Available online at www.sciencedirect.com

ScienceDirect

Parallel Computing 32 (2006) 449–462

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Algorithmic skeletons meeting grids ☆

Marco Danelutto *, Marco Aldinucci

Department of Computer Science, University of Pisa, Largo Pontecorvo 3, Pisa, Italy

Received 15 January 2006; received in revised form 11 June 2006; accepted 12 June 2006
Available online 8 August 2006

Abstract

In this work, we discuss an extension of the set of principles that should guide the future design and development of skeletal programming systems, as defined by Cole in his "pragmatic manifesto" paper. The three further principles introduced are related to the ability to exploit existing sequential code as well as to the ability to target typical modern architectures, those made out of heterogeneous processing elements with dynamically varying availability, processing power and connectivity features such as grids or heterogeneous, non-dedicated clusters. We outline two skeleton based programming environments currently developed at our university and we discuss how these environments adhere to the proposed set of principles. Eventually, we outline how some other relevant, well-known skeleton environments conform to the same set of principles.

Keywords: Structured parallel programming; Skeletons; Coordination languages; Macro data flow; Adaptivity; Heterogeneity

1. Introduction

When algorithmic skeletons were first introduced in late 1980 [1] the idea had an almost immediate success. Several research groups started research tracks on the subject and come up with different programming environments supporting algorithmic skeletons. Darlington's group first developed functional language embeddings of the skeletons [2] and then moved to FORTRAN [3]. Our group designed P3L, which is basically a sort of skeleton-based parallel C [4,5]. Kuchen started the work on Skil [6] and eventually produced the Muesli C++ skeleton library [7]. Serot designed Skipper [8,9], which exploits the macro data flow implementation model introduced in [10].

These groups almost completely embraced the original definition of skeleton programming environment given by Cole in his book [11]: *The new system presents the user with a selection of independent "algorithmic*

☆ This work has been supported by Italian FIRB *GRID.it* project (RBNE01KNFP) and by the European Network of Excellence CoreGRID (IS-2002-004265).
* Corresponding author. Tel.: +39 05022 12742; fax: +39 05 022 12726.
E-mail address: marcod@di.unipi.it (M. Danelutto).

skeleton", each of which describes the structure of a particular style of algorithm, in the way in which "higher order functions" represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton. In particular, the scientific community accepted the idea of a fixed selection of independent skeletons. All the above-mentioned skeleton systems developed in the 1990 only provide the programmer with a fixed set of skeletons.

The fixed, immutable skeleton set was both a source of power and weakness for the skeletal systems. It allowed efficient implementations to be developed, but also it did not allow programmers to express non-standard parallelism exploitation patterns; no matter if these are slight variations of the ones provided by the supplied skeletons. A partial solution to the unavailability of skeletons modeling specific parallel patterns came from the implementation of skeletons as libraries, whose mechanisms adopted to exploit parallelism was partially known, such as the ones discussed in [12,13]. In the former case, skeletons are provided as plain C function calls. The input data stream and the output data stream are implemented by plain Unix file descriptors that are accessible to the user. Therefore the programmer can program his own parallel patterns and make them interact with the predefined skeletons just writing/reading data to/from standard file (pipe, actually) descriptors. In the latter case, skeletons are provided as collective MPI operations. The programmer can access the MPI communicator executing the single parallel activity of the skeleton (e.g. one pipeline stage or a task farm worker) and can freely manage the processors allocated to the communicator. Explicit primitives allow the programmer to receive tasks from the skeleton input stream and to deliver results to the skeleton output stream. In both cases, a limited degree of freedom is left to the programmer to program his own parallelism exploitation patterns either outside or inside the ones modeled by skeletons.

2. Guidelines to design advanced, usable skeletal systems

Despite being around since long time and despite the progress made in skeletal system design and implementation, the skeleton systems did not take off as expected. Nowadays, the skeleton system usage is actually restricted to small communities grown around the teams that develop the skeleton systems.

Cole focused very well the problem in his *manifesto* [14]. Here he stated four principles that have to be tackled in skeletal systems to make them effective and successful:

1. *propagate the concept with minimal conceptual disruption*, that is skeletons must be provided within existing programming environments without actually requiring the programmers to learn entirely new programming languages;
2. *integrate ad-hoc parallelism*, i.e. allow programmers to express parallel patterns not captured by the available skeleton set;
3. *accommodate diversity*, that is provide mechanisms to specialize skeletons, in all those cases where specialization does not radically change the nature of the skeleton, and consequently the nature of the implementation;
4. *show the pay-back*, i.e. demonstrate that the effort required to adopt a skeletal system is immediately rewarded by some kind of concrete results: shorter design and implementation time of applications, increased efficiency, increased machine independence of the application code, etc.

   While the second and the third points are more specifically technical, the first and the last one are actually more "advertising" ones, in a sense. All these points, however, have impacts on both the way the skeleton systems are designed and on the way they are implemented.

   In this work, we propose to add three more principles to the original Cole set:
5. *support code reuse*, that is allow programmers to reuse with minimal effort existing sequential code;
6. *handle heterogeneity*, i.e. implement skeletons in such a way skeleton programs can be run on clusters/networks/grids hosting heterogeneous computing resources (different processors, different operating systems, different memory/disk configurations, etc.);

7. *handle dynamicity*, i.e. implement in the skeleton support mechanisms and policies suitable to handle typical dynamic situations, such as those arising when non-dedicated processing elements are used (e.g. peaks of load that impair load balancing strategies) or from sudden unavailability of processing elements (e.g. network faults, node reboot).

The first additional point (5) is crucial to avoid the necessity to rewrite from scratch already existing and possibly highly optimized code. Skeletal systems must allow programmers to reuse existing portions of sequential, optimized code by just wrapping them into suitable skeleton settings. The wrapping process must be kept as automatic as possible and automatic wrapping of object code is to be considered as well, provided the object code satisfies some minimal requirements (e.g. it is a function with a known signature and semantics). Furthermore, code reuse should support a variety of programming languages. In particular, it should not be limited to the ability of reuse code written in the implementation language of the skeletal system, to allow programmers to get and keep the better sequential software at hand, independently of its implementation language. The second (6) and third (7) additional points actually come from experiences in grid programming systems. Grid systems are *heterogeneous* and *dynamic* by definition [15], and any programming environment targeting grid architectures must include proper techniques and algorithms to take care of these two important aspects, possibly in an automatic and transparent way [16]. In particular, the design of skeletal systems should allow programs to run seamlessly on distributed architectures composed of different processing elements, both in terms of architecture and of operating system. Also, they should cope with the varying features of the distributed target architecture over time, such as varying loads on the processing elements, shared interconnection networks bandwidth and latency, possible faults or unavailability of network links and processing elements.

