International Journal of Parallel Programming - ISSN 0885-7458
DOI: 10.1007/s10766-013-0273-6
The final publication is available at link.springer.com

Design patterns percolating to parallel programming framework implementation

M. Aldinucci[c], S. Campa[a], M. Danelutto[a], P. Kilpatrick[b], M. Torquati[a]

September 2013

Abstract Structured parallel programming is recognised as a viable and effective means of tackling parallel programming problems. Recently, a set of simple and powerful parallel building blocks (RISC-pb^{2}l) has been proposed to support modelling and implementation of parallel frameworks. In this work we demonstrate how that same parallel building block set may be used to implement both general purpose parallel programming abstractions, not usually listed in classical skeleton sets, and more specialized domain specific parallel patterns. We discuss the associated implementation techniques and present experimental evidence of the feasibility and efficiency of the approach.

Keywords Algorithmic skeleton, parallel design patterns, programming frameworks, RISC-pb^{2}l, parallel building blocks.

1 Introduction

Various authors and research communities have recognised that structured parallel programming represents a viable means to improve the efficiency of the entire process of designing, implementing and tuning parallel applications.

Beginning in the late '90s, the algorithmic skeleton [10] community developed various systems providing the application programmer with suitable abstractions modelling the more common, useful and efficient parallelism exploitation patterns [9,11,21,23,19]. Recently, efficient skeleton based parallel programming frameworks have been developed targeting multi-core hardware, possibly equipped with GPUs, as well as distributed clusters [14,15,5].

Meanwhile, parallel design patterns have been proposed [22,24] and recognised to have the potential to induce a radical change in the parallel program-

Work partially supported by FP7 STREP ParaPhrase (`www.paraphrase-ict.eu`)

Dept. of Computer Science of [a]Univ. of Pisa, [b]Queen's Univ. Belfast, [c]Univ. of Torino
E-mail: {campa,marcod,torquati}@di.unipi.it, p.kilpatrick@qub.ac.uk, aldinuc@di.unito.it

ming scenario, suitable for educating a new generation of parallel programmers capable of fully managing the new highly parallel architectures provided by hardware vendors [6].

Algorithmic skeleton and parallel design pattern concepts differ basically in the way structured parallel programming abstractions are provided to the application programmer. Algorithmic skeletons are provided as parametric language constructs or library functions/objects ready to be instantiated/used and to be customised through parameters. Parallel design patterns are provided as software engineering "recipes" to be followed when a given parallel pattern has to be used, and require a larger programming activity than algorithmic skeletons to be actually exploited. This notwithstanding, both skeletons and design patterns simplify the application programmer's task and make the whole application development process more efficient by:

- providing known and efficient parallelism exploitation patterns as composable building blocks, such that complex applications may be implemented by composing certified abstractions rather than designing ad hoc solutions;
- relieving the application programmer from the tasks associated with parallel pattern implementation and tuning; and
- supporting functional *and* performance portability across different target architectures.

Overall, these features raise the level of abstraction of the mechanisms exposed to the application programmer and distinguish "structured parallel programming" frameworks and models from more traditional, low level approaches such as MPI, OpenCL and OpenMP, where the code necessary to implement parallel features within the application remains completely the responsibility of the application programmer, and solutions identified when implementing a parallel application are not readily reusable when working to parallelise another application or when moving the application to a different target architecture.

Recently, we proposed adopting the approach underpinning both algorithmic skeletons and parallel design patterns at a lower level in the software stack used to develop efficient, maintainable and portable parallel applications. The idea is to take advantage of the skeleton/pattern approach not only at application level (that is, in order to ease the software developer's job), but also in the design and implementation of the underlying implementation frameworks. In [12] we proposed adopting a set of skeletons/patterns to be used as implementation building blocks of the high level abstractions eventually exposed to the application programmer, that is algorithmic skeletons or parallel design patterns. In a sense, we abandoned the idea of a highly optimised but monolithic implementation of each skeleton/pattern, optimised for a given target architecture, in favour of an implementation built as an orchestration of a small (RISC) set of simpler, efficient and re-usable building blocks (RISC-pb^{2}l). This further step in "structuring" the parallel programming framework implementation presents several advantages, closely mirroring those usually observed when providing programming abstractions to the application programmer: a) implementations of the set of building blocks may be suitably specialised to

target different architectures; b) the well-defined semantics of the building blocks may present the possibility to automatically refactor building block compositions to better exploit target architecture features; and c) a wider range of high level abstractions may eventually be provided to the application programmer by composing the building blocks in different ways.

The RISC-pb^{2}l features were demonstrated in [12] by showing how several simple, common parallel patterns may be implemented efficiently using the set of building blocks proposed. Here we propose the use of the RISC-pb^{2}l set in two areas that were not covered in that earlier work, namely:

- *General Computing Models.* We show how three different general computing models–rather than classical algorithmic skeletons–may be implemented using RISC-pb^{2}l: BSP, Macro-dataflow and Google map-reduce.
- *Domain Specific Patterns.* We describe implementations, using RISC-pb^{2}l, of domain specific skeletons/patterns which are of particular importance in their respective domains. A genetic algorithm, a network packet processing algorithm and a symbolic computation algorithm are presented.

The original contribution of this paper therefore consists in a) demonstrating the usability of RISC-pb^{2}l to support both abstract programming models and domain specific patterns in quite different application domains (previous work demonstrated only the usability of RISC-pb^{2}l when implementing rather simple, classical patterns); b) demonstrating the efficiency achieved in the implementation of different use case applications using the skeleton abstractions (either abstract models or domain specific patterns) implemented with a prototype implementation of RISC-pb^{2}l using FastFlow [16]; and c) assessing the potential for optimizing RISC-pb^{2}l compositions (skeleton implementations) by simply considering the semantics of the building blocks.

The remainder of the paper is structured as follows: Sec. 2 briefly outlines the features of RISC-pb^{2}l. Sec. 3 and Sec. 4 discuss the main contribution of this paper (with Sec. 6), namely how general programming model abstractions and domain specific patterns may be modelled and implemented with RISC-pb^{2}l. Sec. 5 briefly introduces FastFlow and outlines how FastFlow base classes *de facto* implement RISC-pb^{2}l building blocks. Sec. 6 discusses results relative to the implementation of general programming model abstractions and domain specific skeletons with RISC-pb^{2}l on top of FastFlow. Finally, Sec. 7 presents related work and Sec. 8 draws conclusions.

2 The building block set

The building block set RISC-pb^{2}l (RISC Parallel Building Block Library) proposed in [12] is composed of three kinds of building blocks: *wrappers*, *functionals* and *combinators* (see table in Fig. 1). *Wrappers* are used to embed existing portions of code into parallel programs. Wrapped programs are intended to behave as functions and so no side effects should be present within the wrapped code. *Functionals* model those blocks encapsulating parallel computations, including computing the same or a set of different functions on n input items or

computing in stages over an input item. *Combinators* just route data to/from functionals (input and output) in such a way that these data may be efficiently processed, and include a specific combinator ($\triangleright_{Gatherall}$, see table in Fig. 1) explicitly implementing a barrier synchronization. Each of the components in RISC-pb^{2}l is characterised by an input and an output *arity*. RISC-pb^{2}l components may be arbitrarily nested provided the component combination (we use $\circ$ to express combination as juxtaposition of components) respect input/output arity matching. The last part of the table in Fig. 1 formally describes legal combinations of RISC-pb^{2}l components.

