Structured Data Access Annotations for Massively Parallel Computations*

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

[1] Computer Science Department, University of Torino, Italy
aldinuc@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. We describe an approach aimed at addressing the issue of joint exploitation of control (stream) and data parallelism in a skeleton based parallel programming environment, based on annotations and refactoring. Annotations drive efficient implementation of a parallel computation. Refactoring is used to transform the associated skeleton tree into a more efficient, functionally equivalent skeleton tree. In most cases, cost models are used to drive the refactoring process. We show how sample use case applications/kernels may be optimized and discuss preliminary experiments with FastFlow assessing the theoretical results.

Keywords: algorithmic skeletons, parallel design patterns, refactoring, data parallelism, cost models.

1 Introduction

The structured parallel programming approach has abstracted the concept of control and data parallelism by means of *skeletons* [10], which are well known *patterns of control* [8]. Control parallelism is conceived, designed and implemented as a graph of nodes (a skeleton), each node representing a function. A stream of independent tasks flows through the graph: when each node's inputs are available it computes producing output which is sent to its connected nodes. On the other hand, *data parallel skeletons* describe a pattern of computation defining how to access data in parallel and the function which has to be applied to data partitions to get the final result. Traditionally, orthogonality between control parallelism and data parallelism has been dealt with using two-tier models in which control/stream-driven approaches were enhanced with data parallel capabilities, possibly with parallel data structures exposing collective operations [13] and vice versa. However, control parallel and data parallel oriented approaches

* This work has been supported by European Union Framework 7 grant IST-2011-288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems".

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 381–390, 2013.

often lack the ability to describe efficiently applications in which both concerns are exploited because of the intrinsically different means by which parallelism is expressed and, sometimes, optimized. An efficient distribution of tasks in a control driven environment could be invalidated by a poor data access policy, and vice versa [14].

In this paper we sketch a new approach to confronting the control versus data parallel dichotomy based on the idea that: *i)* data and control parallel concerns needs to be independently expressed since they describe orthogonal aspects of parallelism, and *ii)* data accesses and control parallel patterns need to be *coordinated* in order to efficiently support the implementation of parallel applications. While exploiting parallelism through patterns is not a new approach [11] and coordination efforts have been made in the past in terms of languages or frameworks[17,12], the idea proposed in this work is that such coordination can be expressed by annotating the graph of control implicitly defined by the skeletons with information regarding the data access. Moreover, we will show how such annotations can be used to drive optimizations in the implementation and execution of the graph.

2 The Skeleton Framework

The skeleton system considered includes control (i.e. stream) and data parallel skeletons, modelling the more common and general parallelism exploitation patterns. Our skeleton set is defined by the following grammar:

$$\begin{aligned} Skel ::= & \, Seq(\langle id \rangle) | Pipe(Skel, Skel) | Farm(Skel) | \\ & Map(Skel, Splitter, Composer) | Reduce(Skel) \end{aligned}$$

These skeletons represent well-known parallelism exploitation patterns[4]: Seq wraps existing sequential code, Pipe/Farm are stream parallel skeletons processing streams of items and Map/Reduce are data parallel skeletons processing collections of data. In contrast with many skeleton frameworks (including SkeTo [16], Muesli [9] and SkePU) which consider only maps over "collection" input data, we assume the one used in P3L [7] and Skandium [15]: the responsibility for specifying how the subtask items are build out of the input data (set) is left to the application programmer, as is the specification of the re-construction of the result from the collection of partial results. In P3L, the programmer is asked to use the formal parameters of the map as actual parameters of the worker skeleton using "star variables"–a kind of $\forall i$ variable–to establish correspondences between the task and the subtask data items. For example, a matrix multiplication map could be defined as:

```
1  map MM in(float a[N][N], b[N][N]) out(float c[N][N])
2    IP in(a[*i][], b[][*j]) out(c[*i][*j])
3  end map
```

The star variables were logically interpreted as forall loop variables. In this case the calls to the inner product worker skeleton, IP, corresponded to the pseudocode: $\forall i \in [0, N-1]\ \forall j \in [0, N-1]$ `call(IP,a[i][],b[][j],c[i][j])` although the

schedule eventually produced by the P3L compiler may have been completely different from the (sequential) schedule implicit in the nested loops. When dealing with collections and complex and compositional data structures, there are some particular data access patterns that recur which describe how each piece of data is combined into the final result. For example, a block of contiguous data implementing a matrix in a "row major" memory organization, could be accessed by rows or by columns, each row could be coupled with every column, with all the couples becoming targets of computation and the output of each such computation representing a single position in the output matrix. Variants of this kind of pattern include those considering each row/column coupled with a whole matrix or with a sub-block. Another pattern of access is that describing stencil, i.e. a block of cells in a fixed or variable range around each item. Some patterns deal with triangular matrices as, for example, the one accessing diagonals or stencils relative to elements on the diagonal.