Observe that heterogeneity property may also hold in plain cluster architectures. Clusters upgrades and repairs can rarely rely on the very same machines used for the original cluster set up, due to the constant and impressively fast technology improvement. Therefore working cluster machines end up with hosting different processing nodes, usually. Dynamicity is not strictly related to grid architectures as well. All non-dedicated cluster architectures experiment variable loads on their processing and network elements, and therefore clever dynamic load balancing policies are needed, at least. Despite suffering minor problems with respect to grids due to resource management, non-dedicated workstation networks and clusters need proper tools to cope with dynamicity. As a consequence, requirements (6) and (7) can be considered central to most of the current parallel architectures.

Overall, the seven principles stated above should be considered the basis of "next generation" skeleton systems. In other words, mature skeleton technology should efficiently address all the related problems. In this perspective, here we want to discuss two "second-generation" experiences of our group in Pisa, namely the `muskel` [17] and the ASSIST one [18–20]. Both are programming environments based on the skeleton structured parallel programming concepts. The former is a plain Java library exploiting macro data flow implementation techniques [10] directly derived from Lithium [21]. The latter defines a new programming language, which is actually a coordination language that uses skeletons to model parallelism exploitation patterns. ASSIST exploits implementation techniques derived from the implementation template methodology [22] developed in P3L [5] and adopted by other skeleton frameworks [7].

`muskel` and ASSIST, besides addressing in different ways the principles (1)–(7), share another important common feature: differently from other skeleton systems, they do not just use the structure of skeletons in the program to build the network of processes that implements the parallel program. Actually, they synthesize the high level semantic information of the skeletons and their structure into an intermediate meta description. This description is then used to optimize several aspects of skeletons implementation, such as the network of processes topology, the selection of data that should be marshaled, the selection of data should be consolidated/migrated in case of run-time reconfiguration, the processes involved in the reconfiguration. However, the techniques used to perform these optimization are different for `muskel` and ASSIST. `muskel` implements skeletons exploiting a distributed macro data flow interpreter after transforming skeleton programs into data flow graphs. The optimizations are performed at level of data flow graph [23] and its distributed interpretation [10]. ASSIST exploits high level skeletal semantic to support a partial evaluation strategy of the source program. The compiler generates a network of virtual activities, which are mapped onto the suitable implementation device of the target platform (processes, threads, communication and sharing primitives). This mapping is partially done a compile time, then completed at lunch time, and possibly changed at run-time.

As a matter of fact, this is also a symptom of an evolution in the skeleton programming systems. More and more attention is paid to transfer and exploit meta information deriving from skeleton program structuring to the efficient implementation of the *functionality* of the original skeleton program rather than to the direct matching of skeleton program structure with the template of the process (parallel activity) network used to implement the program itself.

3. muskel

muskel [17,45] is a full Java skeleton library providing user with usual stream parallel skeletons: pipelines, farms as well as arbitrary composition of farm and pipes. It is a compact, optimized subset of Lithium [21], mainly meant as a handy test bed to experiment new implementation strategies. muskel key features are the usage of macro data flow implementation techniques and the normal form concept, derived from Lithium, the application manager and the performance contract concepts, original of muskel and subsequently moved to ASSIST. A typical muskel program looks like the following:

```
import muskel.*;
  public static void main(String [] args) {
  Compute stage1 = new Farm(new doSweep());              comp. doSweep on all tasks
  Compute stage2 = new postProcRes();                    then post process results
  Compute mainProgram = new Pipe(stage1,stage2);                           in pipe
  ParDegree parDegree = new parDegree(5);               ask this performance contract
  ApplicationManager manager =                                   create the manager
    new ApplicationManager(mainProgram);                        to run mainProgram
  manager.setContract(parDegree);                           give contract to manager
  manager.setInputStream("input.dat");                        set input data location
  manager.setOutputStream("output.dat");                          set output location
  manager.eval();                                           run the parallel program
}
```

This parallel program performs parameter sweeping (as coded in the doSweep sequential Java code) on a set of input parameters stored in the input.dat file and produces the results in the output.dat file after post-processing each result of doSweep via postProcRes. The user asks for a performance contract stating that the parallelism degree in the execution of the program should be 5. This code is the full code the programmer must write to get the working muskel parallel program.

Compilation of skeleton programs to macro data flow. Skeletons are implemented in muskel using macro data flow technology [10,21]: the skeleton program is translated into a data flow instruction graph. Instructions are fired (i.e. scheduled for the execution) when all their input tokens are available. A fireable instruction is simply scheduled for the execution on one of the available remote interpreters using RMI. Data flow instructions are actually "macro" data flow instructions. The user provides instruction functions as parameters of the associate skeleton, as shown in the code above. In particular, sequential code to be used in these parameters is supplied as a Compute object, i.e. an object with an Object compute(Object task) method that returns an Object embedded token result after computing some sequential function on the input token task object(s).

The muskel parallel skeleton code structure is captured by the grammar $\mathsf{P} ::= \mathsf{seq}(\mathit{className})|\mathsf{pipe}(\mathsf{P}, \mathsf{P})|\mathsf{farm}(\mathsf{P})$ where the classNames refer to classes implementing the Compute interface. A macro data flow instruction can be described by the tuple: $\langle id, gid, opcode, I^n, O^k \rangle$ where *id* is the instruction identifier, *gid* is the graph identifier (both are either integers or the special *null* identifier), *opcode* is the name of the Compute class providing the code to compute the instruction (i.e. computing the output tokens out of the input ones) and I and O are the input tokens and the output token destinations, respectively (n is the input arity of the opcode function and k its output arity). An input token is a pair $\langle value, presenceBit \rangle$ and an output token destination is a pair $\langle destInstructionId, destTokenIndex\rangle$. The process compiling the skeleton program into the data flow graph can be described as follows. We define a pre-compile function $PC{:}\,\mathsf{P} \times GraphIds \rightarrow MDFgraph$ as

$$PC[\mathsf{seq}(\mathtt{f})]_{gid} = \lambda i.\{\langle newId(), gid, \mathtt{f}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle i, 0\rangle\rangle\rangle\}$$
$$PC[\mathsf{farm}(\mathsf{P})]_{gid} = C[\mathsf{P}]_{gid}$$
$$PC[\mathsf{pipe}(\mathsf{P}_1, \mathsf{P}_2]_{gid} = \lambda i.\{C[\mathsf{P}_1]_{gid}(getId(P')), P'\} \quad \text{where } P' = C[\mathsf{P}_2]_{gid}(i)$$

where $getID()$ is the function returning the id of the first instruction in its argument graph, that is, the one assuming to receive the input token from outside the graph. The compile function $C[\,]{:}\,\mathsf{P} \rightarrow MDFgraph$ is consequently defined as

$$C[\mathsf{P}] = PC[\mathsf{P}]_{newGid()}(\mathtt{null})$$

where $newId()$ and $newGid()$ are stateful functions returning a fresh (i.e. unused) instruction and graph identifier, respectively. The compile function returns therefore a graph, with a fresh graph identifier, hosting all the data flow instructions relative to the skeleton program. The result tokens are identified as those whose destination is `null`. As an example, the compilation of the `main` program $\mathsf{pipe}(\mathsf{farm}(\mathsf{seq}(\mathtt{f})), \mathsf{farm}(\mathsf{seq}(\mathtt{g})))$ produces the data flow graph (assuming that identifiers and token positions start from 1):

$$\{\langle 1, 1, \mathtt{f}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle 2, 1\rangle\rangle\rangle, \langle 2, 1, \mathtt{g}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle \mathtt{null}, \mathtt{null}\rangle\rangle\rangle\}$$