For example, the RISC-pb^{2}l component composition

$$\triangleleft_{Scatter} \bullet [|\ ((codef))\ |]_{nw}$$

takes one input (input arity of $\triangleleft_{Scatter}$), produces nw outputs (nw is the input and output arity of $[|\ \ldots\ |]_{nw}$) and computes the function implemented by the sequential code $((codef))$ over the nw partitions routed by the $\triangleleft_{Scatter}$.

The semantics of RISC-pb^{2}l components is that of data flow: as soon as a complete input data set is presented to the building block, the block "fires" and computes output results using the current input values (or routes input data to appropriate outputs). The design of RISC-pb^{2}l has been greatly inspired by Backus' FP [7]. In particular, we borrow the higher order functions/combinators distinction proposed in that work for sequential building blocks and we aim eventually to provide a similar "algebra of programs" suitable for supporting parallel program refactoring and optimisation. In this work we concentrate the discussion on the suitability of RISC-pb^{2}l to support the implementation of general programming model and domain specific parallel skeletons/design patterns, possibly optimized through RISC-pb^{2}l expression rewriting. The interested reader my refer to [12] for more details of RISC-pb^{2}l.

3 General purpose computing models

To validate the expressive power of the RISC-pb^{2}l set, we consider in this section some examples of parallel patterns. In [12] we have shown how basic stream and data parallel skeletons/patterns can be defined (specifically, *pipe*, *farm*, *map* and *reduce*) by composing the building blocks provided by the set. Here, we wish to highlight the generality of such an approach and the proposed grammar, demonstrating that the RISC-pb^{2}l set can be used not only to define basic, embarrassingly parallel patterns but also general, theoretically well-established and well-known computation patterns as well as domain-specific skeletons/patterns. In this work we consider: the BSP parallel computation model, the MapReduce pattern, the Macro-dataflow model as computation patterns; and also three domain-specific skeletons/patterns belonging to the fields of evolutionary computation, symbolic computation and networking.

Name	*Syntax*	*Informal semantics*
	Wrappers	
Seq wrapper	$((f))$	Wraps sequential code into a RISC-pb^{2}l "function" .
Par wrapper	$(\mid f \mid)$	Wraps any parallel code into a RISC-pb^{2}l "function" (e.g. code offloading to GP-GPU or a parallel OpenMP code).
	Functionals	
Parallel	$[\mid \Delta \mid]_n$	Computes in parallel n identical programs on n input items producing n output items.
MISD	$[\mid \Delta_1, \ldots, \Delta_n \mid]$	Computes in parallel a set of n different programs on n input items producing n output items.
Pipe	$\Delta_1 \bullet \ldots \bullet \Delta_n$	Uses n (possibly) different programs as stages to process the input data items and to obtain output data items. Program i receives inputs from program $i-1$ and delivers results to $i+1$.
Reduce	$({}_g\triangleright)$	Computes a single output item using an l level ($l \geq 1$) $k-ary$ tree. Each node in the tree computes a (possibly commutative and associative) $k-ary$ function g.
Spread	$({}_f\triangleleft)$	Computes n output items using an l level ($l \geq 1$) $k-ary$ tree. Each node in the tree uses function f to compute k items out of the input data items.
	Combinators	
1-to-N	$\triangleleft_{D\text{-}Pol}$	Sends data received on the input channel to one or more of the n output channels, according to policy $D\text{-}Pol$ with $D\text{-}Pol \in [Unicast(p), Broadcast, Scatter]$ where $p \in [RR, AUTO]$. RR applies a round robin policy, $AUTO$ directs the input to the output where a request token has been received
N-to-1	$\triangleright_{G\text{-}Pol}$	Collects data from the n input channels and delivers the collected items on the single output channel. Collection is performed according to policy $G\text{-}Pol$ with $G\text{-}Pol \in [Gather, Gatherall, Reduce]$. $Gatherall$ waits for an input from all the input channels and delivers a vector of items, *de facto* implementing a barrier.
Feedback	$\overleftarrow{(\Delta)}_{cond}$	Routes output data y relative to the input data x ($y = \Delta(x)$) back to the input channel or drives them to the output channel according to the result of the evaluation of $Cond(x)$. May be used to route back n outputs to n input channels as well.

Legal compositions grammar

$$\Delta^n ::= [\mid \Delta \mid]_n \mid [\mid \Delta_1, \ldots, \Delta_n \mid] \mid \overleftarrow{(\Delta^n)}_{cond} \mid \Delta^n \bullet \Delta^n$$
$$\Delta^{1n} ::= \triangleleft_{Pol} \mid ({}_f\triangleleft)$$
$$\Delta^{n1} ::= \triangleright_{Pol} \mid ({}_g\triangleright)$$
$$\Delta ::= ((code)) \mid (\mid code \mid) \mid \Delta \bullet \Delta \mid \overleftarrow{(\Delta)}_{cond} \mid \Delta^{1n} \bullet \Delta^{n1} \mid \Delta^{1n} \bullet \Delta^n \bullet \Delta^{n1}$$

Fig. 1: Base building blocks of the parallel instruction set.

3.1 The BSP model

In the Bulk Synchronous Parallel model [32], a parallel computation running on a set of processors proceeds in a sequence of super-steps, each step organised as follows: *i*) each processor computes locally by accessing only its local variables and environment; *ii*) eventually processors begin an asynchronous communication step in which they exchange data; *iii*) each processor enters a *barrier* waiting for the completion of the communication step. A BSP computation can be described as a sequence of super-steps Δ^i_{SS}, where $i \in [1, k]$ and k is the number of super-steps. Each super-step can be seen as a two-stage pipeline. Letting n be the number of parallel activities or processes involved in

a super-step computation, we use $\Delta^i_{step_j}$ to denote the j^{th} process, $j \in [1, n]$, at the i^{th} super-step. The first stage of the pipeline may be modelled as a *MISD* functional $[|\ \Delta^i_{step_1}, \ldots, \Delta^i_{step_n}\ |]$ where each $\Delta^i_{step_j}$ computes a list of pairs $\langle$`value, index`$\rangle$, where the index denotes to which of the possible n destinations the `value` message is directed. The second stage implements the communication and barrier features of each super-step. It may be implemented using a *spread* functional $(_f\triangleleft)$ to route all the messages towards the target $\Delta^{i+i}_{step_j}$, as follows

$$\triangleright_{Gatherall} \bullet (_{route2Dest}\triangleleft)$$

where *route2Dest* is the function routing input messages to their final destination according to the superstep communication code prescriptions. The single superstep may then be modelled using RISC-pb^{2}l as follows:

$$\Delta^i_{SS} = \underbrace{([|\ \Delta^i_{step_1}, \ldots, \Delta^i_{step_n}\ |]}_{Compute + Prepare\ Msg} \bullet \underbrace{\triangleright_{Gatherall} \bullet (_{route2Dest}\triangleleft))}_{Communication + Barrier}$$

and the whole BSP computation may be expressed as

$$BSP(k) = \Delta^1_{SS} \bullet \ldots \bullet \Delta^k_{SS}$$

When considering the composition of the Δ^i_{SS} phases (see Fig. 2 left) we can clearly recognize the presence of a bottleneck and, at the same time, a discrepancy w.r.t. the way communications are implemented within BSP. By collecting all messages in a single place through the $\triangleright_{Gatherall}$ and then applying the $(_{RouteToDest}\triangleleft)$, a) the constraint imposed by the h-relation of not having more than h communications (incoming or outgoing) incident upon the same node is violated, and b) the node gathering the messages from the supersteps constitutes a bottleneck.