In our proposal, control parallelism is described by a composition of control and data parallel skeletons (i.e. patterns of computation) but the corresponding graph is enriched by a set of annotations exposing data access patterns. The different combinations of skeleton type and access patterns can fully describe how computation evolves by guaranteeing the orthogonal management of both data and control parallelism and, at the same time, providing a theoretical platform on which we can built optimization strategies for better exploitation of resources, bandwidth, service time, and other performance measures. In the following section we will provide a language for defining data and control concerns and we will underline how they can be orthogonally described in order to facilitate skeleton rewriting and subsequent skeleton implementation.

3 Annotations/Metadata

Each of the skeletons introduced in Sec. 2 may be enhanced using various kinds of annotation, represented as metadata associated with the skeleton tree. In particular, we use annotations related to the functional (e.g. data access patterns) and non-functional (e.g. performance related) aspects. Such annotations are expressed using the following grammars:

$$
\begin{array}{ll}
ParDegreeAnnot & ::= \texttt{pardegree}(\langle int \rangle) \\
CodeAnnot & ::= \texttt{sourcecode}(< string >) \ |\texttt{library}(< string >) \\
ArchTypeAnnot & ::= \texttt{GPU}|\texttt{CPU} \\
DataAccessAnnot & ::= AccessKind\ \langle id \rangle\ \texttt{by}\ AccessType \\
AccessKind & ::= \texttt{READ}|\texttt{WRITE} \\
Accesstype & ::= \texttt{ROW}|\texttt{COL}|\texttt{ITEM}|\texttt{BLOCK}
\end{array}
$$

The informal semantics associated with the annotations is as follows:

ParDegreeAnnot parallelism degree (if variable)
CodeAnnot associates source and library code with a sequential skeleton.
ArchTypeAnnot target architecture where the skeleton has to be (preferably) executed (heterogeneous CPU/GPU processing element assumed)

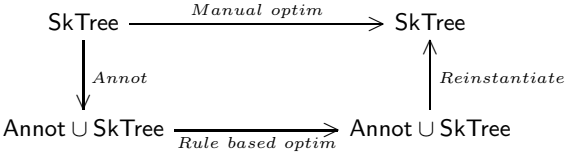

Fig. 1. Optimization workflow

DataAccessAnnot kind of accesses performed on the input data.

Part of these annotations are provided by the user (e.g. the `CodeAnnot` ones). Others are derived directly from the skeleton source code via a compilation step (e.g. the `DataAccessAnnot` ones). A third class of annotations may be either provided by the application programmer or automatically derived by the compiling tools (e.g. the `ParDegree` ones).

The focus here is on the `DataAccessAnnot` annotations. These are used to optimize the execution of a skeleton program as detailed in Sec. 4 relative to data placement in memory, to communication and synchronization management and also to computation partitioning among the available processing elements. The general idea is summarized in Fig. 1. The skeleton program provided by the application programmer is given to a compiler tool (the "Annot" arrow in the Figure) which produces an annotated skeleton tree. This compiler tool is a kind of abstract interpreter. The annotated skeleton tree is parsed and navigated by the optimizer which eventually produces a different skeleton tree. This new skeleton tree may differ in both annotations (e.g. it has the same shape as the original one but hosts different annotations) and tree shape (e.g. it hosts different, or differently connected, nodes). The optimization phase (the "Rule based optim" arrow at the bottom of Fig. 1) uses different heuristics stored as refactoring rules, possibly identifying, among the possible rewritings, the one giving the best performance figures according to a given skeleton performance model.

As illustrative example, we will show how our abstract syntax and annotation system can be used to write both pure control/stream-parallel applications and data parallel ones, highlighting the syntax usage, the expressive power and how annotation and parameters can support reasoning about skeleton trees. In Section 4 we will go a step further, using annotations in applications exposing *both* stream and data parallel concerns.

