Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 325–341. http://www.scpe.org
ISSN 1895-1767

MUSKEL: A SKELETON LIBRARY SUPPORTING SKELETON SET EXPANDABILITY*

MARCO ALDINUCCI† AND MARCO DANELUTTO† AND PATRIZIO DAZZI‡

Abstract. Programming models based on algorithmic skeletons promise to raise the level of abstraction perceived by programmers when implementing parallel applications, while guaranteeing good performance figures. At the same time, however, they restrict the freedom of programmers to implement arbitrary parallelism exploitation patterns. In fact, efficiency is achieved by restricting the parallelism exploitation patterns provided to the programmer to the useful ones for which efficient implementations, as well as useful and efficient compositions, are known. In this work we introduce `muskel`, a full Java library targeting workstation clusters, networks and grids and providing the programmers with a skeleton based parallel programming environment. `muskel` is implemented exploiting (macro) data flow technology, rather than the more usual skeleton technology relying on the use of implementation templates. Using data flow, `muskel` easily and efficiently implements both classical, predefined skeletons, and user-defined parallelism exploitation patterns. This provides a means to overcome some of the problems that Cole identified in his skeleton "manifesto" as the issues impairing skeleton success in the parallel programming arena. We discuss fully how user-defined skeletons are supported by exploiting a data flow implementation, experimental results and we also discuss extensions supporting the further characterization of skeletons with non-functional properties, such as security, through the use of Aspect Oriented Programming and annotations.

Key words. algoritmical skeletons, data flow, structured parallel programming, distributed computing, security.

1. Introduction. Structured parallel programming models provide the user (programmer) with native high-level parallelism exploitation patterns that can be instantiated, possibly in a nested way, to implement a wide range of applications [13, 23, 24, 8, 6]. In particular, such programming models do not allow programmers to program parallel applications at the "assembly level", i. e. by directly interacting with the distributed execution environment via communication or shared memory access primitives and/or via explicit scheduling and code mapping. Rather, the high-level native, parametric parallelism exploitation patterns provided encapsulate and abstract from these parallelism exploitation related details. For example, to implement an embarrassingly parallel application processing all the data items in an input stream or file, the programmer simply instantiates a "task farm" skeleton by providing the code necessary to process (sequentially) each input task item. The system, either a compiler and run time tool based implementation or a library based one, takes care of devising the appropriate distributed resources to be used, to schedule tasks on the resources and to distribute input tasks and gather output results according to the process mapping used. By contrast, when using a traditional system, the programmer has usually to explicitly program code for distributing and scheduling the processes on the available resources and for moving input and output data between the processing elements involved. The cost of this appealingly high-level way of dealing with parallel programs is paid in terms of programming freedom. The programmer is normally not allowed to use arbitrary parallelism exploitation patterns, but he must use only the ones provided by the system, usually including all those reusable patterns that happen to have efficient distributed implementations available. This is aimed mainly at avoiding the possibility for the programmer to write code that could potentially impair the efficiency of the implementation provided for the available, native parallel patterns. This is a well-known problem. Cole recognized its importance in his "manifesto" paper [13].

In this work we discuss a methodology that can be used to provide parallel application programmers with both the possibility of using predefined skeletons in the usual way and, at the same time, the possibility of implementing their own, additional skeletons, where the predefined ones do not suffice. The proposed methodology, which is based on data flow, preserves most of the benefits typical of structured parallel programming models. According to the proposed methodology, predefined, structured parallel exploitation patterns are implemented by translating them into data flow graphs executed by a scalable, efficient, distributed macro data flow interpreter (the term *macro* data flow refers to the fact that the computation of a single data flow instruction can be a substantial computation). User-defined, possibly unstructured parallelism exploitation patterns can be programmed by explicitly defining data flow graphs. These data flow graphs can be used in the skeleton system in any place where predefined skeletons can be used, thus providing the possibility of seamlessly integrating both kinds of parallelism exploitation within the same program.

*This work has been partially supported by Italian national FIRB project no. RBNE01KNFP GRID.it and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).
†Dept. Computer Science—Univ. of Pisa & Programming Model Institute, CoreGRID
‡ISTI–CNR, Pisa & IMT—Institute for Advanced Studies, Lucca & Programming Model Institute, CoreGRID

User-defined data flow graphs provide users with the possibility of programming new skeletons. However, in order to introduce a new skeleton, users need concentrate only on the data flow within the new skeleton, rather than on the implementation issues typically related to the efficient implementation of structured parallelism exploitation patterns. This greatly improves the efficacy of the parallel application development process as compared to classical parallel programming approaches such as MPI and OpenMP that instead provide users with very low level mechanisms and give them complete responsibility for efficiently and correctly using these mechanisms to implement the required parallelism exploitation patterns.

After describing how user defined skeletons are introduced and supported within our experimental skeleton programming environment, we will also briefly discuss other tools we are currently considering to extend the prototype skeleton environment. These tools extend the possibility for users to control some non-functional features of parallel programs in a relatively high-level way. In particular we will introduce the possibility of using Java 1.5 annotations and AOP (Aspect-Oriented Programming) techniques to associate to the skeletons different non-functional properties such as security or parallelism exploitation related properties.

2. Template based vs. data flow based skeleton systems. A skeleton based parallel programming environment provides programmers with a set of predefined and parametric parallelism exploitation patterns. The patterns are parametric in the kind of basic computation executed in parallel and, possibly, in the execution parallelism degree or in some other execution related parameters. For example, a pipeline skeleton takes as parameters the computations to be computed at the pipeline stages. In some skeleton systems these computations can be either sequential computations or parallel ones (i. e. other skeletons) while in other systems (mainly the ones developed at the very beginning of the skeleton related research activity) these computations may only be sequential ones.