The factorization of components in the RISC-pb^{2}l set, however, provides suitable tools to cope with this kind of situation. An expression such as

$$\triangleright_{Gatherall} \bullet (_f\triangleleft)$$

actually routes to the $(_f\triangleleft)$ messages from the n sources of the $\triangleright_{Gatherall}$ leaving those messages unchanged. Under the hypothesis that the $(_f\triangleleft)$ only routes those messages to their correct destinations among the m outputs of the $(_f\triangleleft)$ tree, the $\triangleright_{Gatherall} \bullet (_f\triangleleft)$ expression may clearly be transformed into an equivalent form where a distinct $(_f\triangleleft)$ is used to route messages at each of the inputs of the original $\triangleright_{Gatherall}$ tree. All the n $(_f\triangleleft)$ trees will share the m outputs of the original, single $(_f\triangleleft)$. The original and optimized communication patterns for the two cases are shown in Fig. 2 right. In terms of RISC-pb^{2}l components, this optimization may be expressed as

$$\triangleright_{Gatherall} \bullet (_f\triangleleft) \equiv [|\ (_f\triangleleft)\ |]_n \qquad \texttt{(Opt1)}$$

where all the i^{th} outputs of the n $(_f\triangleleft)$ are assembled in the single i^{th} output of the $[|\ (_f\triangleleft)\ |]_n$.

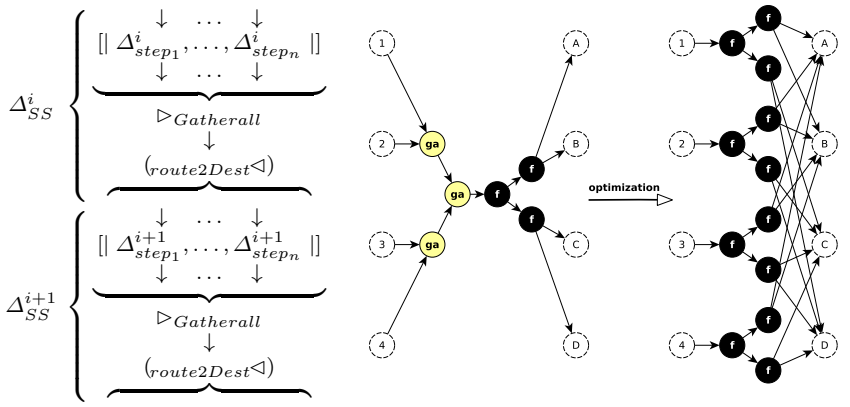

Fig. 2: RISC-pb^{2}l BSP superstep sequence (left). Optimizing $\triangleright_{Gatherall} \bullet (_f\triangleleft)$ by $[|\ (_f\triangleleft)_1, \ldots, (_f\triangleleft)_n\ |]$ (f being *route2Dest*, *ga* being *Gatherall*) (right).

It is worth noting that the optimization just outlined a) removes the original bottleneck, b) ensures the BSP $h-$relation and, last but not least, c) may be introduced in a completely automatic way *any time* we recognize that the (stateless) function f only processes a single message at a time in a $(\triangleright_{Gatherall} \bullet (_f\triangleleft))$ expression. This is exactly in the spirit of RISC-pb^{2}l design: the system programmer responsible for providing an implementation for a given parallel pattern may use the RISC-pb^{2}l building blocks and rely on the optimizations, separately designed by the RISC-pb^{2}l engineers, to achieve efficiency.

The optimized version of the communications in the BSP skeleton as depicted in Fig. 2 (right) actually removed the synchronization of the $\triangleright_{Gatherall}$ of the original expression. Therefore we should add an explicit synchronization immediately before the conclusion of the generic BSP superstep Δ^i_{SS}:

$$[|\ \Delta^i_{step_1}, \ldots, \Delta^i_{step_n}\ |] \bullet [|\ (_{route2Dest}\triangleleft)_1, \ldots, (_{route2Dest}\triangleleft)_n\ |] \bullet \triangleright_{Gatherall} \bullet \triangleleft_{Scatter}$$

(the final $\triangleleft_{Scatter}$ is needed to re-establish the correct arities after the $\triangleright_{Gatherall}$ has collapsed its n inputs into a single output). The $\triangleright_{Gatherall} \bullet \triangleleft_{Scatter}$ implements the barrier and, in order not to introduce inefficiencies (bottlenecks) and violations of the BSP model features (h-relation), it has to be substituted by an efficient implementation of a barrier, if available, with no data routing at all. As a consequence, the BSP super-step may simply be rewritten as

$$[|\ \Delta^i_{step_1}, \ldots, \Delta^i_{step_n}\ |] \bullet [|\ (_{route2Dest}\triangleleft)_1, \ldots, (_{route2Dest}\triangleleft)_n\ |] \bullet ((barrier))$$

The rewriting

$$\triangleright_{Gatherall} \bullet \triangleleft_{Scatter} \equiv ((barrier)) \qquad \texttt{(Opt2)}$$

may be considered a worthwhile optimization any time an efficient barrier operation is available in the target architecture at hand.

3.2 The MapReduce

The MapReduce pattern (MR) introduced by Google models those applications where a collection of input data is processed in two steps [28,13]: *i*) a *map step* computes a $\langle key, value\rangle$ pair for each item in the collection by applying a function f, and *ii*) a *reduce step* "sums up" (in parallel for all *key*s) all the *value* items with a given *key* using an associative and commutative operator $\oplus$. The map reduce pattern may be described as follows. Given an input collection $(x_1, \ldots, x_n)$, $x_i \in X$, a function $f : X \to \langle Key, Y\rangle$, a binary and associative function $\oplus : Y \to Y \to Y$ and assuming $keys\_of : \langle Key, Y\rangle\ list \to Key\ list$ returns the list of keys appearing in the first list of pairs and $K : Key \to \langle Key, Y\rangle\ list \to Y\ list$ returns the list of values of the items in the $\langle Key, Y\rangle$ list with a given *key*, then

$$\mathrm{MR}(f, \oplus)(x_1, \ldots, x_n) = \{\Sigma_\oplus(K(k)) \mid k \in keys\_of(f(x_1), \ldots, f(x_n))\}$$

The computation of $\mathrm{MR}(f, \oplus)(x_1, \ldots, x_n)$ is obviously performed in parallel, as the input collection is usually already partitioned across a set of processing elements and the set of reduces are cooperatively executed by these PEs immediately after having computed the map phase in parallel[1].