Control Parallelism. A pure control parallel skeleton employs stream parallel skeletons only, for example, $Pipe(Seq(f), Farm(Seq(g), Seq(h))$. Fig. 2 shows a possible instantiation of the corresponding syntax tree. The *Seq* skeleton is annotated with a path reference to the code for the sequential function. The *Farm* skeleton is provided with two parameters (the number N of workers and the skeleton implementing each replica) and will be annotated with the actual parallelism degree. *Pipe* is defined in terms of its stages as parameters.

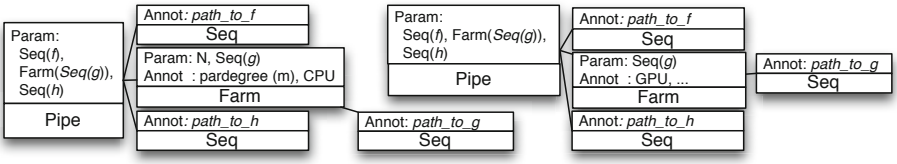

Fig. 2. The syntax tree of Pipe(Seq, Farm(Seq), Seq)

In heterogenous architectures (for instance, those provided with CPUs and GPUs) the farm tree could further be annotated with the available alternatives.

Data Parallelism. Matrix multiplication is a traditional example of data parallelism since it can be written as a map whose function is represented by the sequential inner product applied in parallel to each row of the input matrix A coupled with each column of the input matrix B and getting a single item of a matrix C as result[1]. From the expression $Map(Seq(IP), \underline{in}\ A[*i][], B[][*j], \underline{out}\ C[i][j])$, the annotated tree in Fig. 3-left can be derived. Note that the expression $A[*i][]$

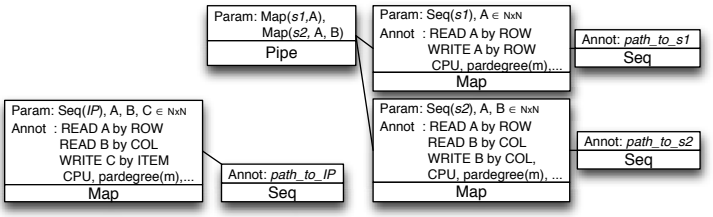

Fig. 3. Left: the syntax tree of Map(Seq(IP)); Right: a more complex syntax tree

denotes a matrix A accessed by rows and, as a consequence, it defines a *set* of blocks, each represented by one of those rows. For example, considering a matrix A with elements a_{ij} then $A[*i][] = \{[a_{11}, a_{12}, a_{13}], [a_{21}, a_{22}, a_{23}], [a_{31}, a_{32}, a_{33}]\}$. In the case of the expression $A[][*i]$–denoting access by columns–the set will change accordingly in $A[][*i] = \{[a_{11}, a_{21}, a_{31}], [a_{12}, a_{22}, a_{32}], [a_{13}, a_{23}, a_{33}]\}$. We will employ set operators, where needed. So, for example, we may write that $A' \subset A[*i][]$ if and only if A' is a subset of $A[*][i]$. Moreover, in the following section, when the structure of such a set is not relevant, we will simply denote it as a result of the *splitting* or *combining* function inside a map definition. Thus, we can write $Map(f, sp, \underline{out}\ A[i][j])$ for some sp, to denote *any* kind of reading access.

[1] In our syntax, $\underline{in}$ and $\underline{out}$ are keywords introducing the map parameters, that is defining the split and compose map policies.

4 From Metadata to Optimization

We now show by examples how the abstract syntax may be used to write applications incorporating both control and data parallelism by coordinating orthogonal concerns and to drive skeleton rewriting in such a way that optimizations are achieved.

Access Driven Optimization. Suppose to have the following abstract description:

$$Pipe(Map(Seq(s1), in\ A[*i][], out\ A[*i][]), Map(Seq(s2), in\ B[][*j]\ A[*i][], out\ B[][*j])$$

for source code $s1$ and $s2$ and $N \times N$ matrices, A and B. The definition allows us to annotate the skeleton tree with the information related to the two maps as depicted in Fig. 3, right.

