AperTO - Archivio Istituzionale Open Access dell'Università di Torino

An Abstract Annotation Model for Skeletons

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1508808 since 2016-10-16T15:41:14Z

Publisher:

Springer

Published version:

DOI:10.1007/978-3-642-35887-6_14

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

29 September 2021

An abstract annotation model for skeletons

Marco Aldinucci[1], Sonia Campa[2], Peter Kilpatrick[3], Fabio Tordini[1], and Massimo Torquati[2]

[1] Computer Science Department, University of Torino, Italy
{aldinuc,fabio.tordini}@di.unito.it
[2] Computer Science Department, University of Pisa, Italy
{campa,torquati}@di.unipi.it
[3] Computer Science Department, Queen's University Belfast, UK
p.kilpatrick@qub.ac.uk

Abstract. Multi-core and many-core platforms are becoming increasingly heterogeneous and asymmetric. This significantly increases the porting and tuning effort required for parallel codes, which in turn often leads to a growing gap between peak machine power and actual application performance. In this work a first step toward the automated optimization of high level skeleton-based parallel code is discussed. The paper presents an abstract annotation model for skeleton programs aimed at formally describing suitable mapping of parallel activities on a high-level platform representation. The derived mapping and scheduling strategies are used to generate optimized run-time code.

1 Introduction

One central challenge of parallel programming today is to achieve performance portability across a range of architectures. Most application programs are currently written at the low level of C or Fortran, combined with a communication library such as MPI; moreover, they are often tuned toward one specific machine configuration. Since parallel computers are typically replaced within five years, parallel programs which have a longer life span have to be re-tuned or redesigned. In addition, programming at this low level of abstraction is cumbersome and error-prone. Recent trends in platform design exacerbate the problem: platforms are increasingly heterogeneous, e.g. including many general-purpose and specialized cores, parallel accelerators (GPUs), soft cores (FPGAs). As a consequence, even the development and tuning of applications for a specific machine configuration is complex and time consuming.

In sequential programming, the problem of having to recode for different machines was apparent three decades ago. The software engineering solution to this issue was to introduce levels of abstraction, effectively yielding a tree of refinements, from the problem specification to alternative target programs [1]. The derivation of a target program then follows a path down this tree. The transition from one node to the next can be described formally by a semantics-preserving program transformation or refinement. Conceptually, porting a program to a

different machine configuration means backtracking to a previous node on the path and then following another path to a different target program.

This approach is not yet popular in the parallel programming setting. For example, typically parallel accelerators such as GPUs are programmed by directly leveraging on low-level accelerator-specific APIs (e.g. NVidia CUDA and OpenCL). Although these programming frameworks have been designed to keep narrow the gap between CPU and GPU programming style, there are still several differences, many of them emanating from the different nature of the architecture and even from the different models of computation of the GPUs. For example, when dealing with GPUs the programmer finds that all hardware facilities that are traditionally used to simplify the programming model have been removed (e.g. cache-coherence, branch prediction, virtual memory, global synchronizations) and so he/she must use very low-level mechanisms and must take into account a range of board specific information in order to obtain acceptable performance (e.g. local memory size, correct memory alignment, number of context, memory interleaving, etc.). Furthermore, the selection of which parts of an application should be executed on the GPU is completely the responsibility of the programmer and even if the code can be easily identified, there is no guarantee that it will be faster on the GPU than on a CPU. The programmer also has to manage data movement between the host processor's main shared memory and the GPU's core local memory taking care of memory alignment. Therefore porting code to GPUs, or developing from scratch an efficient code for GPUs, is not an easy task and can be a huge drain on resources. The typical code optimization curve grows very slowly and requires lots of performance testing and tuning, especially in industrial contexts where standard procedures for accurate testing and validation have to be performed.

Since the nineties, the "skeletons" research community [2] has been working on high-level languages and methods for parallel programming [3–6]. Skeleton programming requires the programmer to write a program using well-defined abstractions (called skeletons) derived from higher-order functions that can be parameterized to execute problem-specific code. Skeletons are parallel ab-initio and do not expose to the programmer the complexity of concurrent code, for example synchronization, mutual exclusion and communication. They instead specify abstractly common patterns of parallelism – typically in the form of parametric orchestration patterns – which can be used as program building blocks, and can be composed or nested like constructs of a programming language. A typical skeleton set includes the pipeline, the task farm, reduction and scan. For a given skeleton, usually, many efficient implementations for a given target platform may exist. Skeletons exhibit well-defined functional semantics, i.e. *what is computed*. As they describe parallelism exploitation paradigms, they also exhibit extra-functional behaviour, i.e. *how results are computed* [7], which can be also expressed by different realizations of the same pattern. For example, the functional composition operator $\circ$ can be interpreted as *pipeline* or as *sequence of functions*. We believe that the patterns/skeletons approach, which has been demonstrated to be effective for multi-core platforms (e.g. TBB [8] and Fastflow

[9] among others) can be used also with heterogeneous architectures to obtain a good trade-off between performance and code portability.

After incubation for over two decades in a quite restricted research community, skeletons gained renewed popularity with the arrival of multi-core platforms, the consequent diffusion of parallel programming frameworks, and their adoption in some successful programming frameworks, such as Intel Threading Building Block (TBB) [8]. Despite being complex to program, current multicores are almost uniform machines and in many cases they can be programmed with decent performance as if they were symmetric multiprocessors. However, this uniformity is progressively decaying with each new generation of machine: the current generation of multi-cores exhibit non-uniform memory access (typically cc-NUMA, i.e. cache-coherent Non Uniform Memory Access), while the next generation (e.g. IBM PowerEN, Intel MIC) will have specialized cores and accelerators to gain peak performance on critical tasks, and a non-uniform connection latency among cores and memory modules.

The heterogeneity and reduced connectivity of forthcoming platforms will make it all the more important for a programming framework to have the ability to generate parallel code according to different orchestration patterns and to map them on to different platforms in such a way that each task is run in the best suited executor and the synchronization and communication patterns are efficiently supported by the targeted platform. Currently, this activity is largely left to programmer expertise and is not effectively supported by development tools.