The mapreduce pattern can therefore be described as the composition of two functionals, modelling the map and reduce phases, respectively:

$$MR(f, \oplus) = \Delta_{map(f)} \bullet \Delta_{red(\oplus)}$$

with:

$$\begin{aligned} \Delta_{map(f)} &= \triangleleft_{Scatter} \bullet [|\ ((f))\ |]_{nw} \\ \Delta_{red\oplus} &= \triangleright_{Gatherall} \bullet ({}_K\triangleleft) \bullet [|\ \triangleleft_{Scatter} \bullet ({}_\oplus\triangleright)\ |]_{nw'} \end{aligned}$$

The *spread* over K $({}_K\triangleleft)$ will route each $\langle K, Y\rangle$ item to the parallel activity dealing with all items with $Key = K$, that is to the parallel activity (worker) computing the $({}_\oplus\triangleright)$ of the keys K. Logically $nw' = \#\{keys\_of(f(x_1), \ldots, f(x_n))\}$ although it is obvious that some smaller nw' will be used to increase the efficiency of the reduce computation, such that each $({}_\oplus\triangleright)$ works on partitions of keys rather than on single key values. In the same way, nw should be equal to n, the cardinality of the input data set, but a smaller value will be used such that each parallel activity in the $[|\ ((f))\ |]_{nw}$ actually computes $((f))$ over a partition of the input data items.

It is worth pointing out that the $\triangleright_{Gatherall} \bullet ({}_K\triangleleft)$ and $\triangleleft_{Scatter} \bullet ({}_\oplus\triangleright)$ portions of the $\Delta_{red(\oplus)}$ part of the computation are naturally suitable for several "routing" optimizations in a concrete implementation.
In fact, as in the *BSP* example and since we are working with stateless building blocks, the $\triangleright_{Gatherall} \bullet ({}_K\triangleleft)$ expression can be automatically recognized and optimized to avoid the bottleneck induced by the barrier represented by the *Gatherall* operation. Assuming that $\Delta_{map(f)}$ produces as output a set of

[1] this is slightly different from the composition of *map* and *reduce* skeletons as perceived by the algorithmic skeleton community [8]

items $\{(k_1, x_1), \ldots, (k_m, x_n)\}$ for $k_i \in Key, i \in [1, m]$, an optimized version of $\triangleright_{Gatherall} \bullet ({}_K\triangleleft)$ may be defined as

$$[|({}_{K^1}\triangleleft), \ldots, ({}_{K^n}\triangleleft)|]$$

where $K^j = K(k, x_j), j \in [1, n]$ for any k. In other words, the gathering followed by the spread of the K function applied over the whole collection is rewritten as a MISD operation on a list of pairs in which each input item (x_i, k_j) of the list is routed to a filter $K(k, (x_i, k_s))$ of keys. As a result, the expression produces a set of lists which become the input for the reduce phase. Summing up, a first rewriting of the $MR(f, \oplus)$ pattern is defined as follows:

$$\triangleleft_{Scatter} \bullet [|\ ((f))\ |]_{nw} \bullet [|({}_{K^1}\triangleleft), \ldots, ({}_{K^n}\triangleleft)|]_{nw''} \bullet [|\triangleleft_{Scatter} \bullet ({}_{\oplus}\triangleright)|]_{nw'}$$

where nw'' is the number of parallel activities evaluating $((f))$. However, by imposing $nw = nw''$, and taking into account that obviously

$$[|\ \Delta_1\ |]_n \bullet [|\ \Delta_2\ |]_n \equiv [|\ \Delta_1 \bullet \Delta_2\ |]_n \qquad \texttt{(Opt3)}$$

a further optimization could be expressed as

$$\triangleleft_{Scatter} \bullet [|((f)) \bullet ({}_{K^1}\triangleleft), \ldots, ((f)) \bullet ({}_{K^n}\triangleleft)|]_{nw} \bullet [|\triangleleft_{Scatter} \bullet ({}_{\oplus}\triangleright)|]_{nw'}$$

where the creation of the pairs and the filtering stages are composed in a pipeline within the same worker of a MISD building block. In an actual implementation this can be translated into a sequential execution of f and k on the same physical machine, thus exploiting locality and reducing communication and data transfer overheads.

3.3 The Macro-dataflow model

In the macro-dataflow (MDF) model a program is interpreted by focusing on the functional dependencies among data. The evaluation of a program is realized as the traversal of a graph whose nodes are macro-dataflow instructions that become "fireable" (i.e. executable) as soon as the input data (token) is available on the input arcs. Different nodes can become fireable in parallel depending on the dynamic availability of tokens through the arcs as computation proceeds. This model can be described as a three-stage pipeline: the first stage, through the function b, builds the graph (possibly dynamically); the second stage determines, via the function e, which instructions have become fireable; and the third stage executes the MDF instructions via a parallel functional whose programs use a function, f, capable of executing any instruction in the graph. RISC-pb^{2}l can be used to describe the pattern via the expression:

$$MDF(b, e, f) = \overleftarrow{((b)) \bullet ((e)) \bullet (\triangleleft_{Unicast(AUTO)} \bullet [|\ ((f))\ |]_{nw} \bullet \triangleright_{Gather})}_{c}$$

where nw denotes the available parallelism degree. The tokens generated by the parallel functional determine which instructions will become fireable at

the next iteration and the process proceeds until no more instructions become fireable (i.e. the feedback condition c is always $true$ up to when no more tokens are produced). In many cases, the dependency graph is built statically once and for all, and so the MDF pattern can be reduced to:

$$MDF(b,e,f) = ((b)) \bullet \overleftarrow{(((e)) \bullet \lhd_{Unicast(AUTO)} \bullet [|\ ((f))\ |]_{nw} \bullet \rhd_{Gather}))}_{c}$$

Independent of the "range" of the feedback combinator, it may be noted that

$$\forall \Delta \in \Delta^n : \overleftarrow{(\Delta \bullet \rhd_P)}_c \equiv \overleftarrow{(\rhd_P \bullet \Delta)}_c \qquad \texttt{(Opt4)}$$

and therefore we may write the MDF as:

$$MDF(b,e,f) = \overleftarrow{(\rhd_{Gather} \bullet ((b)) \bullet ((e)) \bullet \lhd_{Unicast(AUTO)} \bullet [|\ ((f))\ |]_{nw})}_{c}$$

and then, considering that the first $\rhd_{Gather}$ *de facto* implements an alternative command and that the $\lhd_{Unicast(AUTO)}$ connector only routes output of $((e))$ to one of the inputs of $[|\ ((f))\ |]_{nw}$, we can apply the following transformations:

$$\begin{array}{ll} \rhd_{Gather} \bullet ((b)) & \text{into } ((b_{merge})) \\ ((e)) \bullet \lhd_{Unicast(AUTO)} & \text{into } (_e\lhd) \end{array}$$

thus obtaining an optimized expression for the MDF pattern as follows:

$$MDF(b,e,f) = \overleftarrow{(((b_{merge})) \bullet (_e\lhd) \bullet [|\ ((f))\ |]_{nw})}_{c} \qquad (1)$$