The first attempts to implement skeleton programming environments all relied on the implementation template technology. Original Cole skeletons [12], Darlington's group skeleton systems [18, 20, 19], Kuchen's Muesli [23, 26] and our group's P3L [7] and ASSIST [36] all use this implementation schema. As discussed in [27], in an implementation template based skeleton system each skeleton is implemented using a parametric process network chosen from those available in a template library for that particular skeleton and for the kind of target architecture at hand (see [28], which discusses several implementation templates, all suitable for implementing task farms, that is embarrassingly parallel computations implemented according to a master-worker paradigm). The template library is designed once and for all by the skeleton system designer and captures the state of the art knowledge relating to implementation of the parallelism exploitation patterns modeled by the skeletons. Therefore the compilation process of a skeleton program, according to the implementation template model, can be summarized as follows:

1. the skeleton program is parsed and a skeleton tree representing the precise skeleton structure of the user application is derived. The skeleton tree has nodes marked with one of the available skeletons, and leaves marked with sequential code (sequential skeletons).
2. The skeleton tree is traversed, in some order, and templates from the library are assigned to each of the skeleton nodes, apart from the sequential ones, which always correspond to the execution of a sequential process on the target machine. During this phase, parameters of the templates (e.g. the parallelism degree or the kind of communication mechanisms used) are fixed, possibly using heuristics associated with the library entries.
3. The annotated skeleton tree is used to generate the actual parallel code. Depending on the system this may involve a traditional compilation step (e.g. in P3L when using the Anacleto compiler [11] or in ASSIST when using the `astcc` compiler tools [2, 1]) or use of a skeleton library hosting templates (e.g. Muesli [26] and eSkel [14] exploiting MPI).
4. The parallel code is eventually run on the target architecture, possibly exploiting some kind of loader/ deploy tool.

Figure 2.1 summarizes the process of deriving running code from skeleton source code using template technology.

More recently, an implementation methodology based on data flow has been proposed [15]. In this case the skeleton source code is used to compile a data flow graph and the data flow graph is then executed on the target architecture using a suitable distributed data flow interpreter engine. The approach has been used both in our group, in the implementation of Lithium [35, 6], and in Serot's SKIPPER skeleton environment [30]. In both cases the data flow approach was used to support fixed skeleton set programming environments. We adopted the very same implementation approach in the `muskel` full Java skeleton library, but in `muskel`

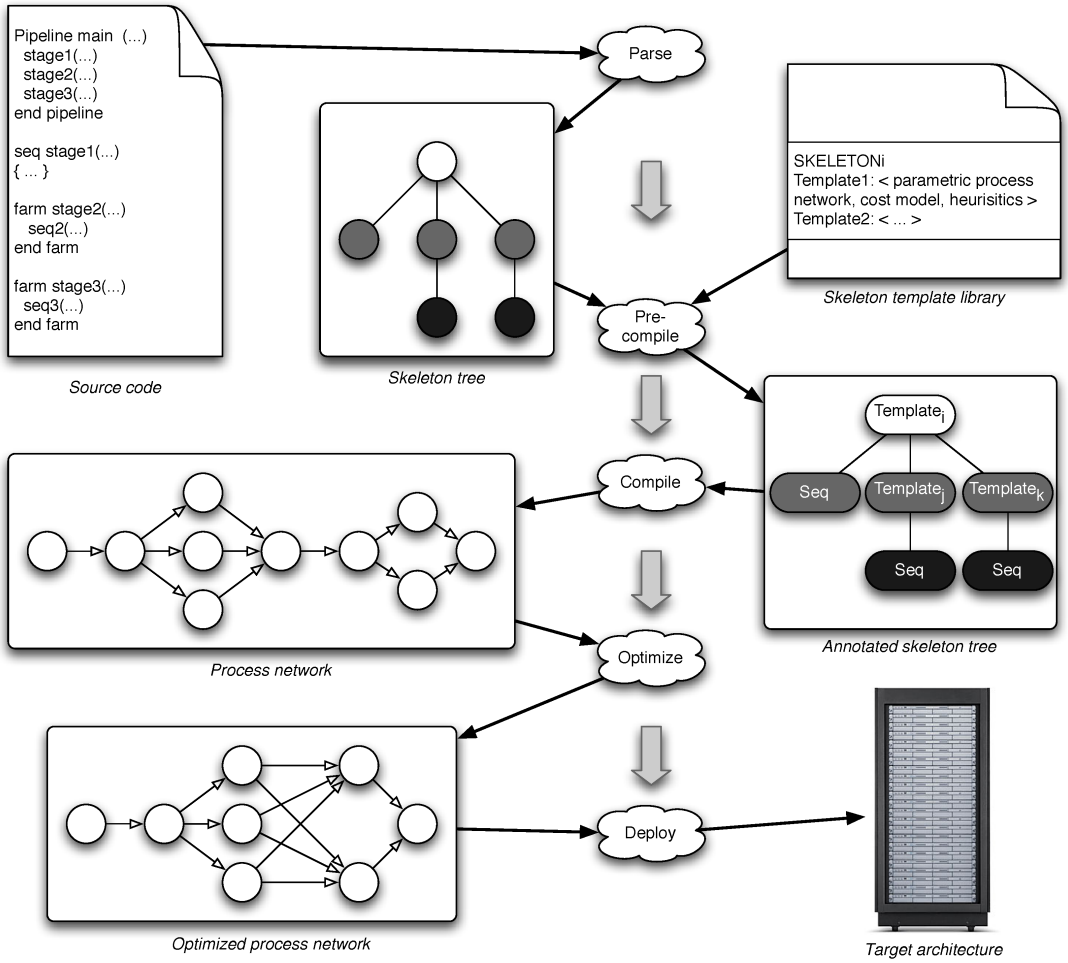

FIG. 2.1. *Skeleton program execution according to the implementation template approach.*

(as shown in the rest of this paper) the data flow implementation is also used to support extensible skeleton sets.

When data flow technology is exploited to implement skeletons, the compilation process of a skeleton program can be summarized as follows:

1. the skeleton program is parsed and a data flow graph is derived. The data flow graph represents the pure data flow behaviour of the skeleton tree in the program.
2. For each of the input tasks, a copy of the data flow graph is instantiated, with the task appearing as an input token to the graph. The new graph is delivered to the distributed data flow interpreter "instruction pool".
3. The distributed data flow interpreter fetches fireable instructions from the instruction pool and the instructions are executed on the nodes in the target architecture. Possibly, optimizations are taken into account (based on heuristics) that try to avoid unnecessary communications (e.g. caching tokens that will eventually be reused) or to adapt the computation grain of the program to the target architecture features (e.g. delivering more than a single fireable instruction to remote nodes to decrease the impact of communication set up latency, or multiprocessing the remote nodes to achieve communication and computation overlap).