This work aims to make a step toward the formalization and the automation of this process. In particular, program refinement is proposed as an abstract tool to deal with the problem of the mapping of parallel activities onto heterogeneous cores (i.e. CPUs and GPUs). These parallel activities are assumed to be automatically generated by a high-level skeletal programming framework. In particular, skeletons are annotated with mapping information along a process of refinements. The first step annotates the tree with functional and extra-functional information such as data access, data dependencies, parallelism degree and so on; the next step maps the annotated tree on the given platform, taking into account the underlying target architecture; the last step executes the mapped tree and constantly notifies performance data to the upper levels so that, in case of performance degradation and driven by a suitable a performance model, the skeleton tree can be rewritten in a functional equivalent but better performing one. This is envisaged as the first step on the path to automated optimization of parallel codes onto heterogeneous platforms. In this respect, important milestones will include definition of a complete set of attributes to abstractly describe the key features of a specific parallel architecture, and the definition of suitable performance models able to drive the optimization process across the tree of refinements. These activities are currently ongoing in the ParaPhrase EC-STREP project.

The remainder of the paper is structured as follows: Sec. 2 introduces typical parallel programming patterns, while Sec. 3 introduces the abstract annotation

model which drives the mapping and rewriting of a user program refactored as a skeleton program. Sec. 4 provides some preliminary results of our approach implementation obtained on a heterogeneous platform. Finally, Sec. 5 discusses related work and Sec. 6 concludes the paper.

2 Parallelism Paradigms and Patterns

Attempts to reduce programming effort by raising the level of abstraction date back at least three decades. Notable results have been achieved by the *skeletal* approach [2, 10, 11], enabling *pattern-based* parallel programming. This approach appears to be becoming increasingly popular after reinforcement by several successful parallel programming frameworks [12–15].

Algorithmic skeletons capture common parallel programming paradigms (e.g. ForAll, MapReduce, Divide&Conquer, etc.) and make them available to the programmer as high-level programming constructs equipped with well-defined functional and extra-functional semantics [7]. Ideally, algorithmic skeletons address the difficult problems of parallel programming (i.e. concurrency exploitation, orchestration, mapping, tuning) moving them from the application design to development tools by capturing and abstracting the common paradigms of parallel programming and providing them with efficient implementations, i.e. a toolkit of code generation techniques and a pre-optimized run-time support.

Differences between algorithmic skeletons and parallel design patterns lie mainly in the motivations leading to these two apparently distinct concepts and in the research environments where they have been developed: the parallel programming community for algorithmic skeletons and the software engineering community for parallel design patterns. As far this work is concerned, the two concepts can be seen as synonymous.

Traditionally, in skeletal (and parallel pattern-based) programming the computation is organized according to application-independent high-level paradigms, which are usually categorized in three classes:

1. *Data Parallelism* is a method for parallelizing a single task by processing independent data elements in parallel. Data parallelism also supports loop-level parallelism where successive iterations of a loop working on independent or read-only data are parallelized in different flows-of-control and concurrently executed. *map* and *reduce* are instances of data parallelism.
2. *Task Parallelism* consists of running the same or different code on different executors (cores, machines, etc.). Task parallelism is usually explicit in the algorithm. Different flows-of-control (threads, processes, etc.) may communicate with one another as they work. Communication usually takes place to pass data from one thread/process to one or many others. The *farm* is a typical representation of such class of patterns
3. *Stream Parallelism* consists in the parallel processing of different items of a data stream, which can be either the input data or generated by the application's internal programming mechanisms (e.g. via asynchronous function

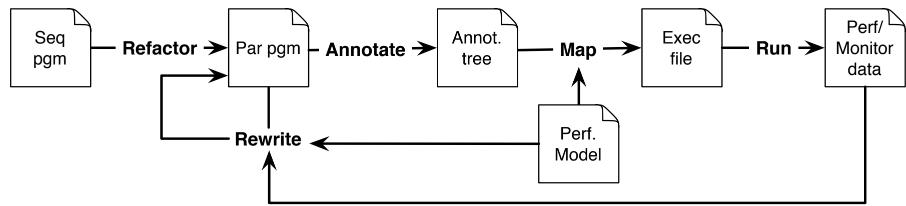


Fig. 1: Sketch of the proposed approach

calls). It can be used when there exists a partial or total order in a computation; the *pipeline* is a paradigmatic stream parallel pattern.

Pragmatically, a given computational problem typically admits several algorithmic solutions exploiting patterns in different classes, or different compositions of them. In addition, in many cases, patterns in different classes can simulate one another. The extent of this generality is dependent on the set of patterns provided by a specific framework, which can also be designed to target one or more application scenarios [15].

After Cole's seminal work [2], early proposals for skeletal programming frameworks have focused mainly on distributed memory platforms (e.g. clusters of workstations, grid); some of them, e.g. Google's MapReduce [13], have evolved in mainstream programming tools [16]. Recent proposals, following the platform architecture trend, have shifted the focus to include multi-cores and the shared address model; in addition to academic initiatives such as FastFlow (Sec. 4), it is worth mentioning consolidated industrial products such as the Intel Threading Building Block (TBB) library [8] and, to a limited extent, the Microsoft Task Parallel Library [17].

More recently, the skeletal approach has been proposed for GPGPUs and hybrid architectures: the SkePU framework is an example [18].

Some of these skeleton frameworks explicitly include stream parallelism as a major source of concurrency exploitation [12, 7, 14, 8]: rather than allowing programmers to connect stages into arbitrary graphs, basic forms of stream parallelism are provided to the programmer in high-level constructs such as *pipeline* (modelling computations in stages), *farm* (modelling parallel computation of independent data tasks), and *loop* (supporting generation of cycles in a stream graph and typically used in combination with a farm body to model Divide&Conquer computations).

3 A refinement process for skeletons

3.1 Approach overview

Our approach to parallelization via skeletons is depicted in Fig. 1. The starting point of the refinement process is a sequential program in which the user (or

eventually a tool) detects those parts of the code which can be parallelized. Parallelism can be introduced by a tool-assisted *Refactoring* process in which the user identifies patterns that can be captured by high level constructs (or calls to libraries) taking sequential code as parameter(s). This Refactoring process results in a high level program written as a composition of patterns/skeletons, i.e. a skeleton tree. The remainder of the development involves successively refining this skeleton through a series of stages to an implementation on a target architecture.

The *Annotate* phase uses a set of annotation rules to annotate the skeleton tree with an abstract description of the target architecture incorporating information such as number CPUs, number of GPUs, etc. In essence this annotated tree represents a set of possible mappings of tree to architecture.