4 Domain specific skeletons

4.1 Global single population genetic skeleton

This skeleton belongs to the family of so-called genetic algorithmic skeletons [25,1], i.e. those patterns of computation inspired by the evolutionary computing field, in which optimization techniques emulate the natural selection process. The convergence is defined in terms of evolution of a population (the solution space) in which only those individuals exhibiting the better fitness are selected to mutate, reproduce (or eventually to die). The skeleton can be represented as follows:

$$GSP(eval, filter, term) = \overleftarrow{(\lhd_{Scatter} \bullet [|\ ((eval))\ |]_{nw} \bullet (_{filter}\rhd))}_{term}$$

The skeleton is a parallel functional whose programs evaluate, using the function $eval$, the fitness of each individual in the current population. The *Reduce* functional is used to gather the data and apply one or more *genetic operators* to a subset of m individuals. This introduces genetic diversity to the populations (using the $filter$ function). The termination condition is tested by the function $term$, which determines if sufficient generations have been produced.

4.2 The orbit skeleton

The orbit skeleton [29,18] belongs to the field of symbolic computation. Starting from an initial set of known "points" it recursively applies a set of *generators* and adds those generated points not already included in the original set until the transitive closure of the generators on the initial set is eventually computed. The behaviour of the pattern can be described by the following algorithm, where $G = \{g_1, \ldots, g_k\}$ is the generator set and $S = \{s_1, \ldots, s_n\}$ the initial set of points:

```
1 repeat
2   ∀s ∈ S
3     ∀g ∈ G
4       S_g = g(s)
5       ∀x ∈ S_g
6         if(x ∉ S)  then S := S ∪ x
7 until (no more items have been added to S)
```

The critical point in this algorithm is represented by the updates to S, as the **for all**s at lines 2 and 3 generate independent computations if the update of S (at lines 5–6) is not taken into account. Therefore the orbit skeleton may be implemented by a two stage pipeline where the first stage computes in parallel all the potential new items to be added to S and the second stage filters the duplicates and eventually updates the set S, before looping back to the first stage. Thus

$$OB(S,G) = \overleftarrow{(OBstep(S,G))}_{NewItemsAdded}$$

where $NewItemsAdded$ holds true if $OBstep(S,G)$ adds at least one new item in S, $OBstep(S,G)$ being defined as

$$OBstep(S,G) = \underbrace{\lhd_{Scatter} \bullet [|\ M\ |]_{nw'}}_{Stage_1} \bullet \underbrace{(_{FilterDup}\rhd) \bullet ((UpdateS))}_{Stage_2}$$

where $FilterDup$ filters duplicate items, $UpdateS$ inserts items in S and

$$M = \lhd_{Broadcast} \bullet [|\ ((g_i)), \ldots, ((g_k))\ |] \bullet \rhd_{Gather}$$

computes the new items from the same s_i applying all the different g_j.

It is worth pointing out that by substituting the initial $\lhd_{Scatter}$ with a $[|\lhd_{Scatter}, \lhd_{Broadcast}|]$ in $OBstep(S,G)$, both s and S will be available to compute in the first stage a local reduce to eliminate duplicates generated on the same s_i using different g_j in the same worker. M may then be expressed as

$$M = \lhd_{Broadcast} \bullet [|\ ((g_1)), \ldots, ((g_k))\ |] \bullet (_{FilterDup}\rhd)$$

and applying map promotion this may be rewritten as

$$M = \lhd_{Broadcast} \bullet [|\ ((g_1)) \bullet (_{FilterDup}\rhd), \ldots, ((g_i)) \bullet (_{FilterDup}\rhd)\ |] \bullet (_{FilterDup}\rhd)$$

thus resulting in a more parallel implementation of the overall $FilterDup$ reduction.

4.3 Network Packet Processing

Analyzing and classifying the type of traffic traversing a given network has always been of great interest. The identification of the application protocol (e.g. http, smtp, Skype, etc.) is performed by extracting the information contained in the packet header and payload.

To correctly extract the data carried by the protocol it is in general necessary to manage the expensive task of IP defragmentation and TCP stream reordering and reassembly. This kind of processing is typically implemented (at least in part) through dedicated hardware. However, full software solutions are more appealing because they are more flexible and economical.

The Network Packet Processing (NPP) pattern can be described as a *pipeline of parallel functional stages*. The number of pipeline stages depends on the specific application, but, typically, the first stage performs packet parsing, the second stage protocol management and the third stage the real processing on the network packets/flows. For simplicity, and without loss of generality, we assume that the protocol management phase is not computationally expensive, such that the second and third stages of the pipeline can be merged into a single stage:

$$NPP(f,h) \quad = \quad \underbrace{(_f\triangleleft) \bullet [|\ \Delta_1\ |]_{nw} \bullet (_{f^{-1}}\triangleleft)}_{Parsing(f)} \bullet \underbrace{(_h\triangleleft) \bullet [|\ \Delta_2\ |]_{nw'} \bullet \triangleright_{Gather}}_{Processing(h)}$$

The packet scheduling function f has output arity nw and it is such that f^{-1} exists. This means that we are able to schedule and collect packets in the exact same order, ensuring that the $Parsing$ nodes do not reorder packets.The packet scheduling function h is able to route all the packets belonging to the same network flow to the program (worker) of the $Processing$ functionals which is in charge of managing that network flow.

5 RISC-pb^{2}l prototype implementation

In Sec. 3 and Sec. 4 we showed how RISC-pb^{2}l is suitable for modelling the implementation of general purpose computing model patterns and domain specific patterns, respectively. In addition, we discussed how optimisations may be identified by considering RISC-pb^{2}l semantics and target architecture features. To assess the usability and efficiency of our approach we need a prototype implementation of the RISC-pb^{2}l building blocks. In [12] we have already used FastFlow [16] to implement the RISC-pb^{2}l components. Here we briefly recall how RISC-pb^{2}l components map onto FastFlow components, to introduce the experiments of Sec. 6 which have been run using our prototype FastFlow implementation of RISC-pb^{2}l (henceforth FastFlow RISC-pb^{2}l).