Figure 2.2 summarizes the steps leading from skeleton source code to the running code using this data flow approach. It is worth pointing out that macro data flow implementation of skeletons is "pure data flow" compliant: no side effects, such as those deriving from the usage of global variables, are supported, nor can data flow graphs compiled from one skeleton in the program affect/modify the graphs compiled from the other skeletons

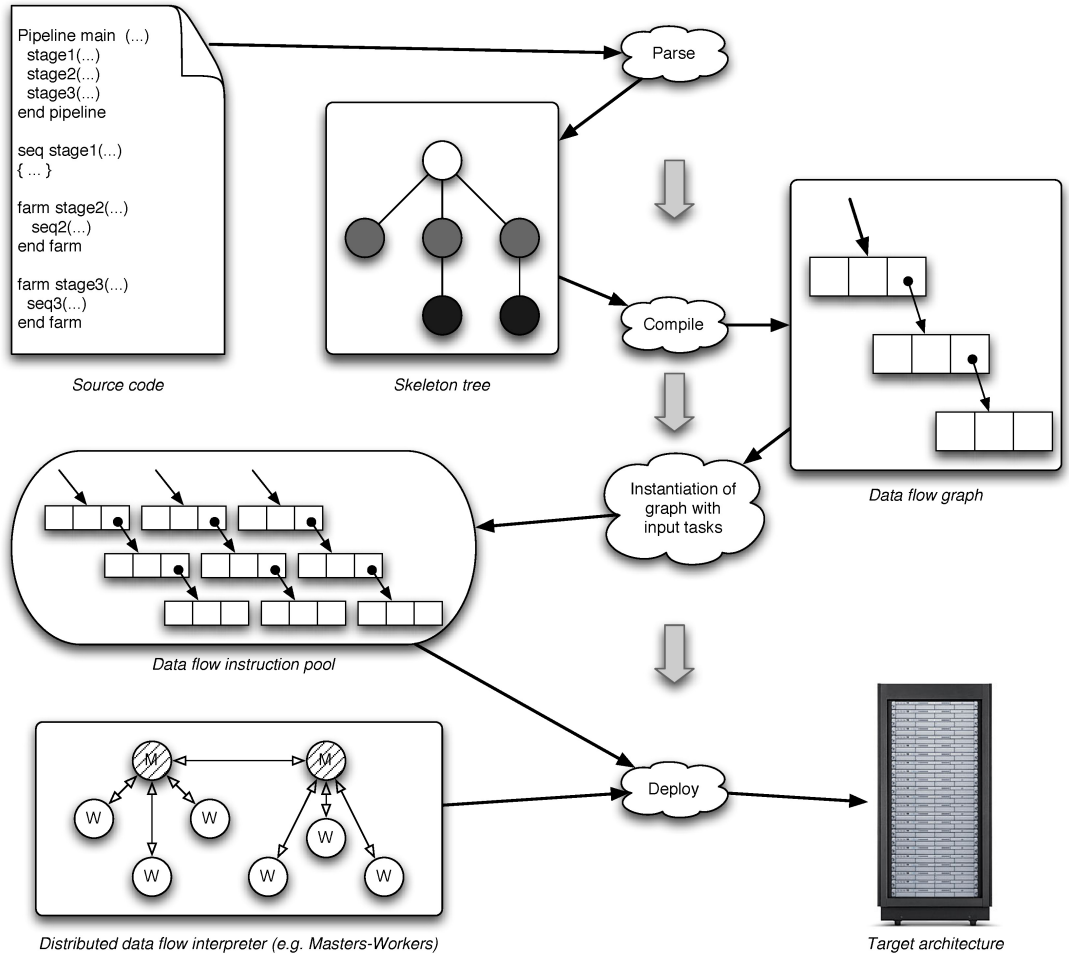

FIG. 2.2. *Skeleton program execution according to the data flow approach.*

in the program. This can be perceived as a limitation if we assume a non-structured parallel programming perspective. However, this represents a strong point in the structured parallel programming perspective as it guarantees that macro data flow graphs separately generated from skeletons appearing in the source code can be composed/unfolded safely in the global macro data flow graph eventually run on the distributed macro data flow interpreter.

The two approaches just outlined appear very different, but they have been successfully used to implement different skeleton systems. To support what will be presented in §4.2, we wish first to point out a quite subtle difference in the two approaches.

On the one hand, when using implementation templates, the process network eventually run on the target architecture is very similar to the one the user has in mind when instantiating skeletons in the source code. In some systems the "optimization" phase of Fig. 2.1 is actually empty and the program eventually run on the target architecture is built by simple juxtaposition of the process networks making up the templates of the skeletons used in the program. Even when the optimization phase does actually modify the process network structure (in Fig. 2.1 the master/slave service process of the two consecutive farms are optimized/collapsed, for instance), the overall structure of the process network does not change very much.

On the other hand, when a data flow approach is used the process network run on the target architecture has almost nothing to do with the skeleton tree described by the programmer in the source code. Rather, the skeleton tree is used to implement the parallel computation in a correct and efficient way, exploiting a set of techniques and mechanisms that are much closer to the techniques and mechanisms used in operating systems rather than to those used in the execution of parallel programs, both structured and unstructured. From a slightly different perspective, this can be interpreted as follows:

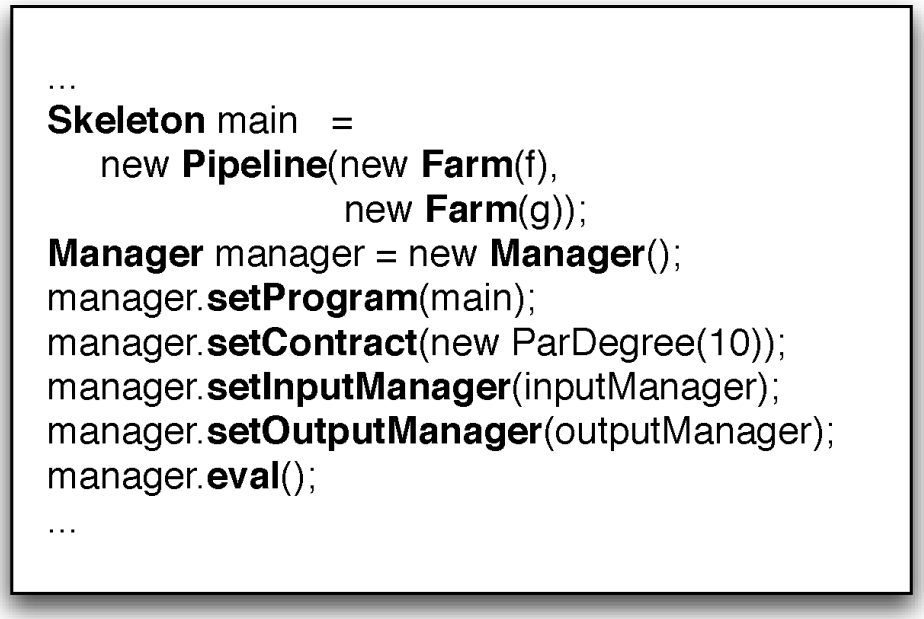

FIG. 3.1. *Sample* `muskel` *code: sketch of all (but the sequential portions of code) the code needed to set up and execute a two-stage pipeline with parallel stages (farms).*

- skeletons in the program "annotate" sequential code by providing the meta information required to efficiently implement the program in parallel;
- the support tools of the skeleton programming environment (the data flow graph compiler and the distributed data flow interpreter, in this case) "interprets" the meta information to accurately and efficiently implement the skeleton program, exploiting (possibly at run time, when the target architecture features are known) the whole set of known mechanisms supporting implementation optimization (e.g. caches, prefetching, node multiprocessing, etc.).

Viewed in this way, the data flow implementation for parallel skeleton programs presents a new perspective in the design of parallel programming systems where parallelism is dealt with as a "non-functional" feature, introduced by programmers via annotations or exploiting Aspect-Oriented Programming (AOP) techniques, and handled by the compiling/runtime support tools in the most convenient and efficient way with respect to the target architecture at hand (see §4.2).

3. `muskel`**.** `muskel` is a full Java skeleton programming environment derived from Lithium [6]. Currently, it provides only the stream parallel skeletons of Lithium, namely stateless task farm and pipeline. These skeletons can be arbitrarily nested, to program pipelines with farm stages, for example, and they process a single stream of input tasks to produce a single stream of output tasks. `muskel` implements skeletons using data flow technology and Java RMI facilities. The programmer using `muskel` can express parallel computations by simply using the provided `Pipeline` and `Farm` classes. For example, to express a parallel computation structured as a two-stage pipeline with a farm in each of the stages, the user should write code such as that shown in Fig. 3.1. `f` and `g` are two classes implementing the `Skeleton` interface, i. e. supplying a `compute` method with the signature `Object compute(Object t)` computing `f` and `g`, respectively. The `Skeleton` interface represents the "sequential" skeleton, that is the skeleton always executed sequentially and only used to wrap sequential code in such a way that it can be used in other, non-sequential skeletons.

In order to execute the program, the programmer first sets up a `Manager` object. Then, using appropriate methods, he indicates to the manager the program to execute, the performance contract required (in this case, the parallelism degree required for the execution), what is in charge of providing the input data (the input stream manager, which is basically an iterator providing the classical `boolean hasNext()` and `Object next()`

methods) and what is in charge of processing the output data (the output stream manager, providing only a `void deliver(Object)` method processing a single result of the program). Finally, he can request parallel program execution simply by issuing an `eval` call to the manager. When the call terminates, the output file has been produced.

Actually, the `eval` method execution happens in steps. First, the application manager looks for available processing elements using a simplified, multicast based peer-to-peer discovery protocol, and recruits the required remote processing elements. Each remote processing element runs a data flow interpreter. Then the skeleton program (the `main` of the example) is compiled into a macro data flow graph (capitalizing on normal form results shown in [3, 6]) and a thread is forked for each of the remote processing elements recruited. Then the input stream is read. For each task item, an instance of the macro data flow graph is created and the task item token is stored in the proper place (initial data flow instruction(s)). The graph is placed in the task pool, the repository for data flow instructions to be executed. Each thread looks for a fireable instruction in the task pool and delivers it for execution to the associated remote data flow interpreter. The remote interpreter instance associated to the thread is initialized by being sent the serialized code of the data flow instructions, once and for all, before the computation actually starts. Once the remote interpreter terminates the execution of the data flow instruction, the thread either stores the result token in the appropriate "next" data flow instruction(s) in the task pool, or it directly writes the result to the output stream, invoking the `deliver` method of the output stream manager. Currently, the task pool is a centralized one, associated with the centralized manager. We are currently investigating the possibility to distribute both task pool and manager so as to remove this bottleneck. The `manager` takes care of ensuring that the performance contract is satisfied. If a remote node "disappears" (e.g. due to a network failure, or to the node failure/shutdown), the manager looks for another node and starts dispatching data flow instructions to the new node instead [16]. As the manager is a centralized entity, if it fails, the whole computation fails. However, the manager is usually run on the user machine, which is assumed to be safer than the remote nodes recruited as remote interpreter instances.

The policies implemented by the `muskel` managers are *best effort*. The `muskel` library tries to do its best to accomplish user requests. If it is not possible to completely satisfy the user requests, the library establishes the closest configuration to the one implicitly specified by the user with the performance contract. In the example above, the library tries to recruit 10 remote interpreters. If only $n < 10$ remote interpreters are found, the parallelism degree is set exactly to n. In the worst case, that is if no remote interpreter is found, the computation is performed sequentially, on the local processing element.

In the current version of the `muskel` prototype, the only performance contract actually implemented is the `ParDegree` one, asking for the use of a constant number of remote interpreters in the execution of the program. The prototype has been designed to support at least another kind of contract: the `ServiceTime` one. This contract can be used to specify the maximum amount of time expected between the delivery of two program result tokens. Thus, with a call such as `manager.setContract(new ServiceTime(500))`, the user may request delivery of one result every half a second (time is in ms, as usual in Java). We do not discuss in more detail the implementation of the distributed data flow interpreter here. The interested reader can refer to [15, 16]. Instead, we will present more detail of the compilation of skeleton code into data flow graphs.