The *Map* phase specializes the set of mappings implicit in the annotation tree to a particular mapping of components to resources. It uses more detailed target architecture specific detail (such as bandwidth of connections, speed of processors, etc.) and is informed by a performance model [19, 20] which allows qualitative assessment of alternative mapping strategies. The mapping phase produces an execution file that will be used by the architecture level for actively running the application (*Run* phase).

In addition to the above process of deriving an initial running program, one can envisage also a *Rewrite* phase which allows restructuring of the program as a result of feedback obtained from the running program. This Rewrite phase restructures the program in accordance with well-know functional equivalences between parallel patterns, again informed by the performance model. The result is a new (functionally equivalent) skeleton tree and so a new annotate phase can be commenced.

In the following section we will give a formal representation of the skeletons included in our semantic framework in order to define the *Annotate* and *Map* phases which are the focus of the current paper.

3.2 Skeleton definition

Data and stream parallelism can be conveniently expressed using high-level patterns with well-defined functional semantics [21, 7, 22], whereas task parallelism, in the most general form, often subsumes low-level parallelism exploitation where synchronizations (as well as functional semantics) are deeply interwoven in the business code. For this reason, usually, they are not embedded in high-level pattern-based programming frameworks. In the following we use a generic two-tier pattern-based programming language including stream and data parallelism. Data parallel patterns can be nested within stream parallel patterns, but not vice-versa.

Let $\mathcal{P}$ be a pattern-based program, and $\mathcal{P}_{nc}$ a *non-cyclic* pattern-based program, i.e. a program not exhibiting cyclic data-dependencies among patterns. Let $\mathcal{P}_{sp}$ and $\mathcal{P}_{dp}$ be stream and data parallel high-level patterns, respectively which can be composed as follows:

$$
\begin{aligned}
\mathcal{P} \;\; &::= \mathcal{P}_{nc} \mid parloop(\mathcal{P}_{nc}, E) \\
\mathcal{P}_{nc} &::= \mathcal{P}_{sp} \mid \mathcal{P}_{dp} \\
\mathcal{P}_{sp} &::= \mathcal{P}_{nc} \circ \mathcal{P}_{nc} \mid farm(\mathcal{P}) \\
\mathcal{P}_{dp} &::= map(Seq) \mid reduce(Seq) \mid Seq \\
Seq &::= \langle seq\ code \rangle \\
E \;\; &::= \langle seq\ expression \rangle
\end{aligned}
$$

Here, for the sake of simplicity, the iterative usage of skeletons via the *loop* pattern, which can also be used to implement Divide&Conquer, is limited to the top level in order to simplify skeleton composition. Notice that in the most general case the *loop* pattern, if nested within other patterns, can receive data items from two different streams (*input* and *feedback* streams) and this requires proper management of non-determinism among them to avoid deadlock.

Patterns are assumed to exhibit a pure functional semantics, i.e. they can be defined as higher-order functions fully determined by their input-output behavior. As happens in the FastFlow framework [9], the approach can be extended to higher-order functions exhibiting a shared state. For example, using Ocaml-like notation to define the functional behavior, farm and pipeline skeletons can be described as follows:

```
let farm f x = (f x);;
let pipeline f g x = (g(f x));;
let map f x = Array.map f x;;
...
```

where streams, i.e. a (finite or infinite) sequence of values of the same type, are represented as lists. Patterns working on streams can be modelled accordingly, e.g.

```
let stream_parallel f x::y = (f x)::(stream_parallel f y);;
```

In $\mathcal{P}_{sp}$ the stream items are potentially computed in parallel. As an example, the farm skeleton uses a set of independent processing elements to compute the input tasks. Each time a new input task is available one of these resources is selected for the execution of the task, possibly using some kind of auto scheduling policy. The pipeline skeleton uses independent processing elements to compute the different stages in such a way that computation of stage i relative to task j can proceed concurrently (in parallel) with both the computation of stage $i-1$ for task $j+1$ and the computation of stage $i+1$ for task $j-1$.
On the other hand, in $\mathcal{P}_{dp}$ the parallel computation is applied to the input data as a whole. As an example, the map skeleton splits the input data collection into chunks on the basis of different policies and the same function is applied in parallel to each chunk by a different executor.

3.3 Skeleton rewriting

As already mentioned, a skeleton is often defined by a functional semantics (*what is computed*) and a non-functional semantics (*how results are computed*) and it is useful to make distinction, even informally, between them. Examples of a formal definition of (functional and non-functional) semantics for parallel patterns and streams can be found in [7, 22].

The functional semantics allows programmers to "compute" the function denoted by a pattern-based program. It also allows reasoning about program equivalence, in terms of the results computed, or to define semantics-preserving program transformations [21, 23]. These transformations can also be driven by some kind of analytical performance model associated with patterns, in such a way that only those rewritings leading to efficient implementations of the pattern are considered [24, 25, 21]. For instance, one can easily determine that the following two programs actually compute the same result, even if they exhibit different parallel behaviors:

```
let progA f g = stream_parallel (pipeline f g);;
let progB f g = stream_parallel (farm (pipeline f g));;
```

Also, streaming patterns can be *normalized* by reducing nesting of any depth of $farm$ and $\circ$ (i.e. *pipeline* in this context) to a $farm(pipeline()))$ [25].

Because patterns carry both a functional and non-functional semantics (thus the *intent* of the code [26]) they can also be used to support a generative approach to machine-specific run-time generation and optimization. For example, in the FastFlow framework (see Sec. 4), patterns are used to generate graphs of parallel activities and their orchestration in terms of (true) data dependencies.

We can refine this approach on a formal basis by defining a semantics allowing augmentation of the skeletal description provided by the application graph with mapping information and synchronization requirements with respect to the specific target architecture at hand. When the skeleton graph can be "rewritten" to a semantically equivalent one but enriched with information related to (potentially) optimal mapping, we can achieve better generation and optimization of the actual run-time to the specific machine at hand.