FastFlow[2] is a C++ based parallel programming framework built on top of POSIX threads aimed at providing the parallel programmer with a set of

[2] FastFlow has been under development since the early '10s as an open source project

RISC-pb^{2}l	FastFlow
$((f))$	`ff_node` (see Fig. 4 bottom, left)
$(\mid f \mid)$	`ff_node` with encapsulated parallel code (e.g. offloading code to GPUs in the `svc` body, or using some OpenMP directive) (see Fig. 4 bottom right)
$\lhd_{Pol}$, $(_f\lhd)$	`ff_loadbalancer` implementing the required Pols and fs
$\rhd_{Pol}$, $(_f\rhd)$	`ff_gatherer` implementing Pols and fs
$\overleftarrow{(\Delta)}_c$	`wrap_around` method for farms and pipes
$[\mid \Delta \mid]_n$, $[\mid \Delta_1, \ldots, \Delta_n \mid]$	`ff_farm`, with possibly different workers (see Fig. 4 top right)
$\Delta_1 \bullet \ldots \bullet \Delta_n$	`ff_pipeline` (see Fig. 4 top left)

Fig. 3: Correspondence between RISC-pb^{2}l building blocks and FastFlow constructs/abstractions

pre-defined algorithmic skeletons modelling the main stream-parallel patterns [4,3]. It provides two algorithmic skeletons: i) a *farm* skeleton, applying in parallel the function modelled by an inner skeleton composition (the farm worker) to all the items appearing on an input stream, and delivering results to its output stream; and ii) a *pipeline* skeleton, applying in sequence the functions implemented by its inner skeleton compositions (the pipeline stages) to the items appearing on an input stream, and delivering the results to its output stream. Both pipelines and farms, when used at the topmost level in the skeleton composition modelling a parallel application, support a `wrap_around` method providing the programmer with the possibility to move data back from the output stream directly to the input stream.

The whole FastFlow implementation is built on top of the abstraction of parallel concurrent activity (the `ff_node`) processing items appearing onto an input stream to deliver outputs to an output stream. Additional building blocks `ff_loadbalancer` (scheduling tasks from an input stream to a set of concurrent activities) and `ff_gatherer` (collecting results from a set of concurrent activities into a single stream, either aggregated in a collection data structure or a sequence of values) are used to implement the various high level skeletons provided as primitive classes to the application programmer.

The RISC-pb^{2}l building blocks have been implemented using the FastFlow components as outlined in Fig. 3.

6 Experimental results

To assess the performance and versatility of the FastFlow RISC-pb^{2}l set, we performed two kinds of experiment aimed at demonstrating the efficiency of the FastFlow RISC-pb^{2}l on: i) simple applications or kernels, and ii) domain specific and general computing model patterns. We used the scalability metric, defined as $S(n, sz) = T(1, sz)/T(n, sz)$ where $T(n, sz)$ is the parallel execution time using n worker threads and keeping fixed the problem size sz, as a measure of efficiency of the RISC-pb^{2}l mechanisms.

```
int main() {
 ...
 ImageReader IR(...);
 Filter fA(...),fB(...);
 ImageWriter IW(...);
 ff_pipeline appl;
 appl.add_stage(&IR);
 appl.add_stage(&fA);
 appl.add_stage(&fB);
 appl.add_stage(&IW);
 ...
 appl.run_and_wait_end();
 ...
}
```

```
int main() {
 ...
 ff_farm<> farm;
 std::vector<ff_node *> W;
 for(int i=0;i<nworkers;++i)
   W.push_back(new SpellChecker(...));
 farm.add_workers(W);

 ff_pipeline appl;
 appl.add_stage(new FileListGen(...));
 appl.add_stage(&farm);
 appl.run_and_wait_end();
 ...
}
```

```
class seqWrapper:public ff_node {
 ...
 void* svc(void* x) {
  InParams*  p = (InParams *) x;
  OutParams* o = new OutParams;
  f(p,o);
  return((void*)o);
 }
 ...
}
```

```
class parWrapper:public ff_node {
 ...
 void* svc(void* x) {
  X=(float*)x;
  #pragma omp parallel for
  for(int i=0; i<NW; i++)
   X[i] = ...;
  ...
  return(x);
 }
```

Fig. 4: Sample FastFlow code: image filtering pipeline (top left) and file spelling checker farm (top right), using an `ff_node` to wrap sequential code (bottom left) and parallel code using OpenMP (bottom right).

6.1 RISC-pb^{2}l general efficiency

To assess scalability, we consider two applications. The first is a simple streaming application: two image filters (blur and emboss) are to be applied to a stream of input GIF images. In the test we considered 256 images and 2 cases: small size images (256KB) and coarser size images (1.7MB). The application was executed on a small cluster of 2 Intel Sandy Bridge multi-core (16-core 2-way hyper threading) connected through an Infiniband network. The application has been designed as a pipeline of two consecutive map filters fused into a a map of pipelined filters:

$$IMGfilter(f_1, f_2) = \lhd_{Unicast(AUTO)} \bullet [|\ ((f_1))\ |]_{nw} \bullet [|\ ((f_2))\ |]_{nw} \bullet \rhd_{Gather}$$

where the channels connecting the output of the workers of the first parallel building block ($[|\ ((f_1))\ |]_{nw}$) to the input of the workers in the second parallel building block ($[|\ ((f_2))\ |]_{nw}$) have been implemented as *unicast* channels over the Infiniband interconnection network (using the *IP over IB protocol*) rather than as standard FastFlow unicast channels in memory. Fig. 5 shows the structure of the concurrent activities of the application (Left) and typical performance results obtained (Right). The maximum speedup obtained is 19.6× and 28.3× for small and medium size images, respectively. The performance

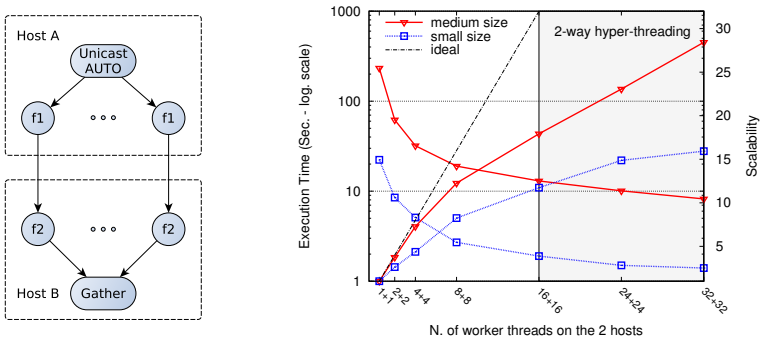

Fig. 5: Image processing application on 2 Intel Sandy Bridge multi-core nodes (16 cores each) connected via an Infiniband (40 Gb/s ports) network.

obtained is very good for coarse images taking into account the non-negligible amount of data that must be transferred across the network links interconnecting the two hosts implementing the two parallel building blocks.

The second application is a data-parallel kernel computing the standard dense matrix multiplication algorithm ($C = A \times B$) for square matrices of 4046×4096 double precision elements. The scalability of the kernel has been measured on a 60-core Intel Xeon PHI coprocessor running at a clock frequency of $1GHz$. The Xeon Phi coprocessor is a SMP computing node connected to the host domain through a PCI Express (PCIe) bus. It runs an embedded Linux x86_64 OS that provides basic functionality such as process/thread creation, scheduling and memory management. Xeon Phi cores run independently of each other, having support for 4-way HW multi-threading and vector processing. Each core has a private L2 cache that is kept fully coherent by a global-distributed tag directory. Cores are connected by the ring interconnect.