A `muskel` parallel skeleton code is described by the grammar:

$$\mathsf{P} ::= \mathsf{seq}(\mathit{className}) \mid \mathsf{pipe}(\mathsf{P}, \mathsf{P}) \mid \mathsf{farm}(\mathsf{P})$$

where the `className`s refer to classes implementing the `Skeleton` interface, and a macro data flow instruction (MDFi) is a tuple:

$$\mathsf{MDFi} \equiv Id \times Id \times Id \times \mathcal{I}^n \times \mathcal{O}^k$$

where the first Id : $paper.tex, v1.352007/03/2316 : 45 : 59 marcodExp$ represents the MDFi identifier distinguishing that MDFi from other MDFi in the graph, the second represents the graph id (both are either integers or the special `NoId` identifier), the third the identifier of the Skeleton code computed by the MDFi; and, finally, $\mathcal{I}$ and $\mathcal{O}$ are the input tokens and the output token destinations, respectively. An input token is a pair $\langle \mathit{value}, \mathit{presenceBit} \rangle$ and an output token destination is a pair $\langle \mathit{destInstructionId}, \mathit{destTokenNumber} \rangle$. With these assumptions, a data flow instruction such as $\langle a, b, \mathtt{f}, \langle \langle 123, \mathtt{true} \rangle, \langle \mathtt{null}, \mathtt{false} \rangle \rangle, \langle \langle i, j \rangle \rangle \rangle$ is the instruction with identifier a belonging to the graph with identifier b. It has two input tokens, one present (the integer 123) and one not present yet. It is not fireable, as one token is missing. When the missing token is delivered to this

instruction, either from the input stream or from another instruction, the instruction becomes fireable. To be computed, the two tokens must be given to the `compute` method of the `f` class. The method computes a single result that will be delivered to the instruction with identifier i in the same graph, in the position corresponding to input token number j. The process compiling the skeleton program into the data flow graph can therefore be more formally described as follows. We define a pre-compile function $PC : \mathsf{P} \times Id \rightarrow (Id \rightarrow \mathsf{MDFi}^\star)$ as follows:

$$PC[\mathsf{P}]_g = \begin{cases} \lambda i.\ \{\langle newId(), g, \mathtt{f}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle i, 1\rangle\rangle\rangle\} & \textit{if } \mathsf{P} = \mathsf{seq}\ (\mathtt{f}) \\ PC[\mathsf{P}_1]_g & \textit{if } \mathsf{P} = \mathsf{farm}(\mathsf{P}_1\ ) \\ \lambda i.((PC[\mathsf{P}_1]_{gid}\ (getId(T))) \cup (T(i))) & \\ \quad \text{where } T = PC[\mathsf{P}_2]_{gid} & \textit{if } \mathsf{P} = \mathsf{pipe}\ (\mathsf{P}_1, \mathsf{P}_2) \end{cases}$$

where $\lambda x.T$ is the usual lambda notation for functions and $getID()$ returns the id of the first instruction in its argument graph, that is, the one assuming to receive the input token from outside the graph.
Then, we define the compile function $C : \mathsf{P} \rightarrow \mathsf{MDFi}^\star$ as follows:

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

where $newId()$ and $newGid()$ are stateful functions returning a fresh (i. e. unused) instruction and graph identifier, respectively. The compile function therefore returns a graph, with a fresh graph identifier, containing all the data flow instructions defining the skeleton program. The result tokens are identified as those whose destination is `NoId`. For 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:

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

(assuming that identifiers and token positions start from 1).

When the application manager is told to 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 replace the data field of the initial 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:

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

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

All the resulting instructions are put in the task pool of the distributed interpreter 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 `NoId` are delivered directly to the output file stream by the threads receiving them from the remote interpreter instances. Those with a non-`NoId` flag are delivered to the appropriate instructions in the task pool, which will eventually become fireable.

4. Expanding `muskel` skeleton facilities. In this section, we will discuss how the skeleton facilities provided by `muskel` can be extended to accomplish particular user requirements. Two issues are considered. First, the mechanisms used to allow programmers to define their own skeletons are discussed, along with their `muskel` implementation. Using these mechanisms, the programmers may declare and use arbitrary, possibly "unstructured"[1] new skeletons. Then, we discuss how alternative mechanisms based on Java annotations and/or AOP techniques are currently being used to provide further expandability of the `muskel` skeleton set, in particular characterizing existing skeletons with new, non-functional features.

4.1. User-defined skeletons. In order to introduce completely new parallelism exploitation patterns, `muskel` provides programmers with mechanisms that can be used to design arbitrary macro data flow graphs. A macro data flow graph can be defined creating some `Mdfi` (macro data flow instruction) objects and connecting them in a `MdfGraph` object.

For example, the code in Fig. 4.1 is that needed to program a data flow graph with two instructions. The first computes the `inc1 compute` method on its input token and delivers the result to the second instruction. The second computes the `sq1 compute` method on its input token and delivers the result to a generic "next" instruction (this is modelled by giving the destination token tag a `Mdfi.NoInstrId` tag). The `Dest` type in the code represents the destination of output tokens as triples containing the graph identifier, the instruction

[1] With respect to classical skeleton frameworks.

```
Skeleton inc1 = new Inc();
Dest d = new Dest(0, 2 ,Mdfi.NoGraphId);
Dest[] dests = new Dest[1];
dests[0] = d;
Mdfi i1 = new Mdfi(manager,1,inc1,1,1,dests);
Skeleton sq1 = new Square();
Dest d1 = new Dest(0,Mdfi.NoInstrId, Mdfi.NoGraphId);
Dest[] dests1 = new Dest[1];
dests1[0] = d1;
Mdfi i2 = new Mdfi(manager,2,sq1,1,1,dests1);
MdfGraph graph = new MdfGraph();
graph.addInstruction(i1);
graph.addInstruction(i2);
ParCompute userDefMDFg = new ParCompute(graph);
```

FIG. 4.1. *Custom/user-defined skeleton declaration.*

identifier and the destination input token targeted in this instruction. Macro data flow instructions are built by specifying the manager they refer to, their identifier, the code executed (must be a `Skeleton` object) the number of input and output tokens and a vector with a destination for each of the output tokens.

