Sample pattern implementation through RISC-pbb

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

December 9, 2013

1 The building block set

The building block set RISC-pb^{2}l proposed in [12] is composed of three kinds of building blocks: *wrappers*, *functionals* and *combinators* (see Tab. 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 different kinds of parallel computation, 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. 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 Tab. 1 formally describes legal combinations of RISC-pb^{2}l components.

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

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

takes one input (input arity of $\lhd_{Scatter}$), 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 $\lhd_{Scatter}$.

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.

2 The BSP model

In the Bulk Synchronous Parallel model [31], 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 Δ_{step_i}, as follows

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

where $routeToDest$ 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:

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

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 of not having more than h communications (incoming or outcoming)[1] incident upon the same node is violated and b) the node gathering the messages from the super-steps 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

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

[1] In order to satisfy the h-relation required by the BSP model

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 a 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, 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

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

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

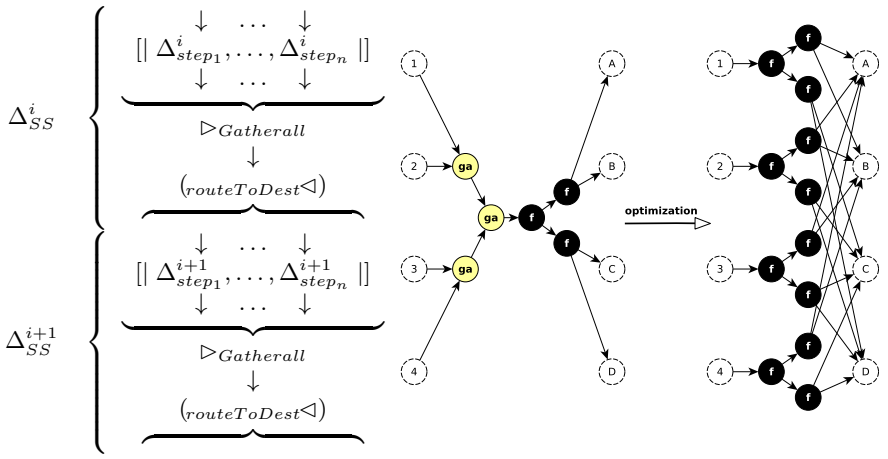

Figure 2: RISC-pb^{2}l modelling of the BSP superstep sequence (left). Optimizing the $\triangleright_{Gatherall} \bullet (_f\triangleleft)$ by $[|\ (_f\triangleleft)_1, \ldots, (_f\triangleleft)_n\ |]$ (f being *routeToDest*, *ga* being *Gatherall*) (right).

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–that is, it does not process the messages "as a whole"–the $\triangleright_{Gatherall} \bullet (_f\triangleleft)$ expression may clearly be transformed into an equivalent form were 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 optimizations 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 $[|\ (_f\triangleleft)\ |]_n$.

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}:

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

(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

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

3 The MapReduce

The MapReduce pattern (MR) introduced by Google models those applications where a collection of input data is processed in two steps [27, 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 mapreduce$(f, \oplus, (x_1, \ldots, x_n))$ 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[2].

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:

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

The *spread* over K $(_K\triangleleft)$ will actually route each $\langle K, Y\rangle$ item to the parallel activity actually dealing with all items with $Key = K$, that is to the parallel activity (worker) computing the $(_\oplus\triangleright)$ of the keys K. Logically $nw' =$

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

$\#\{keys\_of(f(x_1), \ldots, f(x_n))\}$ 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 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

$$[|\ \Delta_1\ |]_n \bullet [|\ \Delta_2\ |]_n \equiv [|\ \Delta_1 \bullet \Delta_2\ |]_n \qquad \texttt{(Opt2)}$$

a further optmization 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 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.

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.

[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] 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.

[18] 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.

[19] Khronos Group. *The OpenCL Specification*, Sept. 2010.

[20] M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *PDP*, pages 289–296. IEEE Computer Society, 2010.

[21] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Parallel programming with a pattern language. *STTT*, 3(2):217–234, 2001.

[22] 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.

[23] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[24] M. Nowostawski and R. Poli. Parallel genetic algorithm taxonomy. In *Proceedings of the Third International*, pages 88–92. IEEE, 1999.

[25] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[26] *PLASMA library website*, 2013. `http://icl.cs.utk.edu/plasma/`.

[27] 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.

[28] SCIEnce deliverable series. *Deliverable D5.13 (JRA 1.6) – Report on Multilevel Parallelism*, Jan. 2012.

[29] Intel TBB for Open Source, 2012. `http://threadingbuildingblocks.org/`.

[30] Task parallel library home page, 2012. `http://msdn.microsoft.com/en-us/library/dd460717.aspx`.

[31] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.