In the example, a pipe is composed by two maps `m1` and `m2`. Both access the same dataset `A` (a matrix) and apply a function on its rows in sequential stages; `m2` takes also the data set `B` as input. From [6,2,3] the following rule holds

$$Pipe(Map(f_1, sp_1, cm_1), Map(f_2, sp_2, cm_2)) \equiv Farm(Map(Comp(f_1, f_2), sp', cm'))$$

where the *Comp* skeleton computes in sequence the two functions. However, from the annotation provided by the data access we can argue that *i)* `m2` is functionally dependent on `m1` since it accesses matrix `A`, written by `m1`; *i)* `m2` accesses `A` by the same policy used by `m1` to write it, i.e. they access the matrix by row and extend the input data space by accessing matrix `B`, too. Both conditions can be formally defined by the inclusion $sp_2 \supset cm_1$, since the set of annotations defined by sp_2 in reading mode includes those defined by cm_1 for writing mode. As a consequence, we could merge the two maps in order to save read and write memory accesses and to have a single map *i)* whose computing elements take as input the whole input data (the union of both map input data); *ii)* their computation function is the composite of the previous one, as the rewriting rule suggests. The rewriting process leads to the configuration of a *new* instance of annotated skeleton tree. Such example gives a first idea of a transformational rule that may expressed as follows

$$\frac{sp_2 \supset cm_1}{Pipe(Map(f_1, sp_1, cm_1), Map(f_2, sp_2, cm_2)) \equiv Farm(Map(Comp(f_1, f_2), sp_1, cm_2))}$$

In other words this rule introduces some access contraints to the applicability of the well-known transformation rule.

Architecture Driven Optimization. As already suggested in [1] the skeleton tree could be annoted also with information related to the target architecture at hand in order to optimize mappings and/or distribution of data. As an example, let us consider a system S provided with n CPUs and r GPUs defined as $S = \{cpu_1, \ldots, cpu_n, gpu_1, \ldots, gpu_r\}$ and the following skeleton definition, for some source code f. As already seen in Fig. 3, the abstract syntax tree of such definition could of a map whose computing elements are located each on a separate CPU, thus involving a huge amount of data transfer, since the matrix is not

shared and parallelism is exploited in terms of every single item of the matrix. As an alternative to this schema, taking advantage of the GPU subsystem, the skeleton could be rewritten as a map of two maps (i.e. how many the number of GPUs), each computing on a block of data as defined by the language. Thus, the preceding portion of code is represented by

$$Map_{CPU}(Map_{GPU}(s, inA'[*i][], outA'[*i][]), inA[*^r][], outA[*^r][])$$

where $*^r$ specifies the distribution of r partitions of A's rows. In other words, Map_{CPU} distributes A by r partitions (blocks) of rows; each partition is taken as input by Map_{GPU} which applies its own policy (distribution by rows) on its block. The syntax tree is represented in Fig. 4, Left which is a map of r Maps

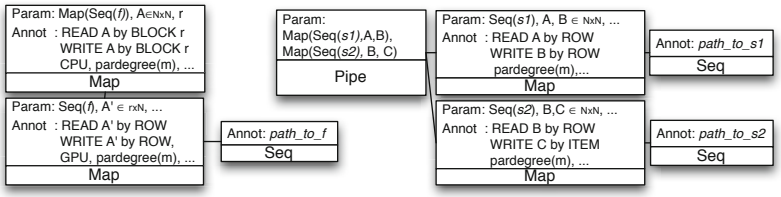

Fig. 4. Map of map on GPU (left); Pipe of two Map (right)

(r being a parameter of Map_{CPU}), each computing a block of data as defined by the language, on a GPU each. Assuming the availability of a cost model $C(Map_{cpu}(Map_{gpu}(s)))$ able to qualify the performance of both skeleton trees, our system can be enriched by the following transformation for some skeleton s

$$\frac{C(Map_{cpu}(s, sp_1, cm_1)) > C(Map_{cpu}(Map_{gpu}(s, sp_2, cm_2), , sp'_1, cm'_1))}{Map_{cpu}(s, sp_1, cm_1) \rightarrow Map_{cpu}(Map_{gpu}(s, sp_2, cm_2), sp'_1, cm'_1)}$$

asserting that the benefit of moving the execution from CPUs to GPUs depends on provisional costs sustained in the two access policies.

Operator Driven Optimization. With this example we will highlight that our system of rules could be enriched by operators able to manipulate data in order to drive optimizations at an abstract level, thus hiding implementation details.