3.4 Annotation semantics

Preliminary notation. For the sake of simplicity, we will provide an abstraction of a target architecture including one CPU (i.e a set of cores) and one or more GPUs, although our approach can be easily extended to a number of CPUs and GPUs available in a system. We will denote the set of $n \geq 0$ cores on the same CPU as

$$CPU = \{core_0, core_1, \ldots, core_n\}$$

representing the set of available cores on a given CPU.

$$GPU = \{gpu_0, gpu_1, \ldots, gpu_k\}$$

represents the set of available GPUs on a given architecture ($k \geq 0$).
Moreover, we assume that given a skeleton P, the mapping of P onto a given architecture x ($x \in CPU$ or $x \in GPU$) is represented by the notation P_x; thus P_{core_1} will define the mapping of skeleton P locally onto $core_1$; P_{gpu_i} will define the mapping of P onto the *i-th* GPU available in the system; if P is a composite skeleton whose mapping could involve a set of computational resources $X = \{core_0, core_1\} \subseteq CPU$, then P_X will define the mapping between P and the sub-architecture represented by X.

Seq annotation. Our goal is to define an abstract semantics driving suitable mappings among (compositions of) skeletons as defined by Section 3 and the available abstract architectures at hand.
The base case is represented by the *Seq* skeleton, which will be simply mapped onto one of the cores available on the current CPU

$$\frac{x \in CPU = \{core_0, \ldots, core_n\}}{Seq \rightarrow Seq_x} \quad (1)$$

or, since it could be encapsulated by a data parallel skeleton, it can be mapped onto a GPU

$$\frac{x \in GPU = \{gpu_0, \ldots, gpu_n\}}{Seq \rightarrow Seq_x} \quad (2)$$

Farm annotation. Each instance of a farm will be rewritten in a notation highlighting the emitter (E) and the collector (C), in order to potentially allow different mappings of all the nodes composing such a skeleton. Thus, hold that

$$farm(P) = farm(E, P, C)$$

There are two possible configurations in mapping a farm: *i)* all the nodes are allocated on different cores of the same CPU:

$$\frac{E \rightarrow E_x \wedge P \rightarrow P_Y \wedge C \rightarrow C_z \wedge x, z \in CPU \wedge x \neq z \wedge Y \subseteq CPU - \{x, z\}}{farm(E, P, C) \rightarrow farm(E_x, P_Y, C_z)} \quad (3)$$

ii) emitter and collector are mapped onto different cores of the same CPU while the workers can be mapped onto a GPU

$$\frac{c_0, c_1 \in CPU \wedge X \subseteq GPU \wedge E \rightarrow E_{c_0} \wedge P \rightarrow P_X \wedge C \rightarrow C_{c_1}}{farm(E, P, C) \rightarrow farm(E_{c_0}, P_X, C_{c_1})} \quad (4)$$

With respect to such a rule we have to point out that in order to make suitable mappings, we should also take into account how the communication costs for moving data to and from the GPU influence the performance. In fact, placing the workers onto a GPU could be worthwhile if a huge set of tasks is ready to be delivered by the emitter for computation so that the workers can execute in a "dataparallel-like" mode on the set of input tasks; or, such mapping could be a good choice in those cases in which the task is, actually, a data parallel structure to be computed. Thus, while rule 4 is a good starting point for formalizing the mapping of workers onto a GPU, it needs to be further studied and enriched by data description details.

Parloop annotation. The parloop skeleton can be mapped to host the inner skeleton on any architecture while the condition is hosted on a CPU architecture:

$$\frac{x_2 \in CPU \wedge (X_1 \subseteq CPU \vee X_1 \subseteq GPU) \wedge P \rightarrow P_{X_1} \wedge E \rightarrow E_{X_2}}{parloop(P, E) \rightarrow parloop(P_{X_1}, E_{x_2})} \tag{5}$$

Since the evaluation of E defines whether the loop stops or continues to iterate, the rule above asserts that E is always evaluated on a CPU, while P (being a data parallel or a stream parallel skeleton) could be mapped onto a CPU or a GPU. Theoretically, if an iteration of P has been evaluated on x_i, the system memory of that node could still provide an up-to-date representation of data needed to proceed in the computation.

Map/Reduce annotation. Map and reduce can both be mapped on a CPU or a GPU architecture

$$\frac{x \subseteq GPU \vee x \subseteq CPU \wedge Seq \rightarrow Seq_x}{map(Seq) \rightarrow map(Seq)_x} \tag{6}$$

$$\frac{x \subseteq GPU \vee x \subseteq CPU \wedge Seq \rightarrow Seq_x}{reduce(Seq) \rightarrow reduce(Seq)_x} \tag{7}$$

Pipeline annotation. How a pipeline will be mapped depends at first on whether its stages are represented by stream parallel or data parallel skeletons, i.e. how data will flow through the graph and which dependencies among them are exploited. In the former case, the stages have to be placed on different cores (in order to exploit parallelism), but possibly of the same CPU (in order to minimize stream transfer costs):

$$\frac{x \neq y \wedge x, y \in CPU \wedge P' \rightarrow P'_x \wedge P'' \rightarrow P''_y \wedge P', P'' \in P_{sp}}{P' \circ P'' \rightarrow P'_x \circ P''_y} \tag{8}$$

In the latter case, a pipeline of data parallel skeletons can be mapped onto different cores (for instance, one stage per core) or onto different GPUs

$$\frac{x \neq y \wedge (x, y \in CPU \vee x, y \in GPU) \wedge P' \rightarrow P'_x \wedge P'' \rightarrow P''_y \wedge P', P'' \in P_{dp}}{P' \circ P'' \rightarrow P'_x \circ P''_y} \tag{9}$$

However, the pipeline of two data parallel stages could imply some synchronization steps between stages in the event of functional dependencies. For this reason, if the system provides just one GPU, the pure functional pipelining of two or more data parallel skeletons has to be rewritten in terms of a composition of stages (denoted by ";") because of the presence of some synchronization points that can serialize the execution.

$$\frac{P' \rightarrow P'_{gpu} \wedge P'' \rightarrow P''_{gpu} \wedge gpu \in GPU}{P' \circ P'' \rightarrow P'_{gpu}; P''_{gpu}} \tag{10}$$

Comp annotation. *Comp* is a skeleton (represented by ";" syntax) defining the sequential composition of two sub-skeletons which will be executed sequentially. This pattern is particularly useful in those rewritings in which part of a skeleton tree has to "collapse" into a sequential piece of code to provide improved performance, for example, in terms of communication costs.

$$\frac{x = y}{P'; P'' \rightarrow P'_x; P''_y} \tag{11}$$

The composed skeletons are mapped both on to the same target node.