When the application manager is told to actually compute the program, via an `eval( )` method call, the input file stream is read looking for tasks to be computed. Each task found is used to fill the input token in the $getId(C[\mathsf{P}])$ data flow instruction in a new $C[\mathsf{P}]$ graph. In the example above, this results in the generation of a set of independent graphs such as

$$\{\langle 1, i, \mathtt{f}, \langle\langle \mathtt{task}_{\,i}, \mathtt{true}\rangle\rangle, \langle\langle 2, 1\rangle\rangle\rangle \langle 2, i, \mathtt{g}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle \mathtt{null}, \mathtt{null}\rangle\rangle\rangle\}$$

for all the tasks $task_i$ ranging from $task_1$ to $task_n$.

All the resulting instructions are put in the distributed interpreter *task pool* in such a way that the control threads taking care of feeding the remote data flow interpreter instances can start fetching the fireable instructions. The output tokens generated by instructions with destination tag equal to `null` are directly delivered to the output file stream by the threads receiving them from the remote interpreter instances. Those with a non-`null` flag are delivered to the proper instructions in the task pool that will eventually become fireable.

Normal form. In `muskel`, the skeleton program executed is not actually the one provided by the user: the skeleton program is first transformed to obtain its normal form as defined in [23] and then this normal form skeleton program is actually compiled to macro data flow and eventually executed. The normal form $\overline{\mathsf{P}}$ of a `muskel` program P is defined as

$$\overline{\mathsf{P}} = \mathsf{farm}(\sigma(\mathsf{P}))$$

where the $\sigma{:}\,\mathsf{P} \rightarrow \mathsf{P}$ recursive auxiliary function is defined as follows:

$$\sigma(\mathsf{seq}(\mathsf{P})) = \mathsf{seq}(\mathsf{P}), \quad \sigma(\mathsf{farm}(\mathsf{P})) = \sigma(\mathsf{P}), \quad \sigma(\mathsf{pipe}(\mathsf{P}', \mathsf{P}'')) = \sigma(\mathsf{P}'); \sigma(\mathsf{P}'')$$

and the ";" operator represents sequential composition.

As an example, the `muskel` skeleton code at the beginning of this section is transformed before compiling it into macro data flow into the following equivalent, normal form `muskel` source code:

```
mainProgram = new Farm(new SeqComp(new doSweep(), new PostProcRes()));
```

The normal form has been proved to implement programs that are functionally equivalent and with better performance with respect to the original, user supplied, non-normal form programs [23,21]. The transformation process is completely transparent to the programmer.

Distributed macro data flow execution. The normalized skeleton code is eventually transformed into a macro data flow instruction graph and executed as any other `muskel` skeleton program. An application manager is started (see next paragraph) that recruits a convenient number of remote data flow interpreters. The fireable macro data flow instructions deriving from the skeleton code are staged for the execution to the remote interpreters. A thread is forked for each one of the remote interpreters recruited. The thread looks for a fireable instruction in a task pool repository, delivers it to the associate remote interpreter and waits for the results of the computation. When results come back from the remote interpreter they are either delivered to the output stream, or reinserted in the proper target macro data flow instruction in the task pool. This macro data flow instruction can possibly become fireable. Immediately after the thread starts trying to fetch a new fireable instruction.

Application manager. The muskel application manager takes care of several aspects related to program execution. It recruits a number of remote interpreter nodes suitable to accomplish the performance contract asked to the user, exploiting a proper discovery algorithm. The remote interpreter instances should have been started once and for all on the nodes of the target architecture as Java RMI (plain or Activatable) objects. The recruiting process is best effort, that is the manager tries to recruit the closest number of remote interpreters to the one stated in the performance contract. Then, the application manager monitors program execution and takes care of constantly ensuring the performance contract. In case a remote interpreter becomes unavailable (e.g. due to a network or to a node failure) the muskel application manager arranges to recruit a new one, if available. The task(s) left un-computed by the missing interpreter(s) are put back to the task pool and they will be eventually re-scheduled to a different interpreter. Results achieved with muskel are very good both in terms of load balancing and in terms of fault tolerance and absolute performance/efficiency [17].

Unstructured parallelism. Unstructured parallelism exploitation mechanisms are also provided to the muskel user by allowing him to write his own macro data flow graphs modeling those parallelism exploitation patterns not captured by the skeletons provided natively by muskel. In particular, muskel users may program their own macro data flow graphs and embed them into ParCompute objects. Those objects can be used in any place a Compute object is used (actually ParCompute is a subclass of Compute). ParCompute objects have a MdfGraph getMdfGraph( ) method that can be used in the skeleton to macro data flow graph compiler to retrieve the user defined macro data flow graph supplied as a parameter in the ParCompute constructor. Therefore, macro data flow graphs happen to be build either compiling the Compute objects as shown above, or by gluing together the macro data flow graphs supplied by users and embedded in the ParCompute objects. The only limitation, at the moment, is that the user supplied macro data flow graphs in the ParCompute objects must have a single input and a single output, i.e. they must compute with a single input token and the result should be a single output token. This is to be able to merge ParCompute macro data flow graphs and graphs generated compiling Compute Pipelines and Farms [45].

The mechanism just outlined can be used to program all those parts of the application that cannot be easily and efficiently implementing using the skeletons subsystem. Being the mechanisms used to execute user supplied macro data flow graphs (instructions) the same as the ones used to execute skeleton programs, the same level of efficiency is granted in the execution to user defined macro data flow code than the one achieved in the execution of macro data flow code compiled from muskel skeletons. Furthermore, users can program once and for all particular macro data flow graphs in a ParCompute subclass and then use this class as a normal muskel skeleton class, thus completely achieving expandability of the muskel skeleton set.

3.1. Assessment

muskel has been used to program several parallel applications, including number crunching, image processing and sequence matching. In case the computational grain (i.e. the ratio between the time spent computing an instruction and the time spent delivering the remote interpreter the input tokens and retrieving from the remote interpreter the result tokens) of the macro data flow instructions is not too small, almost perfect scalability is achieved, as show in Fig. 1. The plots are relative to an experiment performed on a network of Linux/Pentium III blades with Fast Ethernet interconnection. Medium grain is about 10, large grain is about 160, that is the time spent computing a macro data flow instruction is 160 times the time spend in delivering tokens to the interpreter processing it and retrieving the result tokens out of the remote interpreter. Although this can look like to be a large value, the normal form transformation applied in muskel behaves such that the actual grain values measured in real application code are even larger.

