Programming Heterogeneous Parallel Machines Using Refactoring and Monte-Carlo Tree Search

Brown, Christopher J.; Janjic, Vladimir; Goli, Mehdi; McCall, John

Published in:
International Journal of Parallel Programming

DOI:
10.1007/s10766-020-00665-z

Publication date:
2020

Document Version
Publisher's PDF, also known as Version of record

International Journal of Parallel Programming (2020) 48:583–602
https://doi.org/10.1007/s10766-020-00665-z

Programming Heterogeneous Parallel Machines Using Refactoring and Monte–Carlo Tree Search

Christopher Brown[1] · **Vladimir Janjic**[1] · **M. Goli**[2] · **J. McCall**[2]

Received: 16 October 2019 / Accepted: 27 May 2020 / Published online: 10 June 2020

Abstract
This paper presents a new technique 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 speedups obtained by mappings derived by the MCTS algorithm are within 5–15% of the best-obtained manual parallelisation.

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

✉ Christopher Brown
cmb21@st-andrews.ac.uk

Vladimir Janjic
vj32@st-andrews.ac.uk

M. Goli
m.goli1@rgu.ac.uk

J. McCall
j.mccall@rgu.ac.uk

[1] School of Computer Science, University of St Andrews, St Andrews, UK

[2] Robert Gordon University, Aberdeen, UK

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 technique for programming heterogeneous parallel systems that: (1) automatically discovers which parallel structure to exploit; (2) computes a near-optimal mapping of work onto the various heterogeneous processing elements; and, (3) provides a semi-automatic way of introducing the chosen parallel structure into the original program, and instantiating this with the derived mapping information. Our technique is based on a combination of algorithmic skeletons [11] for defining the parallel structure, a method of finding a mapping for tasks on heterogeneous architectures and refactoring tool support for user-guided introduction of the skeletons and mapping decisions.

We show the generality of our technique by using realistic use-cases from three different domains (image processing, heuristic optimisation and molecular dynamics), programmed using the FastFlow [3] skeleton library for C++, which uses OpenCL and CUDA for GPU computations. While some particular parts of the technology (e.g. refactoring) necessarily depend on the syntax of C++ language, the general methodology could, in principle, be applied to other languages and paradigms (e.g. Erlang [18]). The paper makes the following research contributions:

1. we introduce a new technique 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 technique, 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* [11] 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:

- 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. This skeleton can be used, for example, in the map-reduce like computations, such as the one described in Sect. 5.2, to synchronise between the map and reduce phase.

We also consider two widely-used parallel skeletons:

- 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.
- 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 $(f_1 \parallel f_2 \parallel \cdots \parallel f_n)(x)$. Note that the pipeline skeleton does not accept the number of workers as an input, because a pipeline stage is always executed in one thread. If we want multiple threads to execute a single pipeline stage, to parallelise processing of items from its inputs, we compose it with the farm skeleton.

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, (1) a farm whose worker function is f, and (2) 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 [22], and the concept goes back at least to the fold/unfold