Let us consider the transposition operator which, given a matrix A, defines a new matrix A^T such that $\forall i, j.A^T[i, j] = A[j, i]$ and $(A^T)^T = A$. Let us suppose that we have the following skeleton definition

$$Pipe(Map(s1, \textit{ in } A[*i][], \textit{ out } B[*i][]), Map(s2, \textit{ in } B[][*j], \textit{ out } C[i][j]))$$

for some skeleton $s1, s2$ and annotated as depicted in Fig. 4, Right. In this example, we meet a pattern in which one stage writes a matrix and the following stage reads its transposition. A possible optimization involves the first stage placing data into memory and/or computation elements in order to better exploit locality effects. However, this is a valuable strategy only if the block of data on

which the transposition operator has to be applied is small enough to be hosted at cache level; on the contrary, if memory accesses are required to solve cache misses in the reading phase, locality exploitation can raise the cost of referencing data. Both cases can be evaluated by some provisional, qualitative analysis that a cost model monitoring the execution could perform, on the basis of the target architecture and the actual instances of data and skeletons. For simplicity, we will denote using $\overline{C}$ a boolean function evaluating to *true* if such consideration is worthwhile in the context of the actual skeleton instance. Thus, the refactoring rule becomes

$$\frac{P_2 = P_1^T \wedge \overline{C}(P_1^T) \rightarrow true}{\begin{array}{c} Pipe(Map(f_1, sp_1, out\ P_1), Map(f_2, in\ P_2, cm_2)) \rightarrow \\ Pipe(Map(f_1, sp_1, out\ P_1^T), Map(f_2, in\ P_1^T, cm_2)) \end{array}}$$

where P_1 and P_2 represent blocks of data,P_1^T represents the transpose of P_1 and sp_1, cm_2 define generic splitting and combining policies not influencing P_1.

5 Experimental Results

We describe some experimental results aimed at validating the refactorings discussed in Sec. 4. The results discussed here have been achieved using FastFlow, our experimental skeleton framework targeting multicore architectures [5].

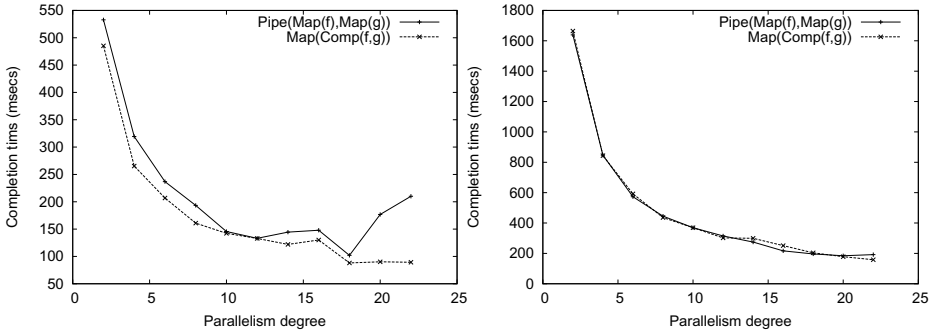

Fig. 5. Effect of map fusion refactoring: small grain (left) and large grain (right)

Access Driven Optimization. We implemented in FastFlow two versions of a program corresponding to the original $Pipe(Map(f), Map(g))$ structuring and to the structuring resulting from fusion, that is $Map(f, g)$, and we measured the performances on a 24 core Magny Cours (Opteron 6174) architecture[2]. The results are shown in Fig. 5. When the amount of time spent scattering and gathering data to/from the map workers is large enough with respect to the time spent in computing the single map worker (fine grain, left plot) map fusion clearly outperforms the original program. When the time spent in computing

[2] FastFlow currently does not support a primitive Map skeleton. We used a prototype implementation that will be available with the next FastFlow release.

map workers is considerably larger than the time needed to scatter and gather data to and from the workers, the fused version of the programs performs more or less like the original program. It is worth pointing out that as the parallelism degree increases (and therefore the computation grain decreases), the fused version becomes competitive with respect to the original program version.

Architecture Driven Optimization. The feasibility of refactoring code in such a way that a map originally targeting CPU cores only is transformed into a map targeting CPU cores and GPUs has already been demonstrated in [1]. There we have shown not only that using both CPU cores and GPUs improves the performance of programs with respect to the performances achieved when using only CPU cores, but also that an automatic scheduling procedure may be set up which dynamically uses GPUs and CPU cores to achieve optimal load balancing and, therefore, performances.