A minimal effort is required to experienced Java programmers to use muskel: basically, a small effort to implement the Compute interface in the existing application dependent code, plus the launch of the remote

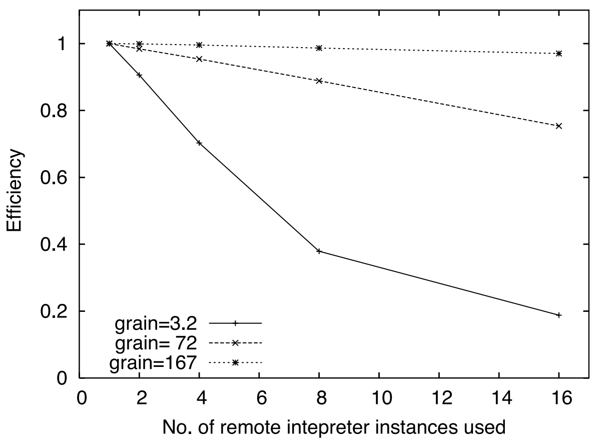

Fig. 1. Efficiency of `muskel` programs, with respect to macro data flow instructions grain.

interpreter RMI objects on the available processing elements (both plain RMI and `rmid` versions of the remote interpreter are available). This addresses principle 1. `FileInputStream` and `FileOutputStream` objects can be passed to the manager to provide input task and retrieve output results. Therefore specialized parallel patterns can be programmed that interact with the existing skeleton (programs) via the streams, in the flavor of what happened in [12] with Unix file handles. Furthermore, with the explicit macro data flow mechanisms just described, the programmer can even program his own macro data flow graphs and embed such graphs in skeleton program in any place where a plain `Compute` object may appear (e.g. as a pipeline stage or as a farm worker). Overall, these two aspects allow both to integrate ad-hoc parallelism and to accommodate diversity, thus addressing (2) and (3). The payback offered by `muskel` is evidenced by the `muskel` code shown in this section: a negligible amount of code is needed to get a fully working, efficient parallel application (3). Target architecture heterogeneity is handled naturally in `muskel` due to portability features of the JVM and RMI (6). We performed several experiments, using a mix of Mac OS/X PowerPC machines and different configurations of Pentium machine with Linux. The experiments run successfully, as expected, that is they computed the correct results despite the fact different macro data flow instructions of the same program were run on machines with different processors and operating systems. Obviously, the runs used the machines differently, that is they scheduled a different number of fireable instructions on them. As an example, we used three machines (a dual Pentium IV and a Pentium III with Linux and a PPC G5 with Mac OS/X Tiger) to run sample muskel code and we end up measuring that 48% of the macro data flow instructions were executed by the Pentium III machine, that was idle at that time, 24% of the instructions were run on the dual Pentium IV machine, that is the group server and at that time it experimented a load equal to 2.15 (uptime measure), and 28% of the instructions on the G5 PPC, that was also quite loaded at that time. Dynamicity is handled in the `ApplicationManager`, where fault tolerance is also dealt with (7). Experimental data show that a negligible overhead is charged to replace a faulty remote interpreter node, provided that other, currently idle remote interpreters can be recruited [17]. Typical results of experiments on dynamicity are show in Table 1.

Table 1
Percentage of MDF instructions computed by remote interpreters instances: during time slices 5–12 an external additional load was executed on WS2, thus some of the work of WS2 has been redistributed to WS1 and WS3

	Time slices						
	0–2 s	3–5 s	6–8 s	9–11 s	12–14 s	15–17 s	18–20 s
WS1 (%)	2.2	1.1	2.2	2.2	1.1	2.2	1.1
WS2 (%)	6.5	5.4	**3.3**	**3.3**	**4.3**	5.4	5.4
WS3 (%)	8.7	7.6	7.6	8.7	7.6	6.5	7.6
Total (%)	17.4	14.1	13.0	14.1	13.0	14.1	14.1

In this case, we run an application on three (heterogeneous) machines. The table has in columns the percentage of the overall macro data flow instruction executed on the machine in that time slice (each column accounts for a time slice of 3 s, thus the whole run was about 21 s). We externally overloaded the second machine 5 s after the application start (second column). The reader can easily verify how this lead to a redistribution of the fireable instructions onto the other two machines. The additional load was killed after 14 s (fifth column). Notice that this instruction re-distribution takes place due to the particular structure of the distributed macro data flow interpreter (the overloaded machines simply succeeds asking less new fireable instructions to execute, due to the time spent for execution of the single macro data flow instruction) and does not require, in particular, any kind of specific, on line action possibly disturbing the execution of the main program. In ASSIST, as we shall see in Section 4, part of the reconfiguration is actually computed off line, but eventually, the main computation as to be interrupted to allow the re-distribution of parallel activities.

Concerning code reuse (5), existing Java code can be easily reused by wrapping existing code in a `Compute` interface. Code written in other languages, as well as object code, requires more effort to be integrated in `muskel` programs, however. The structure used to exploit parallelism heavily relies on serializability of code and it will be difficult to adapt it to support C, FORTRAN or even C++ code reuse. While `muskel` currently requires full RMI access to all the nodes hosting a remote MDF interpreter instance, a version of `muskel` built on top of the ProActive grid middleware [24] is currently being fine tuned that allows to target actual grid architectures supporting `ssh` access to nodes only.

4. ASSIST

ASSIST (A Software development System based on Integrated Skeleton Technology [20]) is a programming environment aimed at supporting parallel and distributed application development on clusters and networks of workstations as well as on grids. ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules, interconnected by typed streams of data. Each stream realizes a one-way asynchronous channel between two sets of endpoint modules: sources and sinks. Data items injected from sources are broadcast to all sinks. A simple ASSIST program for parallel matrix multiplication is shown in Fig. 2: the `matrix_mul` module takes one matrix from each of the two modules `send1`, and `send2`; multiply them and gives the result to `recv` module.