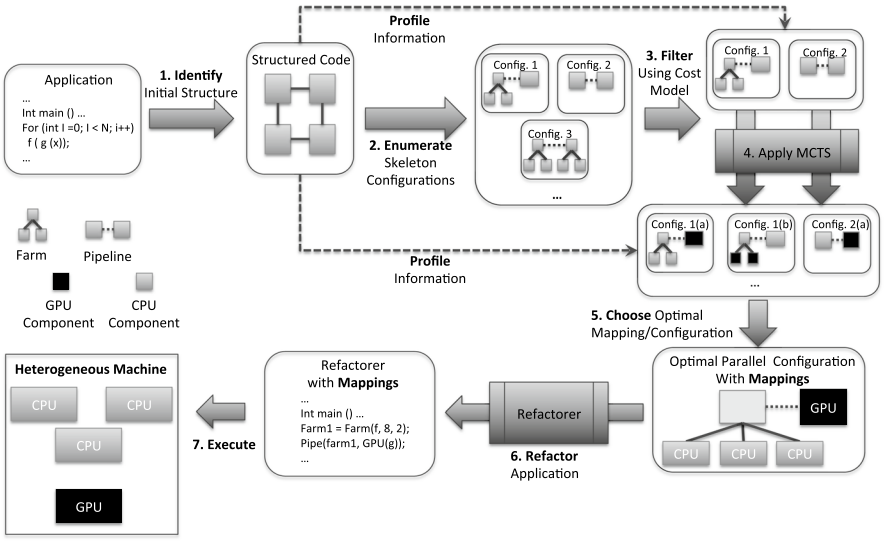

Fig. 1 Overview of our technique for programming heterogeneous multi-core systems

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 refactoring extension [7] for Eclipse, that introduces and tunes parallelism in C++ by introducing a nesting of skeletons into the application semi-automatically by user-guidance.

3 Programming Heterogeneous Parallel Machines

In this section we introduce a new parallel programming technique aimed at increasing the programmability of heterogeneous parallel systems. Our technique 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 technique is shown in Fig. 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 Sect. 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 (components) and tasks. Components

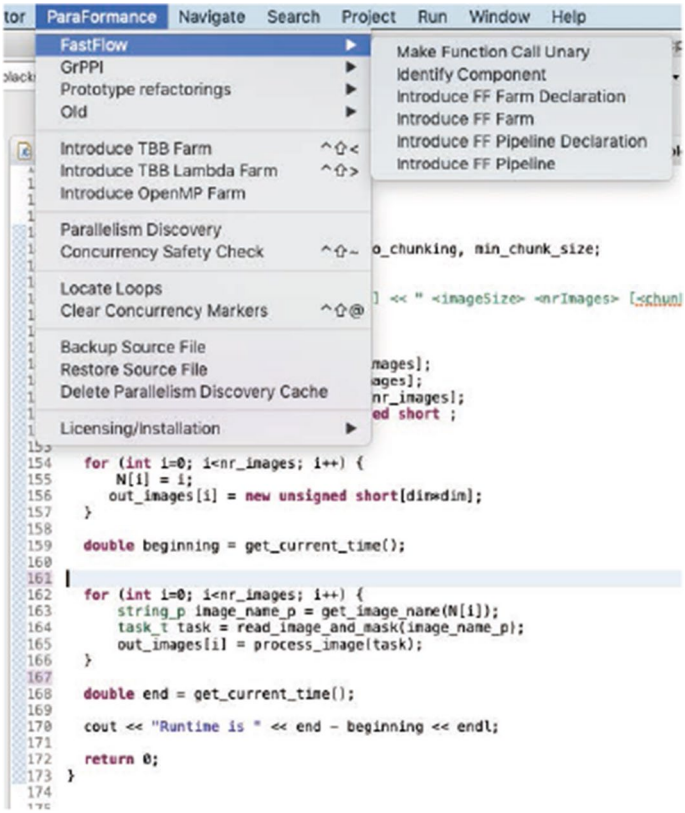

Fig. 2 Source code for image convolution before refactoring

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 piece of code in Fig. 2 at lines 162–166.

The structure of this code is a composition of two functions, `read_image` and `rprocess_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 Sect. 2, 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. In the case of the GPU computation, this also includes the time it takes to transfer the data to/from the GPU. This timing information is used in subsequent

steps of our methodology. 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 0.2 ms, running CPU version of `process_image` on one image takes 6.6 ms and running GPU version of `process_image` on one image takes 0.08 ms.

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.[1] 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 to restrict the number of possibilities that need to be considered. This allows us to eliminate parallelisations with little or no potential speedup at an early stage of development. In Sect. 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 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

[1] Since we assume that all functions operate on streams, it is always possible to replace a function with a farm skeleton operating on elements of the input stream in parallel.