Operator Driven Optimization. We took into account the operator driven refactoring discussed at the end of Sec. 4. We implemented a two stage pipeline in FastFlow with both stages computing a function from bi-dimensional float arrays to bi-dimensional float arrays. The functions computed by the stages were very light. We measured the performances of the program run on three different architectures, namely an Intel Xeon Nehalem, an Intel i3 and AMD Magny Cours architecture. We developed two versions of the program: in the first version the first pipeline stage produces a matrix in "WRITE BY ROW" fashion and the second stage reads that matrix "READ BY COLUMN", while in the second version the first stage produces the transposed matrix and therefore the second stage processes it "READ BY ROW". The following table summarizes the resulting average computation times, in milliseconds.

	1st ($ByRow \rightarrow ByCol$)	2nd ($ByRow^T \rightarrow ByRow$)	% improvement
i3	8786.52	6303.34	28.26
Nehalem	7971.74	5886.94	26.15
Magny Cours	12918.99	11287.23	12.63

The times are for computation of a stream of 100 tasks, each relative to a 1024×1024 floating point matrix. The computation performed on each matrix is negligible–more or less of the "size" of an assignment–for both first and second stage. The refactored version clearly outperforms the original. The smallest performance improvement is on the Magny Cours architecture, which notably sports the smaller memory bandwidth among the three architectures considered here.

6 Conclusions

We outlined a skeleton framework with annotations supporting performance driven refactoring relying on the existence of both suitable skeleton performance

models and the capability to automatically refactor code via rewriting rules. We presented experimental results assessing the approach, which will be used within ParaPhrase WP2 ("Algorithmic skeleton") activities.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting heterogeneous architectures via macro data flow. PPL 22(2) (2012)
2. Aldinucci, M.: Automatic program transformation: The Meta tool for skeleton-based languages. In: Gorlatch, S., Lengauer, C. (eds.) Constructive Methods for Parallel Programming. Advances in Computation: Theory and Practice, ch. 5, pp. 59–78. Nova Science Publishers, NY (2002)
3. Aldinucci, M., Danelutto, M.: An operational semantic for skeletons. Technical Report TR-02-13, University of Pisa, Dip. Informatica, Italy (July 2002)
4. Aldinucci, M., Danelutto, M.: Skeleton based parallel programming: functional and parallel semantic in a single shot. Computer Languages, Systems and Structures 33(3-4), 179–192 (2007)
5. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating Code on Multi-cores with FastFlow. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 170–181. Springer, Heidelberg (2011)
6. Aldinucci, M., Gorlatch, S., Pelagatti, S., Lengauer, C.: Towards parallel programming by transformation: The fan skeleton framework. Par. Algorithms and Applications (2001)
7. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P^3L: A structured high-level parallel language, and its structured support. Concurrency - Practice and Experience 7(3), 225–255 (1995)
8. Bromling, S., MacDonald, S., Anvik, J., Schaeffer, J., Szafron, D., Tan, K.: Pattern-based parallel programming. In: Proc. of Int. Conf. on Par. Processing. IEEE Comp. Society, Washington, DC (2002)
9. Ciechanowicz, P., Poldner, M., Kuchen, H.: The muenster skeleton library muesli - a comprehensive overview (07) (2009)
10. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press, Cambridge (1991)
11. Díaz, M., Rubio, B., Soler, E., Troya, J.M.: Integrating Task and Data Parallelism by Means of Coordination Patterns. In: Müller, F. (ed.) HIPS 2001. LNCS, vol. 2026, pp. 16–27. Springer, Heidelberg (2001)
12. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM 35(2), 97–107 (1992)
13. Kuchen, H.: A Skeleton Library. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)
14. Kuchen, H., Cole, M.: The Integration of Task and Data Parallel Skeletons. PPL 12, 141–155 (2002)
15. Leyton, M., Piquer, J.M.: Skandium: Multi-core programming with algorithmic skeletons. In: Proc. of PDP, pp. 289–296. IEEE Comp. Society (2010)
16. Matsuzaki, K., Iwasaki, H.: A library of constructive skeletons for sequential style of parallel programming. In: InfoScale 2006, p. 13. ACM Press (2006)
17. Rauber, T., Rünger, G.: A coordination language for mixed task and and data parallel programs. In: Proc. of the 1999 ACM Symposium on Applied Computing, SAC 1999, pp. 146–155. ACM, New York (1999)