Modules can be either sequential or parallel. A sequential module *wraps* a sequential function. A parallel module *(parmod)* can be used to describe the parallel execution of a number of sequential activities that are activated and run as *Virtual Processes* (VPs) on items coming from the input streams. Each VP is arranged in a *topology* and accordingly named (e.g. VP[*i*,*j*]). A VP is programmed as the sequence of calls to procedures (*procs*) that wrap standard sequential code (C, C++, Fortran). The *topology* declaration specializes the behavior of the VPs as farm (topology none) or data parallel (topology array). Farm parmods exhibit a logically unbounded number of anonymous VPs, which never synchronize one each other. Data parallel parmods exhibit a programmer-defined number of VPs, which can be named by topology indexes, can share data, and might synchronize in a barrier. Data can be shared via global variables *(attributes)*, which can be read by all VPs, and be written by the owner VP. The ownership of global variables or part of them can be declared by using topology indexes (as an example the ownership of a matrix can be assigned by row, columns and blocks). Barriers are part of the ASSIST API; they might appear between procedure calls. They also behave as memory fences. A sequence of procedure calls in a VP might be surrounded by an iterative command (e.g. for, while), which termination clause

```
generic main() {
  stream long[N][N] s1, s2, s3;
  send1  (output_stream  s1);
  send2  (output_stream  s2);
  matrix_mul (input_stream  s1, s2,
              output_stream  s3);
  recv   (input_stream s3); }
```

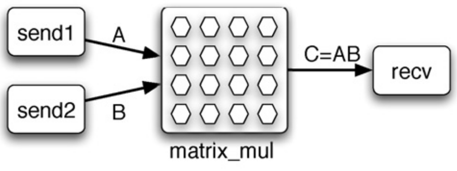

Fig. 2. Multiplication of two streams of matrixes in ASSIST, and the relative graph of modules.

may involve values reduced from VPs local variables. Despite these basic features, ASSIST has several extensions provided to the programmer by means of libraries: among the others, arbitrary data sharing via Distributed Shared Memory, synchronization among subset of VPs, access to a parallel file system. Observe however, that because of the extreme freedom these libraries give to the programmer, their use may prevent or have a negative impact on the performance of other important features of ASSIST, such as transparency in mapping and deployment, program adaptability and resilience to faults (currently under investigation).

A module can non-deterministically accept from one or more input streams a number of input items, which may be decomposed in parts that are (logically) distributed to VPs according to rules specified in *input section* of the parmod and to VPs topology. These rules include broadcast, scatter, multicast or a combination of them. Once triggered by data, each VP executes its sequence of procedures and barriers, if any. The *output_section* has a symmetric role of the *input section*, and enables the gathering of data items, or parts of them, from VPs. Parmod basic behavior is sketched in Fig. 3. More details on the ASSIST coordination language can be found in [18,25].

The example parmod in Fig. 4 exhibits a `topology array` (line 3, $N \times N$ VPs behaving in a SPMD fashion). Once the two input matrixes are received (line 8), they are both scattered to the VPs which store them in the distributed shared matrixes `A` and `B` (lines 9–10) that has been previously declared (lines 4–5). Then, all

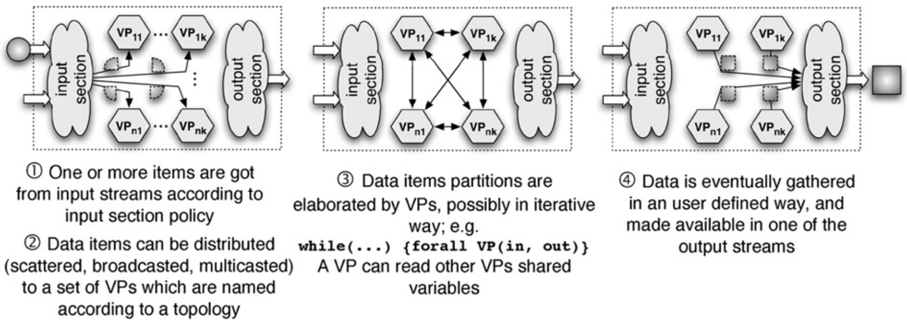
input section
VP11
VP1k
VPn1
VPnk
output section
① One or more items are got from input streams according to input section policy
② Data items can be distributed (scattered, broadcasted, multicasted) to a set of VPs which are named according to a topology
③ Data items partitions are elaborated by VPs, possibly in iterative way; e.g.
while(...) {forall VP(in, out)}
A VP can read other VPs shared variables
④ Data is eventually gathered in an user defined way, and made available in one of the output streams

Fig. 3. Parmod basic behavior.

```
1  parmod matrix_mul (input_stream long M1[N][N], long M2[N][N],
2                     output_stream long M3[N][N]){
3    topology array [i:N][j:N] Pv;
4    attribute long A[N][N] scatter A[*ia][*ja] onto Pv[ia][ja];
5    attribute long B[N][N] scatter B[*ib][*jb] onto Pv[ib][jb];
6    stream long ris;
7    do input_section {
8      guard1: on , , M1 && M2 {
9        distribution M1[*i0][*j0] scatter to A[i0][j0];
10       distribution M2[*i1][*j1] scatter to B[i1][j1];
11     }   } while (true)
12   virtual_processes {
13     elab1 (in guard1 out ris) {
14       VP i, j { f_mul (in A[i][], B[][j] output_stream ris);}}}
15   output_section {
16     collects ris from ALL Pv[i][j] {
17       int elem; int Matrix_ris_[N][N];
18       AST_FOR_EACH(elem) {
19         Matrix_ris_[i][j]=elem;
20       }
21       assist_out(M3, Matrix_ris_);
22     }<>; } }
23 proc f_mul(in long A[N], long B[N] output_stream long Res)
24 $c++{ register long r=0;
25   for (register int k=0; k<N; ++k)
26     r += A[k]*B[k];
27   assist_out(Res,r); }c++$
```

Fig. 4. ASSIST code of the matrix_mul parmod.

elements of the result matrix C are computed in parallel (lines 12–14). Once all VPs completed the operation, a result matrix is collected from the distributed matrix C and sent into the output stream (lines 15–22). A standard row by column multiplication C++ code is wrapped within a `proc` declaration (lines 24–27).

4.1. Assessment

The ASSIST parmod represents the major innovation of ASSIST with respect to previous skeleton systems developed at our group. The parmod represents a *generic* parallel module. By specializing this generic construct many classical parallel skeletons can be derived: farm, deal, map/forall, haloswap, and reduce with or without shared state. Pipelines, and more in general graphs can be easily expressed by wiring modules with streams. ASSIST clearly meets several principles in the set proposed in Section 2. In particular, a graph of parmods represents, by its very nature, the essence of *accommodating diversity* (3), since both the structure of the graph can be freely defined and the parmod can be specialized to model several different classical skeletons and many variants of them. Moreover, the VPs virtualize parallel algorithms with respect to the mapping onto the distributed platform. On the one hand, this enables the delay at launch time of the real mapping of VPs onto processes on platforms, in such a way a good communication/computation ratio for the algorithm/platform pair can be tuned (usually #VPs > #processes > #platforms). On the other hand, it enables to further improve malleability of algorithms by allowing the mapping between VPs and target platforms to change during application run (adaptability). Ad-hoc parallelism (2) can be realized by enriching VPs code with both ASSIST native (e.g. barriers, shared memory accesses) and guest language synchronization or communication primitives towards external processes and services (e.g. Unix sockets, OS or library calls). Moreover, modules can be wired almost automatically to standard services via special streams (e.g. CCM-CORBA/IIOP channels, HTTP/SOAP Web Services), thus bridging ASSIST native code with third-party legacy sequential and parallel code [26–29]. This, together with code wrapping in `procs`, enables *code reuse* (5), both in binary and source form, respectively. Notably, the framework nature of the ASSIST language does not require changing or interleaving sequential code with library calls.