3.5 Mapping strategies

Data parallelism onto heterogeneous architectures. Let us assume we have the skeleton composition

$$map(Seq_1) \circ map(Seq_2)$$

Which suitable mappings can be provided, if the abstract target architecture is represented by the system $S = \{cpu_0, cpu_1, gpu_0\}$ where $CPU = \{cpu_0, cpu_1\}$ and $GPU = \{gpu_0\}$ and $\sharp GPU = 1$ represents the cardinality of the GPU set? As skeletons, Seq_1 and Seq_2 can be indifferently placed onto a CPU or a GPU architecture (rules 1 and 2), and so two branches are possible for the mapping of the two outer maps, since it holds that

$$\frac{Seq_1 \rightarrow (Seq_1)_x \wedge \forall x.x \in S}{map(Seq_1) \rightarrow map(Seq_1)_x}$$

and

$$\frac{Seq_2 \rightarrow (Seq_2)_x \wedge \forall x.x \in S}{map(Seq_2) \rightarrow map(Seq_2)_x}$$

However, at a higher level of the skeleton graph, the maps are composed by a *pipeline* operation. Recalling that our system provides just one GPU and two cores, we have two different options: *i)* we could place the pipeline on the same GPU but in a compositional manner so that they can eventually communicate via a shared memory system, i.e. by applying rule 10

$$\frac{map(Seq_1) \rightarrow map(Seq_1)_{gpu} \wedge map(Seq_2) \rightarrow map(Seq_2)_{gpu}}{map(Seq_1) \circ map(Seq_2) \rightarrow map(Seq_1)_{gpu}; map(Seq_2)_{gpu}}$$

ii) we could place the pipeline so that the first map is executed on cpu_1 and the second one on cpu_2 and the stages will then communicate via a stream of data, thus applying rule 9.

$$\frac{cpu_0, cpu_1 \in CPU \wedge map(Seq_1) \rightarrow map(Seq_1)_{cpu_0} \wedge map(Seq_2) \rightarrow map(Seq_2)_{cpu_1}}{map(Seq_1) \circ map(Seq_2) \rightarrow map(Seq_1)_{cpu_0} \circ map(Seq_2)_{cpu_1}}$$

Which of these two options will be actually chosen will depend on the ability of the system to make predictions on the cost of each configuration. Providing the semantics with a cost model allowing an estimation of each candidate configuration will be the goal of future work.

noise	Seq (1 CPU)	8 cores + 24 CPUs	8 cores + 1 GPUs
10 %	32.0 s	1.8 s	1.9 s
50 %	162.1 s	6.5 s	2.3 s
90 %	290.0 s	10.9 s	2.8 s

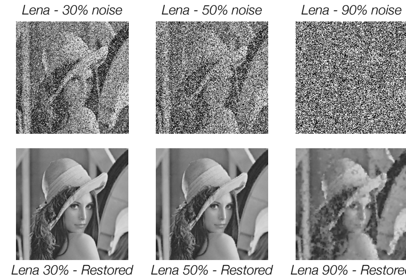


Fig. 2: Left) Execution time of different configuration of the *Detect+Restore* functions on Lena image. Right) Restoration result with PSNR (Peak Signal-to-Noise Ratio) and MAE (Mean Absolute Error).

Bringing down data transfer costs. Let now assume that we have the following skeleton composition

$$Seq1 \circ Seq2 \circ Seq3$$

From a functional perspective the *pipeline* operation exploits the associative properties so that

$$(Seq1 \circ Seq2) \circ Seq3 \equiv Seq1 \circ (Seq2 \circ Seq3)$$

However, from a mapping point of view, these two options could imply very different performance effects: for example, let us suppose that a number of dual-core CPUs are available so that $CPU_0 = \{core_0, core_1\}$ and $CPU_1 = \{core_2, core_3\}$: the better mappings are those assigning cores belonging to the same CPU to (possibly) contiguous stages. Thus, while

$$(Seq1_{core_0} \circ Seq2_{core_1}) \circ Seq3_{core_2}$$

$$(Seq1_{core_1} \circ Seq2_{core_0}) \circ Seq3_{core_2}$$

$$Seq1_{core_1} \circ (Seq2_{core_2} \circ Seq3_{core_3})$$

$$Seq1_{core_0} \circ (Seq2_{core_2} \circ Seq3_{core_3})$$

would be good combinations since they minimize the extra-CPU communication to just one occurrence, all the other combinations, such as for example

$$Seq1_{core_0} \circ (Seq2_{core_2} \circ Seq3_{core_1})$$

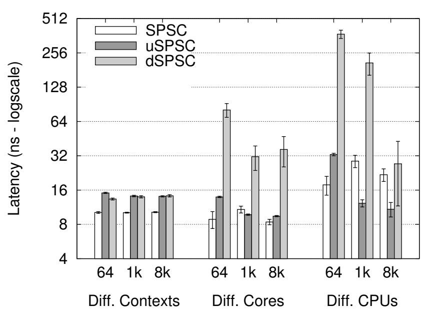
will need two extra-CPU communications, maybe accessing a shared memory or even across the network in the case of a cluster. In Sec. 4 we will see a concrete instantiation of this principle applied to a specific architecture.

4 Preliminary results

In the current section we will exemplify the proposed methodology through some examples implemented on top of FastFlow, a parallel programming framework aimed at simplifying the development of applications for multi-core platforms, whether these applications are brand new or ports of existing legacy codes [27]. FastFlow promotes pattern-based programming and has been specifically designed to efficiently support fine-grained parallel computations. The FastFlow patterns can be arbitrarily nested to model increasingly complex parallelism exploitation patterns. The FastFlow implementation guarantees an efficient execution of the skeletons on currently available multi-core systems by building the skeletons themselves on top of a library of lock-free producer/consumer queues. The workstation on which we performed the tests is a "homogeneous" Intel Nehalem microarchitecture equipped with 4 eight-core double context Xeon E7-4820 @2.0GHz with 18MB L3 shared cache, 256K L2, and 24 GBytes of main memory with Linux x86_64.