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

ff_farm<> processFarm;
for(int i = 0 ; i< nworker2; i++)
    processFarm.push_back(&process_image_cpu);

for(int i = 0 ; i< nworker2; i++)
    processFarm.push_back(&process_image_gpu);

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

Fig. 3 Source code for image convolution after refactoring

Carlo Tree Search (MCTS) algorithm [9]. This step is fully automatic, and is also computationally the most expensive part of the technique. Exactly how much time it takes to rank the configurations and derive mappings depends mostly on the method used for estimating 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. *Refactoring 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 output as in Fig. 3, 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 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 estimated near-optimal mappings (with some of the skeleton parameters fixed) and the leaves of the tree correspond to complete mappings. The root of the tree corresponds to the near-optimal mapping of the whole skeleton configuration (with none of the parameters fixed). The children of a node represent different possibilities for fixing a yet unfixed 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 a fixed number of times, attaining a reliable evaluation of the partial mapping in step 1 by evaluating a fixed number of random complete mappings that correspond to it. In this way, we avoid the exhaustive search of all complete mappings corresponding to that partial one. 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 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

Table 1 Solution space and time needed for its full evaluation for Image Convolution on different hardware configurations

CPU cores	GPUs	App components	Sol. size	Time for eval. (s)
16	1	4	1240	> 86,400
24	2	4	6624	> 604,800
64	2	4	129,204	> 3,628,8000

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

where $S(M)$ is the estimated throughput of the whole system (i.e. the number of tasks per unit of time that get processed, obtained using profiling) $\delta_U(M)$ 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. In this way, if two mappings have a similar throughput, the one which has smaller deviation from the standard utilisation of the resources (and which, hence, uses resources more uniformly) will be preferred. 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).

4.1 Adaptation of the MCTS Technique to the Static Mapping Problem

It is well known that the MCTS technique is most often used to find a single best move at the root of the game tree. In our adaptation of this technique to the static mapping problem, nodes of the game tree correspond to fixing of the parameters of the skeleton configuration. The best move at the root of the tree represent assignments of all the parameters to all the components of the application. This move is computed by considering all the children of the tree, which correspond to moves where we fix the first parameter of the configuration (i.e. we allocate one type of resources, CPUs or GPUs, to one of the skeleton components) and the others are chosen freely. Grandchildren of the root represent moves where we fix the first two parameters and freely chose the others and so on.

Suitability of Using MCTS to Derive Static Mappings The main target for the MCTS-based approach for deriving static mappings are computationally-heavy parallel applications that contain nested parallelism in the form of farms and pipelines. Such applications might take hours or even days to execute and may need to be executed repeatedly, so the effort required by the MCTS model to derive near-optimal mappings is well justified by savings in time and energy of the optimised parallel applications. In addition, the solution space, even when the degree of nesting of skeletons is relatively small, is sufficiently large to justify the use of MCTS.

For example, for the Image Convolution problem considered in Sect. 5.1, with the depth of skeleton nesting of 2, the Table 1 gives an example sizes of solution space for different hardware configurations and the estimated time needed to evaluate all of them using full profiling (which is the only way to give the accurate estimation of the execution time). From the table, we can see that for even modestly-sized parallel systems, the time to evaluate all parameters would be hundreds of days order of magnitude. Since parallel systems are becoming larger and larger, with more CPU cores and more GPU devices being available in a single shared-memory system, the problem will only become more time-consuming.

4.2 MCTS parameters

The selection strategy that we use is the *Upper Confidence bounds applied to Trees* (UCT) [19]. 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 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. The experiments showed that the value of around 1/5th of the average throughput for C_P gives the best results, being a good tradeoff between reducing the search space and making sure we do not get stuck in the local optimum. As for the back-propagation policy, we considered two policies—*Max* policy, where the maximal reward of all the children is propagated to their parent, and the *Average* policy, where the average reward of all the children is propagated to their parent. The experiments showed that the *Average* policy works better, being less greedy. For more details, see [14].

5 Case Studies

In this section we demonstrate our technique 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 [6] 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.