The implementation of parmod is highly optimized and both rely on a huge compilation process and on an optimized run time system, namely the ASSISTlib. The compiler also guarantees the transparent usage of different guest languages (C, C++, Fortran, Java) and the correctness onto heterogeneous platforms (6), among the set of supported ones, i.e. several flavors of Unix and Mac OS/X). In particular, the memory layout of data structures (including alignment and endianess) is managed transparently to the user: the compiler generates highly optimized (template-based) marshalling code for each type and for each pair of platform kind. At the application launch time, each communication primitive is enriched with the correct marshaler if and only if it is required by the kind of involved communication endpoints: a communication between two platforms of the same kind does not need any marshalling [30]. Both the transparency and the efficiency of the communications and the distributed memory sharing between different kind of platforms reachable in this way is one the typical byproducts of skeletons (4): high-level, semantic information about the parallel code and data types is used to generate and tune an efficient marshalling process.

The price to pay is a somehow heavy language, thus not fully compliant with *minimal conceptual disruption* principle (1). Also, the possibility to express arbitrary graphs of parallel modules is a notable step away from previous experiences. As `muskel`, ASSIST supports autonomic control of parallel modules and of the overall application [31]. A parmod can be executed in such a way that the user asks a given performance contract to be satisfied. In this case, the parmod automatically provides to keep the contract satisfied, if possible: exploiting the knowledge coming from the parmod analytic performance models, the parmod dynamically adapts the number of resource used to execute the parmod in order to satisfy the user supplied performance contract (7).

As an example, Fig. 5 reports a run of a data parallel shortest path algorithm implemented with an ASSIST parmod (with 400 VPs). The figure shows the time needed to complete an iteration of the algorithm from the start to the convergence (iterations 0–400), the respective relative unbalance[1] for the iteration, and the load

[1] Defined for each iteration as $(\max T_i - \min T_i)/\max T_i$, where $\max T_i$, $\min T_i$ are the maximum and minimum across of VP_i of the time needed to complete the iteration.

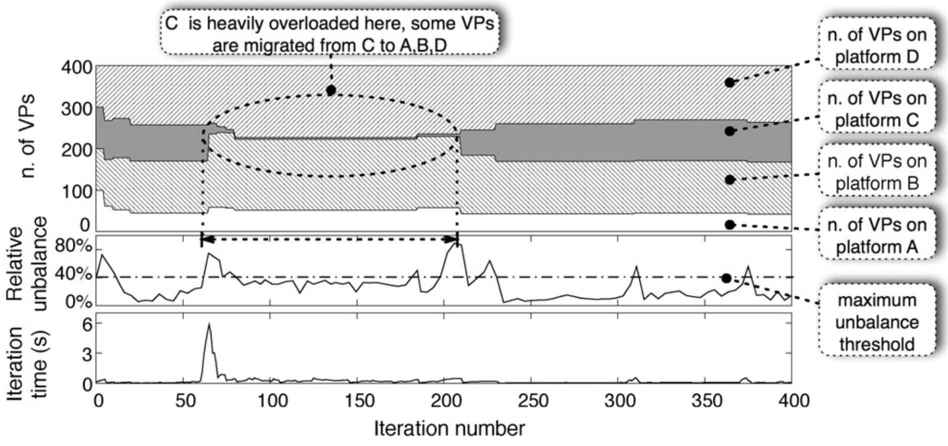

Fig. 5. ASSIST parmod with autonomic control computing a data parallel algorithm (shortest path). The parmod is run on 4 heterogeneous platforms (A, B, C, D) exhibiting different computing power. Platform C happens to be externally overloaded for a while during the run, the parmod reacts redistributing load on other platforms.

distribution across the platforms. As soon as the relative unbalance climbs up a given threshold, the autonomic control rebalances the load by redistributing VPs across the platforms. All the process is completely transparent to the application programmer, who is not charged to write the code to realize adaptivity (such as make decisions, synchronize and migrate activities). The compiler instruments the application code with everything needed to correctly migrate VPs across platforms. Again, this is possible because of the high-level, structured nature of the ASSIST language. The compiler knows in which moments, during the run of the parmod, the distributed state of the VPs is "safe" for a reconfiguration (so-called *reconf-safe* points) because their interaction follows a known paradigm, which is actually its skeleton. This knowledge is also used to optimize reconfiguration latency. As a simple example consider two parmods behaving as data parallel and farm, respectively. In the former case, a reconfiguration requires VPs consensus and data migration; while in the latter case no synchronization is needed since computation on different VPs are independent. Indeed, as shown in Table 2, the reconfiguration overhead (add/remove a number of processing elements (PE) during the run) of a farm parmod is almost zero, while the data parallel parmod exhibits an overhead that depends on the algorithm data size and the number of PEs involved in the reconfiguration [31].

The ASSIST environment is currently released under GPL by the Dept. of Computer Science of the University of Pisa and targets workstation networks, clusters and grids either equipped with plain TCP/IP and `ssh/scp` tools (in this case Linux and Mac OS/X machines are supported as target nodes) or equipped with Globus 2.4. Several real world applications has been developed with ASSIST by partners of the national project GRID.it (including the Italian Space Agency) in these areas: graphics (isosurface detection), bioinformatics (protein folding), massive data-mining, "social" applications including remote sensing and image analysis (sea oil spill detection, landslip detection), chemistry (ab-initio molecule simulation), and numerical computing [32–35].

Table 2
Evaluation of reconfiguration latency (R_l) of an ASSIST parmod (ms) on the cluster described in Section 3

parmod kind	Data-parallel (with shared state)						Farm (without shared state)					
reconf. kind	add_PEs			remove_PEs			add_PEs			remove_PEs		
n. of PEs	$1 \to 2$	$2 \to 4$	$4 \to 8$	$2 \to 1$	$4 \to 2$	$8 \to 4$	$1 \to 2$	$2 \to 4$	$4 \to 8$	$2 \to 1$	$4 \to 2$	$8 \to 4$
R_l barrier	1.2	1.6	2.3	0.8	1.4	3.7	–	–	–	–	–	–
R_l stream-item	4.7	12.0	33.9	3.9	6.5	19.1	~0	~0	~0	~0	~0	~0

On this cluster, 50 ms are needed to ping 200KB between two PEs, or to compute a 1 M integer additions.

5. Related work

Skeleton based programming environments had been very popular in the 1990, right after the Cole' results. Currently, there is quite a limited number of research groups developing/supporting skeleton based programming environments. Beside our group in Pisa, which is active since early 1990, the most notable skeleton based programming systems are eSkel by Cole [14], Muesli by Kuchen [7], and the ones from the group of Gorlatch [36]. Furthermore the group in Alberta University is maintaining the CO_2P_3S programming environment [37] which is actually based on the design pattern technology, but is very close to the skeleton world.