The RISC-pb^{2}l parallel structure of the algorithm is given by a standard map modelled as:

$$\triangleleft_{Scatter} \bullet [|\ ((MM))\ |]_{nw} \bullet \triangleright_{Gatherall}$$

where the *Scatter* splits the input A matrix into row chunks and directs the pairs $\langle RowChunk(A)_i, B\rangle$ to the parallel workers (the matrix B is broadcast); the workers compute the sequential algorithm on the input blocks producing $RowChunk(C)_i$; and finally, the *Gatherall* rebuilds the result matrix C. Fig. 6 outlines the application structure and shows the performance obtained when varying the number of worker threads up to 224 using the FastFlow map skeleton (in which communications are implemented via shared memory by passing pointers to data structures) and an OpenMP version of the same algorithm (Intel C++ Compiler XE 13.1)[3]. The maximum scalability on the Xeon PHI

[3] OpenMP thread affinity has been set using: KMP_AFFINITY="granularity=fine,scatter". In the tests we used the static scheduling policy for OpenMP.

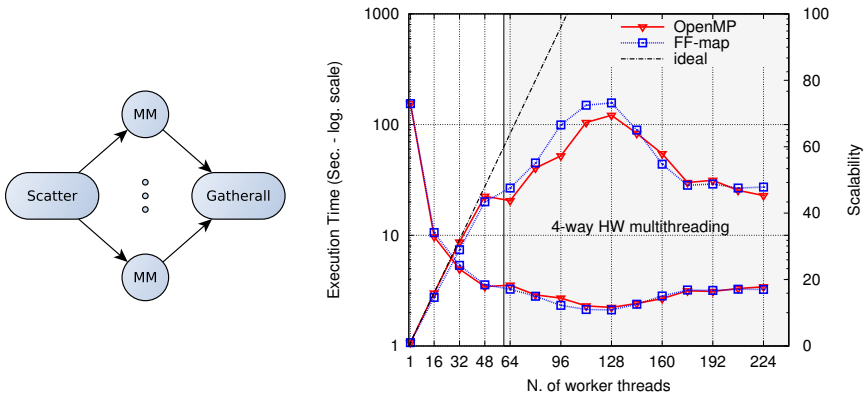

Fig. 6: Matrix multiplication on the Intel Xeon PHI coprocessor (60 physical cores, 240 HW threads).

coprocessor is 73.19× using 128 worker threads corresponding to an overall execution time of 2.12s. The same test executed on the host platform, a dual 6-core Xeon E5-2630 @2.3GHz, obtains a minimum execution time of 5.36s using 24 threads and a scalability of 11.6. We obtained a 2.5× gain in using the Xeon PHI coprocessor and a speedup of 29.3 with respect to the sequential time on the host platform.

6.2 RISC-pb^{2}l versatility

As testbed, we consider the Network Packet Processing pattern (NPP) and the macro-dataflow pattern (MDF) described in Sec. 3.3 and in Sec. 4.3, respectively. The implementations of both patterns use the FastFlow RISC-pb^{2}l framework. The tests have been executed using a single multi-core platform with 2 CPUs with 8 cores each (16 cores, 2-way multi-threading) Intel Sandy Bridge Xeon E5-2650 2.0GHz with 20MB L3 cache and 32GB of RAM.

For the NPP pattern we report the scalability of network packet parsing, implemented using a farm skeleton. Tasks passing through a farm can be subjected to reordering because of different execution times in the worker threads. To overcome the problem of sending packets in a different order with respect to input, tasks can be reordered after collection from the workers, although this solution might induce extra latency caused by the TCP normalisation.

To avoid this problem a scheduling and collection strategy which preserves the order of the packets is required. In Fig. 7 (Right) we compare two non order-preserving strategies (RR and AUTO), with a very simple order-preserving RR scheduling policy where a function–executed both in the emitter and in the collector of the farm–schedules/collects packets to/from workers in a strict round-robin order. In the graph we plot the scalability of the three

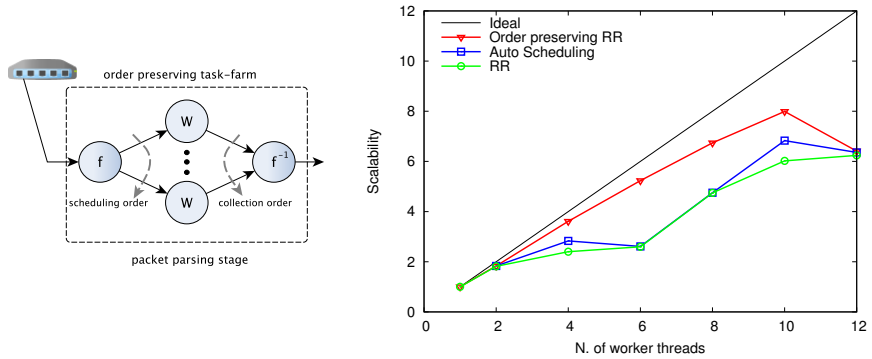

Fig. 7: Comparing different scheduling strategies for NPP's farm pattern.

strategies varying only the number of worker threads (W in Fig. 7 (Left)) up to 12 since other threads are used by the application run-time.

As can be seen, the ordering scheduling strategy provides better and more stable results than the other two solutions. This is because the overhead introduced by the ordered RR is less than the extra latency introduced by the explicit user reordering of packets in the farm collector. This can be explained by observing that, for the packet parsing phase, the packets already arrive at the farm collector almost in the correct order due to the small variance introduced during packet computation.

To validate the implementation of the macro-dataflow pattern (MDF) when complex fine-grained data-parallel computations are considered, we compared the block Cholesky factorisation algorithm operating on a single input matrix of complex elements (single precision) against the PLASMA 2.5.0 library (Parallel Linear Algebra for Scalable Multi-core Architectures) [27] implementation of the same algorithm. We implemented the MDF pattern as described in Eq. 1 in Sec. 3.3. The FastFlow implementation schema of the MDF pattern is sketched in Fig. 8) (Left).

Fig. 8 (Right) shows the execution times obtained when varying the number of worker threads in the farm stage for the MDF pattern and the number of threads in the PLASMA library for different matrix sizes (1024×1024 with 64×64 blocks, 2048×2048 with 256×256 blocks and 4096×4096 with 512×512 blocks). As can be seen, the results obtained using the MDF implementation are comparable with or better than those obtained using the highly optimized and very efficient static pipeline implementation offered by the PLASMA library[4]. As expected, when more worker threads than available cores are used, there is no performance gain for this kind of application which makes heavy use of vectorisation and floating point operations. On the contrary, the hyper threading support allows efficient use of more threads than physical cores in

[4] The PLASMA library can be considered the current reference implementation in Dense Linear Algebra for modern multi-core.