Current multi-core machines, such as Intel or AMD multi-core platforms are typically programmed and managed as if they were symmetric multiprocessors. However, the relation between performance and mapping of parallel activities onto core can be easily shown. For example, Fig. 3 reports the latency of three different implementations of the FastFlow Single-Producer Single-Consumer queue on the tested platform: a bounded array-based queue (SPSC), a dynamically linked-list queue (dSPSC) and an unbounded array-based queue (uSPSC). All implementations are lock-free and particularly optimized to avoid cache invalidations [28].The queue implementations are compared on three different mappings for the producer (P) and the consumer (C): 1) P and C are placed on two different hardware contexts of the same core; 2) P and C are placed on two different cores of the same socket; 3) P and C are placed on two different sockets. As can be seen, the dSPSC queue is particularly sensitive to mapping as the latency from one mapping to another changes the performance more than two orders of magnitude. This gap is expected to grow in forthcoming platforms with increasing core count and platform heterogeneity.

In the Table of Fig. 3, we report the performance obtained when running a very simple benchmark test where one 3 stage pipeline computes a stream of 1M tasks (double elements). Each stage is connected with the previous and following one (if present) using the FastFlow dSPSC unbounded queue. The first stage mainly generates the stream of tasks whereas the other two stages apply on each input a function computing a trigonometric computation. The third stage of the pipeline (s3) is the most computationally demanding. In this test we consider 4 possible mapping strategies for the three stages on the considered architecture. The best performance is obtained when the first 2 stages are mapped on the same core (different contexts) of the first CPU and the third stage is mapped on the second CPU (mappingC in the Table). In this way we are able to obtain a good trade-off between communication costs and computation. In fact, the first and the second stage do not interfere too much when placed on the same context since the first stage does not perform any significant numerical computation; instead,



mapping strategies	Compl. Time (ms)
mappingA	360
mappingB	530
mappingC	295
mappingD	480

– mappingA) s1,s2,s3 on adjacent cores of CPU1;
– mappingB) s1 on CPU1, s2 on CPU2 and s3 on CPU3;
– mappingC) s1 and s2 on the same context of one core of CPU1 and s3 on CPU2;
– mappingD) s2 and s3 on the same context of one core of CPU1 and s1 on CPU2.

Fig. 3: Left) Latency of 3 different implementations of FastFlow queues tested with three different mapping for the Producer and the Consumer threads [27]. Right) Performance obtained for the 3-stage pipeline(s1,s2,s3) benchmark varying the stage mapping.

they are able to benefit from the lower level cache to increase communication performance.

In order to validate the proposed methodology we describe a prototypical example, an image restoration application. The edge-preserving denoiser is a two-step filter for removing salt-and-pepper noise (see Fig. 2). In the first step, an adaptive median filter is used to identify the set of noisy pixels; in the second step, these pixels are restored according to an iterative variational approach up to convergence. The detailed description of the sequential algorithm is beyond the scope of this paper; it ensures state-of-the-art restoration quality and execution time, and is able to restore also very noisy images (e.g. up to 90% random noisy pixels) [29]. The same algorithm can also be used to restore video streams by iterating frame-by-frame the detect-denoise filters.

Pattern/Skeleton selection. Let $ReadImg, Detect, Restore, WriteImg, Fixpoint$ be chunks of sequential code (e.g. functions, i.e. *Seq*). The core of the edge-preserving denoiser can be sketched as

```
Img=ReadImg;
NoisySet=Detect(Img);
while(!Fixpoint(MAE((Img)){Img=Restore(Img,NoisySet);}
WriteImg(Img);
```

which can be iterated in a loop to realize a video version that simply repeats the same process on successive video frames. Notice that the *Restore* process is iterated up to *fixpoint* times by way of the $Fixpoint$ function. The *fixpoint* is reached when the restoration process brings no improvement in the "quality" of the image across two successive iteration. The quality of the image is usually measured in term of *Peak Signal-to-Noise Ratio* (PSNR) or *Mean Absolute Error*

(MAE). The video version can be sketched as follows:

```
while(true){
  Img=ReadImg;
  NoisySet=Detect(Img);
  while(!Fixpoint(MAE(Img))){Img=Restore(Img,NoisySet);}
  WriteImg(Img);
}
```

where the two filters *Detect* and *Denoise* are both executed sequentially and successively. Again, the latter filter is iterated up to *fixpoint* times. In order to detect when the *fixpoint* value is reached, the MAE filter has to be computed at each iteration. Computing MAE requires the analysis of the whole image `Img`. The visual effect on a noisy image of the two filters is shown in Fig. 2 right), together with the quality measures obtained (PSNR and MAE). The two principal filters can be parallelized in a data-parallel fashion using the *map* pattern, as follows:

```
while(true){
  Img=ReadImg;
  NoisySet=map(Detect(Img));
  while(!Fixpoint(MAE(Img))){Img=map(Restore(Img,NoisySet));}
  WriteImg(Img);
}
```

The MAE computation can be also parallelized in a data-parallel fashion using the *reduce* pattern. In addition, the parallelized versions of *Restore* and MAE can be composed and executed in a parallel loop in such a way that the whole restoration loop can be wrapped and possibly offloaded to an accelerator.

```
while(true){
  Img=ReadImg;
  NoisySet=map(Detect(Img));
  parloop((Fix=reduce(MAE,Img))∘(Img=map(Restore(Img,NoisySet))),!Fix);
  WriteImg(Img);
}
```

Annotate. From a semantic perspective and by following the syntax presented in this proposal, the preceding piece of code can be represented as

$$map(Seq_D) \circ parloop((reduce(Seq_M) \circ map(Seq_R)), E)$$

where we assume $Seq(Detect(Img)) = Seq_D$, $Seq(Restore(Img, NoisySet)) = Seq_R$, $Seq(MAE) = Seq_M$ and $E =!Fix$; our set of rules is then able to derive for us the annotation of the syntax tree associated to this composite skeleton as follows:

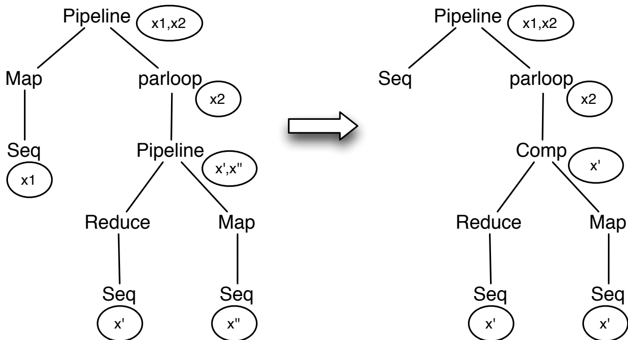


Fig. 4: The annotated skeleton tree of the application: its implementation and performance are parametric w.r.t. x_1, x_2, x', x''

$$map(Seq_D) \circ parloop((reduce(Seq_M) \circ map(Seq_R)), E)$$
$\rightarrow$\{let x_1, x_2 two cores, rule 9 holds and $Seq_D \rightarrow Seq_{Dx_1}$ hold \}

$$map(Seq_{Dx_1}) \circ parloop((reduce(Seq_M) \circ map(Seq_R)), E)_{x_2}$$

$\rightarrow$\{ rule 5 and x', x'' potentially fresh id\}

$$map(Seq_{Dx_1}) \circ parloop((reduce(Seq_M)_{x'} \circ map(Seq_{Rx''})), E_{x_2})_{x_2}$$

The annotated tree associated with such mapping evaluation is depicted in Fig. 4 (left) where x_1, x_2, x', x'' could identify a set of different or overlapping cores. In addition, each of x_1, x_2, x', x'' can also be either CPU or GPU cores. The performance model provides the information needed to choose the mapping which gives the best performance.

It is worth pointing out that at this level the semantic framework could also be able to define (under specific performance requirements) an alternative skeleton tree, functional equivalent to the preceding one for which a new set of mapping alternatives could hold. Fig. 4 suggests a possible rewriting of the skeleton in which $map(Seq_{D_1})$ has been rewritten as sequential and the pipeline iterated by *parloop* has collapsed in a *comp*. Such skeleton could be eligible if, for instance, the costs involved in the implementation of the map and the pipeline are too high with respect to a sequential execution, possibly because of a too fine grain computation.

Mapping. According to the methodology introduced in Sec. 3, the parallelized versions of *Detect* and *Restore* can be mapped onto different processors (i.e. $x_1 \neq x''$), resulting in different performance figures. For example, Fig. 2 presents, together with the sequential execution time, the performance when the *Detect* and *Restore* filters are executed in parallel using, respectively, 8 and 24 cores

of the 32 cores of the Intel workstation described at the beginning of this section. Alternatively, the restoration loop can be offloaded to a GPGPU (NVidia Tesla 448 cores) with even better results. In this mapping process, the described methodology is intended to ensure that only appropriate compositions of patterns are mapped onto the GPGPU.

5 Related Work

Early proposals of pattern-based parallel programming frameworks have been focused mainly on distributed memory platforms, such as clusters of workstations and grids [12, 30]. Google MapReduce [13] brings to the mainstream of out-of-core data processing the map-reduce paradigm. All these skeleton frameworks provide several parallel patterns (algorithmic skeletons) covering mostly task and data parallelism. These patterns can usually nested to model more complex parallelism exploitation patterns according to the constraints imposed by the specific programming framework. More recent pattern-based frameworks, following the platform architecture trend, have shifted the focus to multi-cores and the shared address model; in addition to FastFlow, it is worth mentioning the Intel Threading Building Block (TBB) library [8], and to a limited extent the Microsoft Task Parallel Library [17]. All of them are certainly higher-level compared to the Pthread library that has been used in the shared memory implementations of classification algorithms previously mentioned. The main features of these and other frameworks are surveyed in [11].

Programming frameworks based on algorithmic skeletons have been recently introduced to alleviate the task of the application programmer when targeting data parallel computations on GPUs. In Muesli [31] the programmer must explicitly indicate whether GPUs are to be used for data parallel skeletons. SkePU [18] provides programmers with GPU implementations of map and reduce skeletons and relies on StarPU for the execution of stream parallel skeletons (pipe and farm).

In addition to pattern-based frameworks, other high-level programming frameworks also aim to simplify the design of efficient applications on multi-cores and thus are related to FastFlow and to the present work. StreamIt [14] is an explicitly parallel programming language based on the Synchronous Data Flow model that enables the assembly of program modules (called filters) in a *pipeline* fashion, possibly with a *FeedbackLoop*, or according to a *SplitJoin* data-parallel schema. Streaming applications are also targeted by TBB through the *pipeline* construct, which also provides programmers with thread-safe containers and some parallel patterns (called "algorithms"); TBB does not support any kind of non-linear streaming network, which thus has to be embedded in a pipeline with significant programming and performance drawbacks. Intel's Concurrent Collections (CnC), which declaratively models concurrent activities as data streams and control dependencies, has been recently proposed as a candidate substrate for parallel patterns [32]. *OpenMP* [33] is a popular thread-based framework for multi-core architectures mostly targeting data parallel program-

ming even if it is currently being extended to incorporate stream processing. OpenMP supports, by way of language pragmas, the low-effort parallelization of sequential programs; however, these pragmas are mainly designed to exploit loop-level data parallelism (e.g. *do_independent*). CnC and OpenMP do not natively support either *farm* or *Divide&Conquer* patterns, even though they can be simulated with lower-level features.

MCUDA [34] is a framework to mix CPU and GPU programming. In MCUDA it is mandatory to define kernels for all available devices but the framework can not make any assumptions about the relative performance of the supported devices.

Recently OpenACC [35] has been proposed by some major vendors as a possible new standard for programming GPUs and HW accelerators in general. Like OpenMP it is based on a set of pragma directives allowing automatic acceleration of loops and parallel regions by directly offloading computation on the accelerator. Our approach differs from #pragma-based approaches because we require an explicit parallelization of the code thus making it possible to avoid cluttering the sequential code with complex directives. Furthermore, in our vision, the declarative approach gives, in many cases, less control to the programmer and hence lessens the possibility to exploit all the available parallelism in the application.

Overall, our approach aims to provide a high-level programming model based on algorithmic skeletons and a high-level skeleton-based intermediate representation with mapping annotations, which are used for taking mapping decisions. In our vision such an approach is able to ensure portability of the parallel code onto different heterogeneous platforms while maintaining good performance.

6 Conclusions

The mapping problem, and in general the optimization of parallel code for current and next generation parallel platforms is a particularly important problem as it might significantly affect performance and efficiency of applications. Ideally, solutions to this problem should address performance portability while not requiring excessive effort on the part of the application developer. In this respect, the pattern-based approach has been demonstrated to have the potential to address the problem. In this position paper we have stated the problem and the approach we are undertaking to define an intermediate formalism to support the compilation and the optimization of patterns on heterogeneous multi-core and many-core platforms. The intermediate language basically aims to equip patterns with several platform attributes in such a way that suitable mapping and scheduling heuristics aimed at generating optimized run-time code can be derived.