Cole's eSkel system implements skeletons as collective MPI operations. It has been introduced in [14], that is in the same paper where the four principles (1) to (4) were discussed. All these points are addressed in eSkel [14,13,38]. eSkel also provides some code reuse facilities (5) as most C and C++ code can simply be adapted in eSkel programs. In eSkel heterogeneous architectures are supported (6) through the usage of MPI, much in the sense heterogeneous architectures are supported through the usage of Java in `muskel`. However, current implementation of eSkel (2.0) does not support user defined MPI data types in the communication primitives, that actually use `MPI_INT` data buffers, and therefore heterogeneous architectures can be targeted using proper MPI implementations just when all the nodes have the same type of processors. No support for dynamicity handling (7) is provided in eSkel, however.

Muesli is basically a C++ library built on top of MPI. Differently from eSkel, it provides stream parallel skeletons, data parallel objects (arrays, basically) and data parallel operations as C++ template classes. The user interface is definitely very good, as the full power of object oriented paradigm along with templates is exploited to provide Muesli programmers with user-friendly skeletons, and consequently C++ programmers can develop parallel programs very rapidly. In particular, despite exploiting MPI as eSkel, Muesli does not require any MPI specific knowledge/action to write a skeleton program. Therefore point (1) is very well addressed here. Points (2) and (3) are addressed providing the programmer with a full set of (data parallel) operations that can be freely combined. The payback (4) is mainly related to the OO techniques exploited to provide skeletons. Code reuse (5) is supported as it is supported in eSkel, as programmers can use C++/C code to build their own skeletons as well as sequential code to be used in the skeletons. Also in this case there is limited support to heterogeneity (6): the MPI code in the Skeleton library directly uses `MPI_BYTE` buffers to implement Muesli communications, and therefore MPI libraries supporting heterogeneous architectures may be used just in case the nodes sport the same kind of processor and the same C/C++ compiler toolset. Dynamicity handling (7) is not supported at all in Muesli.

Gorlatch's group work was more related on data parallel skeleton optimizations, actually [39,40]. Recently, they presented a grid programming environment HOC [36,41], which provides suitable ways of developing component based grid applications exploiting classical skeleton components. The implementation exploits Web Services technology [42]. Overall the HOC programming environment addressed principles (1) and (4). Points (2) and (3) rely on the possibility given to programmers to insert/create new HOCs in the repository. Point (6) is handled via Web Services. This technology is inherently multiplatform, and therefore heterogeneous target architectures can be easily used to run HOC programs. Point (5) is guaranteed as sequential code can easily (modulus the fact some XML code is needed, actually) be wrapped in Web Services. However, no support to (7) is included in the current HOC version.

CO_2P_3S is a design pattern based parallel programming environment, entirely developed in Java. It has been the first structured programming environment supporting controlled user additions to the pattern (skeleton) library [43], therefore fully addressing (2) and (3) requirements. Requirement (1) is addressed, being a plain Java library and (4) is clearly shown by the time spent in developing structured programs using CO_2P_3S [44]. Point (6) is addressed through Java portability, as in `muskel`, but the CO_2P_3S implementation only targets SMPS. Point (5) is also taken into account, as Java code can be seamlessly reused in CO_2P_3S programs (only Java code, actually), and point (7) is not addressed at all.

6. Conclusions

We proposed an extension of the set of principles to be satisfied when developing efficient and hopefully successful skeletal programming systems presented in [14] by Cole. Actually, we proposed three new, additional principles, sharing the same level of abstraction of the original ones. We believe they are fundamental to the success of skeletal systems as they guarantee the possibility to preserve the investments made in sequential software development and to target a larger and more significant class of architectures. Then we discussed the way these requirements are addressed in two skeleton programming environments currently being developed in Pisa, `muskel` and ASSIST. These environments are shortly outlined before discussing how they satisfy the original and the new requirements. Both `muskel` and ASSIST accomplish to satisfy the seven requirements, possibly using different strategies and using for sure very different implementations. The `muskel` and ASSIST experience demonstrated that the ability of adapting application execution to varying features of heterogeneous target architecture is a key point in convincing a user to migrate to a skeleton programming environment.

Eventually, we outlined how some of the more significant skeleton based parallel programming system currently available deal with the seven requirements. Most of these environments address fairly well some of the original Cole principles although they did not take into suitable account the three further principles taken into account in this work.

References

[1] M. Cole, A skeletal approach to exploitation of parallelism, in: Proc. of CONPAR 88, British Computer Society Workshop Series, Cambridge University Press, 1989, pp. 667–675.

[2] J. Darlington, A.J. Field, P. Harrison, P.H.J. Kelly, D.W.N. Sharp, R.L. While, Q. Wu, Parallel programming using skeleton functions, in: Proc. of Parallel Architectures and Languages Europe (PARLE'93), LNCS, vol. 694, Springer, Munich, Germany, 1993, pp. 146–160.

[3] J. Darlington, Y. Guo, H.W. To, Q. Wu, J. Yang, M. Kohler, Fortran-S: a uniform functional interface to parallel imperative languages, in: Proc. of 3rd Parallel Computing Workshop (PCW'94), Fujitsu Laboratories Ltd., 1994.

[4] M. Danelutto, R.D. Meglio, S. Orlando, S. Pelagatti, M. Vanneschi, A methodology for the development and support of massively parallel programs, Future Generation Computer Systems 8 (1–3) (1992) 205–220.

[5] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi, P^3L: a structured high level programming language and its structured support, Concurrency: Practice and Experience 7 (3) (1995) 225–255.

[6] G.H. Botorog, H. Kuchen, Efficient high-level parallel programming, Theoretical Computer Science 196 (1–2) (1998) 71–107.

[7] H. Kuchen, A skeleton library, in: Proc. of 8th Int. Euro-Par 2002 Parallel Processing, LNCS, vol. 2400, Springer, Paderborn, Germany, 2002, pp. 620–629.

[8] J. Sérot, Tagged-token data-flow for skeletons, Parallel Processing Letters 11 (4) (2001) 377–392.

[9] J. Sérot, D. Ginhac, Skeletons for parallel image processing: an overview of the SKiPPER project, Parallel Computing 28 (12) (2002) 1785–1808.

[10] M. Danelutto, Dynamic run time support for skeletons, in: Proc. of Int. PARCO 99: Parallel Computing, Parallel Computing Fundamentals & Applications, Imperial College Press, 1999, pp. 460–467.

[11] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computations, Research Monographs in Parallel and Distributed Computing, Pitman, 1989.

[12] M. Danelutto, M. Stigliani, SKElib: parallel programming with skeletons in C, in: Proc. of 6th Int. Euro-Par 2000 Parallel Processing, LNCS, vol. 1900, Springer, Munich, Germany, 2000, pp. 1175–1184.