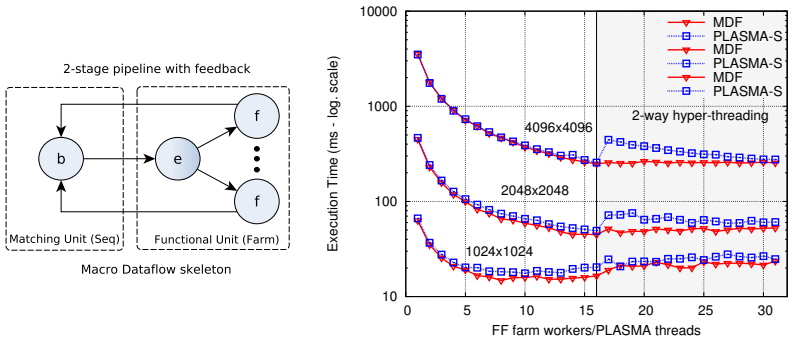

Fig. 8: FastFlow MDF skeleton vs. PLASMA's static implementation of the block Cholesky factorization algorithm.

the MDF skeleton. In fact, for bigger matrices (2048×2048 and 4096×4096) where the minimum execution time is obtained with 16 worker threads, the MDF pattern uses 18 threads, with the 2 extra threads used for managing the dependency graph generation (b) and task scheduling (e), respectively.

7 Related work

Since multi-core processors are becoming available in almost all computing platforms, the problem of developing usable and efficient parallel programming frameworks is garnering much interest in both research and industry.

Intel Threading Building Blocks (TBB) [30] is a C++ template library which provides easy development of concurrent programs by exposing (simple) skeletons and parallel data structures used to define tasks of computations. TBB is designed as an application-level library for shared-memory programming only; furthermore it does not provide any formal definition of its own skeletons to support global optimisations of the code.

The Task Parallel Library [31], which targets the .NET framework, supports parallel programming activities through the definition of skeletons such as parallel loops and parallel containers, but it neither supports the definition of new patterns out of the provided set of skeletons nor the optimisation of existing pattern compositions.

S-Net [17] is a declarative coordination language describing communication of asynchronous sequential components (a.k.a. boxes) through strongly typed streams. It separates concurrency management from application code and is designed to facilitate the composition of components developed in isolation. The S-Net run-time system is process based with blocking semantics, which limits scalability for fine-grain computation. Despite some similarities, RISC-pb^{2}l differs from S-Net in several respects: i) it (RISC-pb^{2}l) aims to facilitate implementation of parallel patterns rather than directly supporting

parallel applications; ii) it is focused on performance issues rather than on formal aspects; and iii) it provides finer grain constructs modelling orchestration of concurrent activities.

MPI is often considered as a solution for writing efficient parallel applications and frameworks [26]. The low-level approach advocated by MPI falls short in supporting performance portability, especially when hundreds or thousands of concurrent activities are involved and hybrid solutions have to be adapted (i.e. MPI+OpenMP). Applications must often be re-designed or manually tuned for each new platform by an expert parallel programmer.

OpenCL compilers and libraries target both multi-core CPUs, GP-GPUs and accelerators [20]. OpenCL is quite low level, focusing on low level feature management rather than high-level parallelism exploitation patterns.

8 Conclusions

In this paper we discussed how RISC-pb^{2}l, a set of efficient and re-usable building blocks for supporting efficient design of parallel skeleton/patterns, may be used to design and implement both general purpose programming model and domain specific patterns. We showed how optimization techniques[5] may be applied by rewriting RISC-pb^{2}l expressions *before* moving to actual implementation. We reported experimental results on different, modern multi-core architectures demonstrating: i) the scalability and efficiency of the RISC-pb^{2}l prototype implementation, and ii) how the RISC-pb^{2}l implementations of general purpose programming model patterns (macro-dataflow) and domain specific skeletons (network packet processing) deliver good performance.

References

1. E. Alba, G. Luque, J. Garcia-Nieto, G. Ordonez, and G. Leguizamon. MALLBA a software library to design efficient optimisation algorithms. *International Journal of Innovative Computing and Applications*, 1(1):74–85, 2007.
2. M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Targeting distributed systems in FastFlow. In *Euro-Par 2012 Workshops, Proc. of the CoreGrid Workshop on Grids, Clouds and P2P Computing*, volume 7640 of *LNCS*, pages 47–56. Springer, 2013.
3. M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating code on multi-cores with FastFlow. In *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, France, Aug. 2011. Springer.
4. M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, Jan. 2013.
5. M. Aldinucci, M. Meneghin, and M. Torquati. Efficient smith-waterman on multi-core with FastFlow. In *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, Pisa, Italy, Feb. 2010. IEEE.
6. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[5] the `OptNN` rules in Sec. 3

7. J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, Aug. 1978.
8. D. Buono, M. Danelutto, and S. Lametti. Map, reduce and mapreduce, the skeleton way. *Procedia CS*, 1(1):2095–2103, 2010.
9. P. Ciechanowicz and H. Kuchen. Enhancing muesli's data parallel skeletons for multi-core computer architectures. In *HPCC*, pages 108–113. IEEE, 2010.
10. M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.
11. M. Danelutto. QoS in parallel programming through application managers. In *Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing*, pages 282–289, Lugano, Switzerland, Feb. 2005. IEEE.
12. M. Danelutto and M. Torquati. A RISC building block set for structured parallel programming. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing PDP-2013*. IEEE Computer, 2013. pages 46–50.
13. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.
14. J. Enmyren and C. W. Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proc. of the 4th Inter. workshop on High-level parallel programming and applications*, HLPP '10, New York, NY, USA, 2010.
15. S. Ernsting and H. Kuchen. Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *Int. J. High Perform. Comput. Netw.*, 7(2):129–138, Apr. 2012.
16. FastFlow project website, 2013. `http://sourceforge.net/projects/mc-fastflow/`.
17. C. Grelck, SB. Scholz, and A. Shafarenko. Asynchronous stream processing with s-net. *International Journal of Parallel Programming*, 38(1):38–67, 2010.
18. K. Hammond, A. A. Zain, G. Cooperman, D. Petcu, and P. Trinder. Symgrid: A framework for symbolic computation on the grid. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 457–466, 2007.
19. N. Javed and F. Loulergue. Osl: Optimized bulk synchronous parallel skeletons on distributed arrays. In *APPT*, volume 5737 of *Lecture Notes in Computer Science*, pages 436–451. Springer, 2009.
20. Khronos Group. *The OpenCL Specification*, Sept. 2010.
21. M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *PDP*, pages 289–296. IEEE Computer Society, 2010.
22. B. L. Massingill, T. G. Mattson, and B. A. Sanders. Parallel programming with a pattern language. *STTT*, 3(2):217–234, 2001.
23. K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *Infoscale*, volume 152 of *ACM International Conference Proceeding Series*, page 13. ACM, 2006.
24. T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.
25. M. Nowostawski and R. Poli. Parallel genetic algorithm taxonomy. In *Proceedings of the Third International*, pages 88–92. IEEE, 1999.
26. P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.
27. *PLASMA library website*, 2013. `http://icl.cs.utk.edu/plasma/`.
28. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.
29. SCIEnce deliverable series. *Deliverable D5.13 (JRA 1.6) – Report on Multilevel Parallelism*, Jan. 2012.
30. Intel TBB for Open Source, 2012. `http://threadingbuildingblocks.org/`.
31. Task parallel library home page, 2012. `http://msdn.microsoft.com/en-us/library/dd460717.aspx`.
32. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.