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

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

We consider applications that belong to different domains, showing the generality of our parallelisation technique. 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 2×2.4 Ghz 12-core AMD Opteron 6176 CPUs, coupled with an NVidia Tesla C2050 graphic card with 448 CUDA cores running at 1.16 GHz, running CentOS Linux. 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 for blurring, smoothing and edge detection. We consider an instance of the image convolution from video processing applications, where we are given a sequence of images, each of which is first read from the disk and then subsequently processed by applying a filter. This can be represented as a composition of two functions (applied to a stream of images), $r \circ p$, where r is the function that reads the file and p is the function that processes it. Applying a filter to an image consists of computing a scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels:

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

5.1.1 Configurations and Cost-Model Filtering

Since the composition of functions is applied to a stream of images, it is possible to parallelise both of the functions in it—we can read multiple images at the same time, apply a filter to a multiple images at the same time, or do both of these together. Table 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, and the second column shows the cost-estimated minimal

Table 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

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

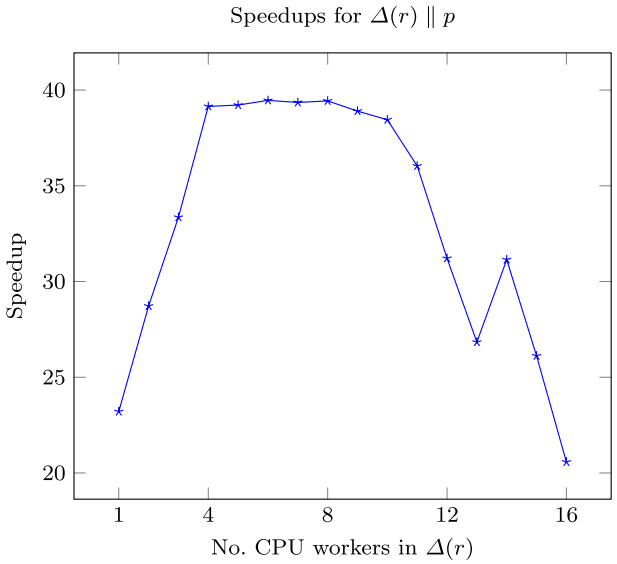

Fig. 4 Speedup graph for the Image Convolution configuration $\Delta(r) \parallel p$, where p is executed on a GPU

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 $T(r_{CPU}) = 0.2$ ms, $T(p_{CPU}) = 6.6$ ms, $T(p_{GPU}) = 0.08$ s. In Table 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

Table 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

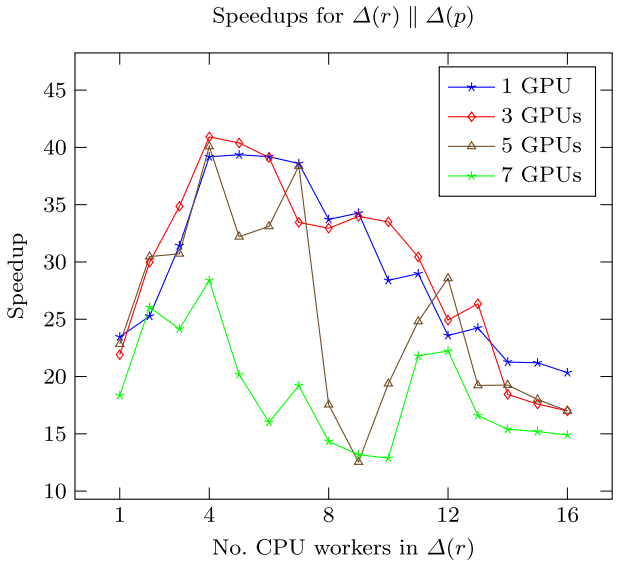

Fig. 5 Speedup figures for the image convolution configuration $\Delta(r) \parallel \Delta(p)$, with 0 CPU and a different number of GPU workers for $\Delta(p)$

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 Fig. 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 Fig. 5 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, 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, Fig. 6 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 Fig. 6 demonstrates, the best speedup obtained for

