "FF::NODE"
},
{
"Assigned_Processor": 1,
"FF::Node_Subclass": "Worker",
"Local_ID": 3,
"Status": 1,
"Type": "FF::NODE"
}
]
6. Conclusion

In this deliverable 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. Finally we introduced PEI, a Performance Enhancement Infrastructure that provides profiling tools to be used within the methodology/static mapping. 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 Erlang. 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 Chapter 4 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 straightforward 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, mapreduce, 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.
Bibliography