Acknowledgment The work described in this paper is supported by the EU ParaPhrase project (`http://www.paraphrase-ict.eu`, 2011-2014).

References

1. Parnas, D.L.: On the design and development of program families. IEEE Trans. on Software Engineering **SE-2**(1) (March 1976) 1–9
2. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computations. Research Monographs in Par. and Distrib. Computing. Pitman (1989)
3. Botorog, G.H., Kuchen, H.: Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In: Proc. of the 5th International Symposium on High Performance Distributed Computing (HPDC'96), IEEE Computer Society Press (1996) 243–252
4. Darlington, J., Guo, Y., Jing, Y., To, H.W.: Skeletons for structured parallel composition. In: Proc. of the 15th Symposium on Principles and Practice of Parallel Programming. (1995)
5. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P^3L: A Structured High level programming language and its structured support. Concurrency Practice and Experience **7**(3) (May 1995) 225–255
6. Hamdan, M., King, P., Michaelson, G.: A scheme for nesting algorithmic skeletons. In Hammond, K., Davie, T., Clack, C., eds.: Proc. of the 10th International Workshop on the Implementation of Functional Languages (IFL'98), Department of Computer Science, University College London (1998) 195–211
7. Aldinucci, M., Danelutto, M.: Skeleton based parallel programming: functional and parallel semantics in a single shot. Computer Languages, Systems and Structures **33**(3-4) (October 2007) 179–192
8. Intel Corp.: Threading Building Blocks. (2011)
9. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In Pllana, S., Xhafa, F., eds.: Programming Multi-core and Many-core Computing Systems. Parallel and Distributed Computing. Wiley (2012)
10. Cole, M.: Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. Parallel Computing **30**(3) (2004) 389–406
11. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. Software: Practice and Experience **40**(12) (2010) 1135–1160
12. Vanneschi, M.: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing **28**(12) (December 2002) 1709–1732
13. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Usenix OSDI '04. (December 2004) 137–150
14. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Proc. of the 11th Intl. Conference on Compiler Construction (CC), London, UK, Springer (2002) 179–196
15. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A view of the parallel computing landscape. Comm. of the ACM **52**(10) (2009) 56–67
16. Apache Software Foundation: Hadoop. (2008) `http://hadoop.apache.org/`.
17. Leijen, D., Hall, J.: Optimize managed code for multi-core machines. MSDN Magazine (October 2007)
18. Enmyren, J., Kessler, C.W.: Skepu: a multi-backend skeleton programming library for multi-gpu systems. In: Proceedings of the fourth international workshop on High-level parallel programming and applications. HLPP '10, New York, NY, USA, ACM (2010) 5–14

19. Aldinucci, M., Coppola, M., Danelutto, M.: Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In Gorlatch, S., ed.: Proc of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming, Fakultät für mathematik und informatik, Uni. Passau, Germany (May 1998) 44–58
20. Skillicorn, D.B., Cai, W.: A cost calculus for parallel functional programming. J. Parallel Distrib. Comput. **28**(1) (1995) 65–83
21. Aldinucci, M., Gorlatch, S., Lengauer, C., Pelagatti, S.: Towards parallel programming by transformation: The FAN skeleton framework. Parallel Algorithms and Applications **16**(2–3) (March 2001) 87–121
22. Caromel, D., Henrio, L., Leyton, M.: Type safe algorithmic skeletons. In: 16th Euromicro Intl. Conference on Parallel, Distributed and Network-Based Processing (PDP), Toulouse, France, IEEE (February 2008) 45–53
23. Gorlatch, S., Lengauer, C., Wedler, C.: Optimization rules for programming with collective operations. In: Proc. of the 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99). IEEE Computer Society Press (1999) 492–499
24. Skillicorn, D.B., Cai, W.: A cost calculus for parallel functional programming. Journal of Parallel and Distributed Computing **28** (1995) 65–83
25. Aldinucci, M., Danelutto, M.: Stream parallel skeleton optimization. In: Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems, Cambridge, Massachusetts, USA, IASTED, ACTA press (November 1999) 955–962
26. Pottenger, B., Eigenmann, R.: Idiom recognition in the Polaris parallelizing compiler. In: Proc. of the 9th Intl. Conference on Supercomputing (ICS '95), New York, NY, USA, ACM Press (1995) 444–448
27. Aldinucci, M., Torquati, M.: FastFlow website. (2009) `http://mc-fastflow.sourceforge.net/`.
28. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: An efficient unbounded lock-free queue for multi-core systems. In: Proc. of 18th Intl. Euro-Par 2012 Parallel Processing. LNCS, Rhodes Island, Greece, Springer (2012)
29. Aldinucci, M., Drocco, M., Giordano, D., Spampinato, C., Torquati, M.: A parallel edge preserving algorithm for salt and pepper image denoising. Technical Report 138/2011, Università degli Studi di Torino, Dip. di Informatica, Italy (May 2011)
30. Kuchen, H.: A skeleton library. In Monien, B., Feldman, R., eds.: Proc. of 8th Euro-Par 2002 Parallel Processing. Volume 2400 of LNCS., Paderborn, Germany, Springer (August 2002) 620–629
31. Ernsting, S., Kuchen, H.: Data parallel skeletons for gpu clusters and multi-gpu systems. In: Proceedings of PARCO 2011, IOS Press (2011)
32. Newton, R., Schlimbach, F., Hampton, M., Knobe, K.: Capturing and composing parallel patterns with Intel CnC. In: Proc. of USENIX Workshop on Hot Topics in Parallelism (HotPar 2010), Berkley, CA, USA (June 2010)
33. Park, I., Voss, M.J., Kim, S.W., Eigenmann, R.: Parallel programming environment for OpenMP. Scientific Programming **9** (2001) 143–161
34. Stratton, J.A., Stone, S.S., mei W. Hwu, W.: MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In Amaral, J.N., ed.: Languages and Compilers for Parallel Computing, 21th Intl. Workshop (LCPC). Volume 5335 of LNCS., Springer (2008) 16–30
35. Khronos Compute Working Group: OpenACC Directives for Accelerators. (November 2012) `http://www.openacc-standard.org`.