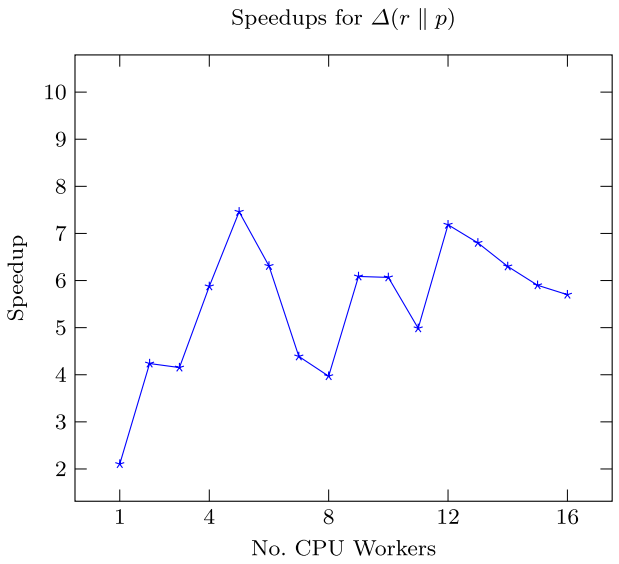

Fig. 6 Speedup figures for the image convolution configuration $\Delta(r \parallel p)$, where the number of GPU workers and the number of CPU workers for $\Delta(r \parallel p)$ are equal

this configuration is 7.45 for (5, 5) CPU and GPU workers for $\Delta(r \parallel p)$, confirming the prediction given by MCTS (Table 3).

5.2 Ant Colony Optimisation

Ant Colony Optimisation (ACO) [13] 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 (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 ith 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

Table 4 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

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

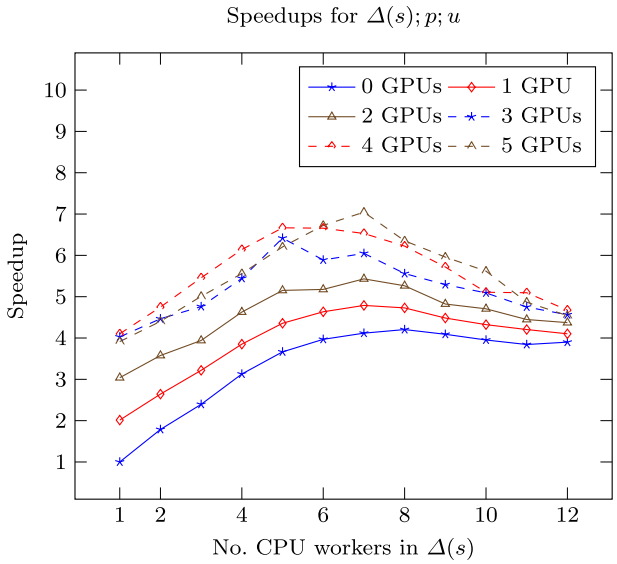

Fig. 7 Speedup graph for the ACO configuration $\Delta(s);p;u$

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.

5.2.2 Optimal Static Mapping Determined by MCTS

Table 4 shows the output of MCTS for the $\Delta(s);p;u$ configuration for the ACO example.

5.2.3 Evaluation of Skeleton Configurations

Figure 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 Table 4) 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 [5]. 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.

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: (1) how much work to offload onto the GPU (GPU workers), as