We do not present all the details of arbitrary macro data flow graph construction here (a complete description is provided with the `muskel` documentation). The example is just to give the flavor of the tools provided in the `muskel` environment. Bear in mind that the simple macro data flow graph of Fig. 4.1 is actually the same macro data flow graph obtained by compiling a primitive `muskel` skeleton call such as: `Skeleton main = new Pipeline(new Inc(), new Sq())` More complex user-defined macro data flow graphs may include instructions delivering tokens to an arbitrary number of other instructions, as well as instructions gathering input tokens from several distinct other instructions. In general, the mechanisms of `muskel` permit the definition of any kind of graph with macro data flow instructions computing sequential (side effect free) code wrapped in a `Skeleton` class. Any parallel algorithm that can be modeled with a data flow graph can therefore be expressed in `muskel`[2]. Non deterministic MDFi are not yet supported (e.g. one that merges input tokens from two distinct sources) although the firing mechanism in the interpreter can be easily adapted to support this kind of macro data flow instructions. Therefore, new skeletons added through the macro data flow mechanism always model pure functions.

`MdfGraph` objects are used to create new `ParCompute` objects. `ParCompute` objects can be used in any place where a `Skeleton` object is used. Therefore, user-defined parallelism exploitation patterns can be used as pipeline stages or as farm workers, for instance. The only limitation on the graphs that can be used in a `ParCompute` object consists in requiring that the graph has a unique input token and a unique output token.

When executing programs with user-defined parallelism exploitation patterns the process of compiling skeleton code to macro data flow graphs is slightly modified. When an original `muskel` skeleton is compiled, the process described in §3 is applied. When a user-defined skeleton is compiled, the associated macro data flow graph is directly taken from the `ParCompute` instance variables where the graph supplied by the user is maintained. Such a graph is linked to the rest of the graph according to the rules appropriate to the skeleton where the user-defined skeleton appears.

To show how the whole process works, let us suppose we want to pre-process each input task in such a way that for each task t_i a new task

$$t'_i = h_1(f_1(t_i), g_2(g_1(f_1(t_i))))$$

[2]Note, however, that common, well know parallel application skeletons are already modelled by pre-defined `muskel Skeletons`.

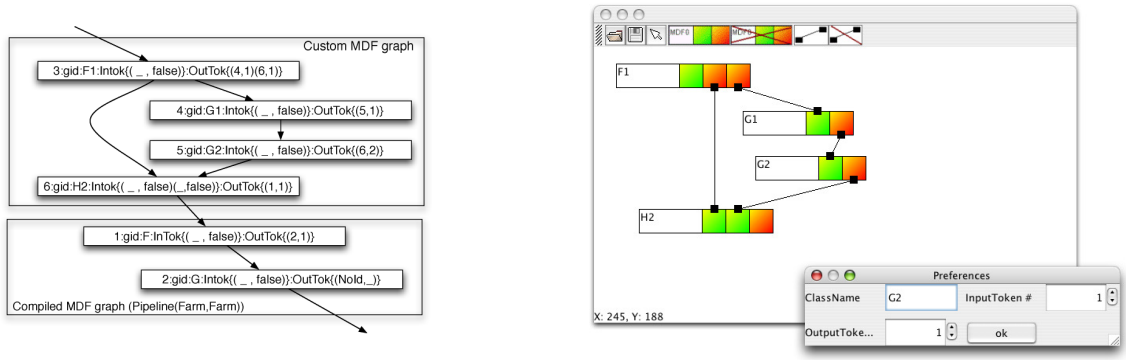

FIG. 4.2. *Mixed sample macro data flow graph (left): the upper part comes from a user-defined macro data flow graph (it cannot be derived using primitive* `muskel` *skeletons) and the lower part is actually coming from a three stage pipeline with two sequential stages (the second and the third one) and a parallel first stage (the user-defined one). GUI tool designing the upper graph (right).*

is produced. This computation cannot be programmed using the stream parallel skeletons currently provided by `muskel`. In particular, current pre-defined skeletons in `muskel` allow only processing of one input to produce one output, and therefore there is no way to implement the graph described here. In this case we wish to process the intermediate results through a two-stage pipeline to produce the final result. To do this the programmer can set up a new graph using code similar to the one shown in Fig. 3.1 and then use that new `ParCompute` object as the first stage of a two-stage pipeline whose second stage happens to be the postprocessing two-stage pipeline. When compiling the whole program, the outer pipeline is compiled first. As the first stage is a user-defined skeleton, its macro data flow graph is directly taken from the user-supplied one. The second stage is compiled according to the (recursive) procedure described in §3 and eventually the (unique) last instruction of the first graph is modified in such a way that it sends its only output token to the first instruction in the second stage graph. The resulting graph is outlined in Fig. 4.2 (left).

Making good use of the mechanisms allowing definition of new data flow graphs, the programmer can arrange to express computations with arbitrary mixes of user-defined data flow graphs and graphs coming from the compilation of structured, stream parallel skeleton computations. The execution of the resulting data flow graph is supported by the `muskel` distributed data flow interpreter in the same way as the execution of any other data flow graph derived from the compilation of a skeleton program. At the moment the `muskel` prototype allows user-defined skeletons to be used as parameters of primitive `muskel` skeletons, but not vice versa. We are currently working to extend `muskel` to alow the latter.

While the facility to include user-defined skeletons provides substantial flexibility, we recognize that the current way of expressing new macro data flow graphs is error prone and not very practical. Therefore we have designed a graphic tool that allows users to design their macro data flow graphs and then compile them to actual Java code as required by `muskel` and shown above. Fig. 4.2 (right) shows the interface presented to the user. In this case, the user is defining the upper part of the graph in the left part of the same Figure. It is worth pointing out that all that is needed in this case is to connect output and input token boxes appropriately, and to configure each MDFi with the name of the sequential `Skeleton` used. The smaller window on the right lower corner is the one used to configure each node in the graph (that is, each MDFi). This GUI tool produces an XML representation of the graph. Then, another Java tool produces the correct `muskel` code implementing the macro data flow graph as a `muskel ParCompute` skeleton. As a result, users are allowed to extend, if required, the skeleton set by just interacting with the GUI tool and "compiling" the graphic MDF graph to `muskel` code by clicking on one of the buttons in the top toolbar.