[13] A. Benoit, M. Cole, S. Gilmore, J. Hillston, Flexible skeletal programming with eSkel, in: Proc. of 11th Int. Euro-Par 2005 Parallel Processing, LNCS, vol. 3648, Springer, Lisboa, Portugal, 2005, pp. 761–770.

[14] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (3) (2004) 389–406.

[15] I. Foster, C. Kesselmann (Eds.), The Grid 2: Blueprint for a New Computing Infrastructure, Morgan Kaufman, 2003.

[16] Next Generation GRIDs Expert Group, NGG3, Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Vision and Research Directions 2010 and Beyond (January 2006). Available from: <ftp://ftp.cordis.lu/pub/ist/docs/grids/ngg3_eg_final.pdf>.

[17] M. Danelutto, QoS in parallel programming through application managers, in: Proc. of Int. Euromicro PDP: Parallel Distributed and Network-based Processing, IEEE, Lugano, Switzerland, 2005, pp. 282–289.

[18] M. Vanneschi, The programming model of ASSIST, an environment for parallel and distributed portable applications, Parallel Computing 28 (12) (2002) 1709–1732.

[19] M. Aldinucci, S. Campa, P.P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, C. Zoccolo, The implementation of ASSIST, an environment for parallel and distributed programming, in: Proc. of 9th Int. Euro-Par 2003 Parallel Processing, LNCS, vol. 2790, Springer, Klagenfurt, Austria, 2003, pp. 712–721.

[20] The ASSIST home page. Available from: <http://www.di.unipi.it/Assist.html>.

[21] M. Aldinucci, M. Danelutto, P. Teti, An advanced environment supporting structured parallel programming in Java, Future Generation Computer Systems 19 (5) (2003) 611–626.

[22] S. Pelagatti, Structured Development of Parallel Programs, Taylor & Francis, 1998.

[23] M. Aldinucci, M. Danelutto, Stream parallel skeleton optimization, in: Proc. of PDCS: Int. Conference on Parallel and Distributed Computing and Systems, IASTED, ACTA Press, Cambridge, MA, USA, 1999, pp. 955–962.
[24] ProActive home page (2006). Available from: <http://www-sop.inria.fr/oasis/proactive/>.
[25] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, C. Zoccolo, ASSIST as a research framework for high-performance grid programming environments, in: Grid Computing: Software Environments and Tools, Springer, 2006, pp. 230–256, Ch. 10.
[26] M. Aldinucci, M. Danelutto, A. Paternesi, R. Ravazzolo, M. Vanneschi, Building interoperable grid-aware ASSIST applications via WebServices, in: Proc. of Int. PARCO 2005: Parallel Computing, Malaga, Spain, 2005.
[27] S. Magini, P. Pesciullesi, C. Zoccolo, Parallel software interoperability by means of CORBA in the ASSIST programming environment, in: Proc. of 9th Int. Euro-Par 2003 Parallel Processing, LNCS, vol. 2790, Springer, Klagenfurt, Austria, 2003, pp. 679–688.
[28] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, C. Zoccolo, Components for high performance grid programming in GRID.it, in: Component Models and Systems for Grid Applications, CoreGRID series, Springer, 2005, pp. 19–38.
[29] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, C. Zoccolo, Structured implementation of component based grid programming environments, in: Future Generation Grids, CoreGRID Series, Springer, 2005, pp. 217–239.
[30] M. Aldinucci, S. Campa, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, C. Zoccolo, Targeting heterogeneous architectures in ASSIST: experimental results, in: Proc. of 10th Int. Euro-Par 2004 Parallel Processing, LNCS, vol. 3149, Springer, 2004, pp. 638–643.
[31] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, C. Zoccolo, Dynamic reconfiguration of grid-aware applications in ASSIST, in: Proc. of 11th Int. Euro-Par 2005 Parallel Processing, LNCS, vol. 3648, Springer, 2005, pp. 771–781.
[32] S. Crocchianti, A. Laganà, L. Pacifici, V. Piermarini, Parallel skeletons and computational grain in quantum reactive scattering calculations, in: Parallel Computing: Advances and Current Issues. Proc. of the Int. Conference ParCo2001, Imperial College Press, 2002, pp. 91–100.
[33] G. Sardisco, A. Machì, Development of parallel paradigms templates for semi-automatic digital film restoration algorithms, in: Parallel Computing: Advances and Current Issues. Proc. of the Int. Conference ParCo2001, Imperial College Press, 2002, pp. 498–509.
[34] A. Giancaspro, L. Candela, E. Lopinto, V.A. Lorè, G. Milillo, SAR images co-registration parallel implementation, in: Proc. of the Int. Geoscience and Remote Sensing Symposium (IGARSS '02), vol. 3, IEEE, 2002, pp. 1337–1339.
[35] P. D'Ambra, M. Danelutto, D. di Serafino, M. Lapegna, Integrating MPI-based numerical software into an advanced parallel computing environment, in: Proc. of Int. Euromicro PDP: Parallel Distributed and network-based Processing, IEEE, Genova, Italy, 2003, pp. 283–291.
[36] M. Alt, J. Dünnweber, J. Müller, S. Gorlatch, HOCs: higher-order components for grids, in: Component Models and Systems for Grid Applications, CoreGRID Series, Springer, 2005, pp. 157–166.
[37] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, K. Tan, From patterns to frameworks to parallel programs, Parallel Computing 28 (12) (2002) 1663–1683.
[38] M. Cole, A. Benoit, The eSkel home page. Available from: <http://homepages.inf.ed.ac.uk/abenoit1/eSkel/>.
[39] J. Fischer, S. Gorlatch, H. Bischof, Foundations of data-parallel skeletons, in: Patterns and Skeletons for Parallel and Distributed Computing, Springer, 2003, pp. 1–27.
[40] H. Bischof, S. Gorlatch, E. Kitzelmann, Cost optimality and predictability of parallel programming with skeletons, Parallel Processing Letters 13 (4) (2003) 575–587.
[41] J. Dünnweber, S. Gorlatch, HOC-SA: a grid service architecture for higher-order components, in: IEEE International Conference on Services Computing, Shanghai, China, IEEE Computer Society Press, 2004, pp. 288–294.
[42] W3C, Web services home page (2005). Available from: <http://www.w3.org/2002/ws/.
[43] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, S. MacDonald, Using generative design patterns to generate parallel code for a distributed memory environment, in: Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2003, pp. 203–215.
[44] D. Szafron, J. Schaeffer, An experiment to measure the usability of parallel programming systems, Concurrency: Practice and Experience 8 (2) (1996) 147–166.
[45] M. Danelutto, P. Dazzi, Joint structured/unstructured parallelism exploitation in MUSKEL, in: Proc. of the ICCS, 2006, Part II, LNCS, vol. 3992, Springer, 2006, pp. 937–944.