Table 5 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

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

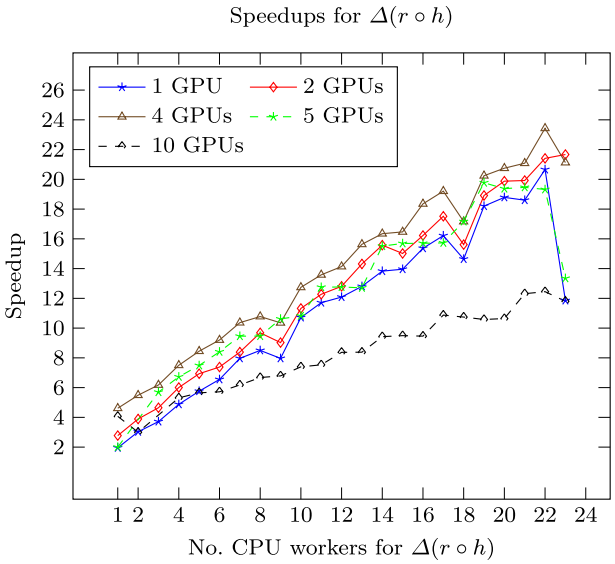

Fig. 8 Speedup graph for the Molecular Dynamics configuration $\Delta(r \parallel p)$

the CPU and the GPU can work on the farm concurrently; and, (2) how many CPU workers should be utilised.

5.3.2 Optimal Static Mapping Using MCTS

Table 5 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 8 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 Table 5 illustrates, the predicted mapping is (22, 1) (i.e., 22 CPU workers and 1 GPU worker). From Fig. 8, 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.

6 Related Work

Since the 1990s, the skeleton research community has been working on high-level languages and methods for parallel programming [11]. A rich set of skeleton rewriting rules, used to derive functionally equivalent programs that exploit different kinds of parallelism, has been proposed in [2, 4, 26]. Usually cost models are used to determine the best of a set of equivalent parallel programs. The technique presented in this paper 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 [16]. In [6, 7, 8], we introduced a parallel refactoring methodology for introducing and tuning skeletons in Erlang and C++ programs, respectively. However, unlike the technique proposed in this paper, both of these methodologies did not support heterogeneous architectures, or provide support for deriving mapping information.

There is an extensive body of work on mapping task, data and pipeline parallelism to parallel architectures providing static partitioning [20, 24, 27], using runtime scheduling [23], heuristic-based mappings [15], analytical models [21]. 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 [28] or the runtime adaptation approach used in FlexStream [17] 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 [25], Serban et al. use an analytic model to devise partitioning between CPUs and GPUs of the tasks from data-parallel computations in a heterogeneous computing settings. In [14] 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 [1] rather than orchestration. Monte Carlo Tree Search has classically been applied to challenging game playing, for example the GO and Bandit problem [12]. 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 technique that employs new refactoring and static mapping technology, and is based on algorithmic skeletons. The technique 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. 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, alternative, like exhaustive search over the parameter space, are also possible, as we have shown in Sect. 5 to verify our MCTS predictions. Our technique 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. intention, in time, to develop a generic refactoring and mapping technique capable of using a common set of refactoring rules and skeletons.

In the future, we will extend our technique to cover a wide range of parallel skeletons including parallel workpools, divide-and-conquer, map-reduce and other domain-specific parallel patterns, such as parallel orbit enumerations. In addition, we intend to demonstrate the use of our technique on a further set of case studies, showing greater skeleton nesting and heterogeneity.

Acknowledgements This work was supported by the EU Horizon 2020 project, TeamPlay https://www.teamplay-h2020.eu, Grant Number 779882, and UK EPSRC Discovery, Grant Number EP/P020631/1.

References

1. Agrawal, S., Thies, W., Amarasinghe, S.: Optimizing stream programs using linear state space analysis. In: Proceedings of CASES '05, pp. 126–136. ACM (2005)
2. Aldinucci, M., Coppola, M., Danelutto, M.:Rewriting skeleton programs: how to evaluate the data-parallel stream-parallel tradeoff. In: CMPP, pp. 44–58. Germany (May 1998)
3. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with fastflow. In: Proceedings of Euro-Par '11, pp 170–181 (2011)
4. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: the FAN skeleton framework. Parallel Algorithm Appl. **16**(2–3), 87–121 (2001)
5. Allen, Michael P.: Introduction to molecular dynamics simulation. Comput. Soft Matter Synth. Polym. Proteins **23**, 1–28 (2004)
6. Brown, C., Hammond, K., Danelutto, M., Kilpatrick, P., Elliott, A.: Cost-directed refactoring for parallel erlang programs. In: International journal of parallel programming. Springer (2013)
7. Brown, C., Janjic, V., Hammond, K., Schöner, H., Idrees, K., Glass, C.: Agricultral reform: more efficient farming using advanced parallel refactoring tools. In: Proceedings of PDP 2014, Euromicro (2014)
8. Brown, C.: D4.4 Final Pattern Transformation System from the ParaPhrase Project. http://paraphrase-enlarged.elte.hu/downloads/D4-4.pdf University of St. Andrews, Scotland, UK, (2011)