As a final example, consider the code of Fig. 4.3. This code outlines how a new `Map2` skeleton, performing in parallel the same computation on all the portions of an input vector, can be defined and used. It is worth pointing out how user-defined skeletons, once properly debugged and fine-tuned, can simply be incorporated in

```
public class Map2 extends ParCompute {

    public Map2(Skeleton f, Manager manager) {
        super(null);
        // first build the empty graph
        program = new MdfGraph();
        // build the emitter instruction
        Dest [] dds1 = new Dest[2];
        dds1[0]=new Dest(0,2);
        dds1[1]=new Dest(0,3);
        Mdfi emitter =
                new Mdfi(manager, 1,
                new MapEmitter(2), 1, 2, dds1);
        // add it to the graph
        program.addInstruction(emitter);
        // build first half map Skeleton node
        Dest [] dds2 = new Dest[1];
        dds2[0] = new Dest(0,4);
        Mdfi if1 = new Mdfi(manager,2, f, 1, 1, dds2);
        program.addInstruction(if1);
        // build second half map Skeleton node
        Dest []dds3 = new Dest[1];
        dds3[0] = new Dest(1,4);
        Mdfi if2 = new Mdfi(manager,3, f, 1, 1, dds3);
        program.addInstruction(if2);
        Dest[] ddslast = new Dest[1];
        ddslast[0] = new Dest(0,Mdfi.NoInstrId);
        Mdfi collector = new Mdfi(manager,4,new
                        MapCollector(), 2, 1, ddslast);
        program.addInstruction(collector);
        return;
    }
    ...
}
```

```
public class SampleMap {
    public static void main(String[] args) {
        Manager manager =
            new Manager();
        Skeleton worker = new Fdouble();
        Skeleton main =
            new Map2(worker,manager);

        InputManager inManager =
            new DoubleVectIM(10,4);
        OutputManager outManager =
            new DoubleVectOM();

        ParDegree contract =
            new ParDegree(10);
        manager.setInputManager(inManager);
        manager.setOutputManager(outManager);
        manager.setContract(contract);
        manager.setProgram(main);

        manager.compute();
    }
}
```

FIG. 4.3. *Introducing a new, user-defined skeleton: a map working on vectors and with a fixed, user-defined parallelism degree.*

the `muskel` skeleton library and used seamlessly, as the primitive `muskel` ones, but for the fact that (as shown in the code) the constructor needs the manager as a parameter. This is needed so as to be able to link together the macro data flow graphs generated by the compiler and those supplied by the user. It is worth noting that skeletons such as a general form of `Map` are usually provided in the fixed skeleton set of any skeleton system and users usually do not need to implement them. However, as `muskel` is an experimental skeleton system, we concentrate the implementation efforts on features such as the autonomic managers, portability, security and expandability rather than providing a complete skeleton set. As a consequence, `muskel` has no predefined map skeleton and the example of user defined skeleton just presented suitably illustrates the methodology used to expand the "temporary" restricted skeleton set of the current version of `muskel` depending on the user needs. The `Map2` code shown here implements a "fixed parallelism degree" *map*, that is the number of "workers" used to compute in parallel the skeleton does not depend on the size of the input data. It is representative of a more general `Map`skeleton taking a parameter specifying the number of workers to be used. However, in order to support the implementation of a map skeleton with the number of workers defined as a function of the input data, some kind of support for the dynamic generation of macro data flow graphs is needed, which is not present in the current `muskel` prototype.

4.2. Non-functional features. We briefly discuss here how annotations and aspect-oriented programming techniques and mechanisms can be used to introduce convenient ways of expressing non-functional features of parallel skeleton programs in `muskel`. Unlike the work discussed in the previous section, which has already been implemented in the current `muskel` prototype, this is more on-going work. We have preliminary results demonstrating the approach is feasible and we are currently working to transfer the experimental techniques to the "production" `muskel` prototype.

```
public aspect Normalize {

      pointcut callSetProgram(Skeleton c):
            call(public void Manager.setProgram(Skeleton)) && args(c);


         void around(Skeleton c)
      : callSetProgram(c) {
            proceed(new NormalForm(c));
         }

}
```

FIG. 4.4. *AspectJ code modeling normal form in* `muskel`.

In this context, let us consider as non-functional features all that is not related to the control flow that the programmer needs to set up to compute the final program result. For instance, we consider as non-functional features the necessity to secure code and data management in a program execution, the application of optimization rules transforming the user-supplied program into an equivalent, possibly more efficient one, or the hints given by programmers as to the features to exploit during the execution of the parallel skeleton code.

We wish to outline how these features can be implemented in the `muskel` framework using some innovative programming techniques.

First consider security issues. When executing a `muskel` program on a network of workstations, it may be the case that the workstations used happen to be in different local networks, possibly interconnected by public, untrusted network segments. Also, it may be the case that the user running the program does not have complete control of the machines used to run the remote data flow interpreter instances, and therefore cannot exclude malicious user activity on the remote machines aimed at reading or modifying the program or the data involved in the parallel program run. Therefore, it is appropriate to provide mechanisms that can be used in the `muskel` support to authenticate and encrypt all the communications happening during a `muskel` program run, both those relating to the transmission of the (serialized) program code and those relating to input and output token communications. As an example, an `ssl` transport layer can be used instead of plain TCP/IP to implement the `muskel` communications. However, the use of the `ssl` transport layer involves a communication cost which is definitely higher than the cost involved in plain TCP/IP configurations (see results shown in §5). Therefore, the user may wish to denote in the program which are the sensitive data or code segments that must not be transmitted in clear on untrusted networks. Java annotations can be used to the purpose, as follows:

- the programmer annotates (using some `@SensitiveCode` and `@SensitiveData` annotations) those `Skeleton`s whose code must be properly secured and those data that must be kept secret;
- then the `Manager`, in the `eval` implementation may use reflection to access these annotations and to process them properly. That is, in the case of `Skeleton` objects annotated as `@SensitiveCode` it provides for distribution of the code using `ssl` tunnelled RMI, in the case of tasks/tokens annotated as `@SensitiveData` it provides for invocation of remote `compute` execution again using `ssl` tunnelled RMI, while in all other cases it uses plain RMI over unencrypted, more efficient TCP/IP connections.