9. Browne, C.: A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **1**(2), 1–43 (2012)
10. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. J. ACM **24**(1), 44–67 (1977)
11. Cole, M.: Algorithmic skeletons: structured management of parallel computations. In: Research Monographs in Parallel and Distributed Computing. MIT Press (1989)
12. Coulom, R.: Efficient selectivity and backup operators in Monte–Carlo tree search. In: Computers and Games, pp. 72–83 (2007)
13. den Besten, M., Stuetzle, T., Dorigo, M.: Ant colony optimization for the total weighted tardiness problem. PPSN **6**, 611–620 (2000)
14. Goli, M., McCall, J., Brown, C., Janjic, V., Hammond, K.: Using machine learning to derive mappings for heterogeneous parallel computations. In: Proceedings of CEC. IEEE (2013)
15. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: ACM SIGOPS Operating Systems Review, vol. 40, pp. 151–162. ACM (2006)
16. Hammond, K., Aldinucci, M., Brown, C., Cesarini, F., Danelutto, M., Gonzalez-Velez, H., Kilpatrick, P., Keller, R., Natschlager, T., Shainer, G.: The ParaPhrase project: parallel patterns for adaptive heterogeneous multicore systems. In: FMCO (2012)
17. Hormati, A.H., Choi, Y., Kudlur, M., Rabbah, R., Mudge, T., Mahlke, S.: Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In: 18th International Conference on Parallel Architectures and Compilation Techniques, pp. 214–223 (2009)
18. Janjic, V., Brown, C., Hammond, K.: Lapedo: Hybrid skeletons for programming heterogeneous multicore machines in erlang. In: Parallel Computing: On the Road to Exascale. Advances in Parallel Computing, vol. 27, pp. 185–195. IOS Press (2015). https://doi.org/10.3233/978-1-61499-621-7-185
19. Kocsis, L., Szepesvári, C., Willemson, J.: Improved Monte–Carlo Search University Technical Report, Tartu, Estonia (2006)
20. Kwok, Y.-K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. (CSUR) **31**(4), 406–471 (1999)
21. Navarro, A., Asenjo, R., Tabik, S., Cascaval, C.: Analytical modeling of pipeline parallelism. In: 18th International Conference on Parallel Architectures and Compilation Techniques, pp. 281–290 (2009)
22. Opdyke, W.: Refactoring object-oriented frameworks. PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL, USA (1992)
23. Ramamritham, K.: Dynamic task scheduling in hard real-time distributed systems. J. Softw. IEEE **1**(3), 65–75 (1984)
24. Saraswat, V.A., Sarkar, V., von Praun, C.: X10: Concurrent programming for modern architectures. In: Proceedings of PPOPP '07, pp 271–271. ACM (2007)
25. Serban, T., Danelutto, M., Kilpatrick, P.: Autonomic scheduling of tasks from data parallel patterns to CPU/GPU core mixes. Proc. HPCS **2013**, 72–79 (2013)
26. Skillicorn, D.B., Cai, W.: A cost calculus for parallel functional programming. J. Parallel Distrib. Comput. **28**(1), 65–83 (1995)
27. Subhlok, J., Stichnoth, J.M., O'Hallaron, D.R., Gross, T.: Exploiting task and data parallelism on a multicomputer. ACM SIGPLAN Notices **28**(7), 13–22 (1993)
28. Wang, Z., O'Boyle, M. F.: Partitioning streaming parallelism for multi-cores: a machine learning based approach. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, pp. 307–318. ACM (2010)