Now consider a different kind of non-functional feature: source-to-source program optimization rules. For example, let us consider our previous result on skeleton program *normal form*. Such result [3] can be informally stated as follows: an arbitrary `muskel` program whose structure is a generic skeleton tree made out of pipelines, farms and sequential skeletons may be transformed into a new, equivalent one, whose parallel structure is a farm with each worker made up of the sequential composition of the sequential skeletons appearing in the original skeleton tree taken left to right. This second program is the skeleton program normal form and happens to

```
public aspect Normalize {

	public boolean ContractSpecified = false;
	public boolean normalized = false;
	// contract is an integer, to simplify ...
	public int contract = 0;

	pointcut calledContract(int i): call(public void Manager.setContract(int)) && args(i);

	void around(int i): calledContract(i){
		ContractSpecified = true;
		contract = i;
		proceed(i);
	}

	pointcut callSetProgram(Skeleton c): call(public void Manager.setProgram(Skeleton)) && args(c);

	void around(Skeleton c):
	callSetProgram(c) {
		normalized = true;
		proceed(new NormalForm(c));
	}

	pointcut callEval(Manager m) : call(public void Manager.eval()) && target(m);

	before(Manager m):callEval(m){
		if(ContractSpecified)
			if(normalized)
				m.setContract(Manager.NormalizeContract(contract));
		else
			m.setContract(Manager.DefaultNormalizedContract);
		}
	}
}
```

FIG. 4.5. *AspectJ code handling performance contracts in* `muskel`.

perform better than the original one in the general case and in the same way in the worst case (this with respect to the service time). As an example, the code of Fig. 3.1 can be transformed into the equivalent normal form code: `Skeleton main = new Farm(new Seq(f,g));` where `Seq` is basically a pipeline whose stages are executed sequentially on a single processor.

In Lithium, normal form can be used by explicitly inserting statements in the source code. This means that the user must change the source code to use the normal form or the non-normal form version of the same program. Using AOP (and AspectJ, in particular) we can define an aspect dealing with normal form transformation by defining a pointcut on the execution of the `setProgram Manager` method and associating to the pointcut the action performing normal form transformation on the source code in the aspect, such as the one of Fig. 4.4. As a consequence, the user can decide whether to use the original or the normal form version of the program just by choosing the standard Java compiler or the AspectJ one. The fact that the program is left unchanged means the programmer may debug the original program and have the normal form one debugged too as a consequence, provided the AOP code in the normal form aspect is correct, of course. Moreover, if normal form is handled by aspects as discussed above, it is better to handle also related features by means of suitable aspects. For example, if the user provided a performance contract (a parallelism degree, in the simpler case) and then used the AspectJ compiler to request normal form execution of the program, it turns out to be quite natural to imagine a further aspect handling the performance contract consequently. Fig. 4.5 shows the AspectJ code handling this feature. In this case, contracts are stored as soon as they have been issued by the programmer, with the first pointcut, then, when normalization has been required (second pointcut) and program parallel evaluation is required, the contract is handled consequently (third pointcut); in this case it is either left unchanged or a new contract is derived from the original one according to some normal form related procedure.

At the moment we are experimenting with both annotations and AOP techniques to provide the `muskel` programmer with better tools supporting more and more possibilities to customize parallelism exploitation in `muskel` programs.

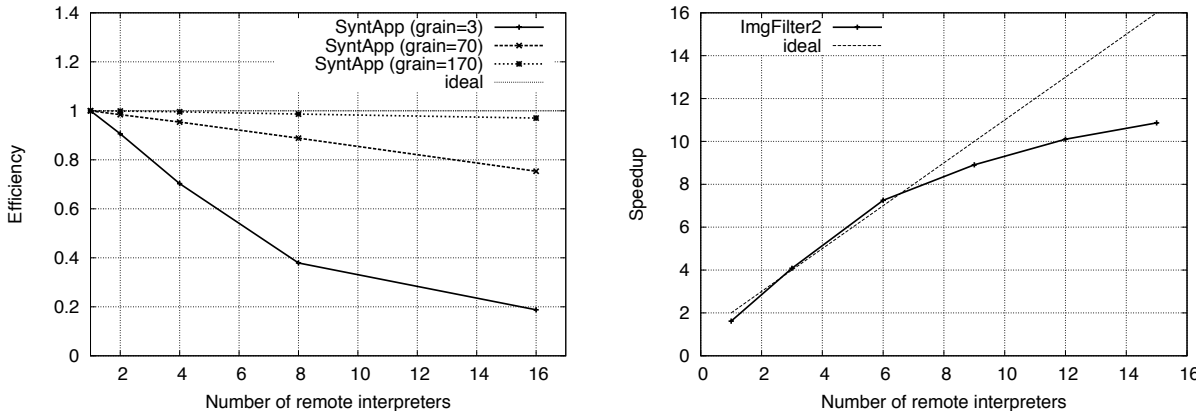

FIG. 4.6. *muskel performance versus number of remote interpreters on a homogeneous cluster. Left) Efficiency of* SyntApp *for several computation grains. Right) Speedup of* ImgFilter2 *compared with ideal speedup.*

In particular, we have investigated the possibility of relieving the programmer of the need to specify farm skeletons at all. Instead of declaring farm skeletons, programmers may simply annotate as `@Parallel` the `Skeleton` objects and the run time support directly manages to transform calls to the `compute` methods of such objects into farms [17]. This is not a completely new technique, but it can be used to evaluate the effectiveness of the approach, compared both to the original `muskel` farm handling and to a similar approach defining `Skeleton` objects to be computed in parallel in a farm by properly setting up a farm aspect with actions establishing task farm like computation patterns upon the invocation of the `Skeleton compute` method.

5. Experimental results. We ran some experiments aimed at validating the `muskel` prototype supporting user defined skeletons. The results shown refer to two applications. SyntApp is a synthetic application processing 1K distinct input tasks and designed in such a way that the macro data flow instructions appearing in the graph had a precise "average grain" (i. e. average ratio among the time spent computing the instruction at the remote interpreter and the time spent communicating data to and from the remote interpreter, $G = T_w/T_c$). ImgFilter2 is an image processing application based on the pipeline skeleton, which applies two filters in sequence to 30 input images. All input images are true-color (24 bit color depth) of 640x480 pixels size. ImgFilter2 basically applies "blur" and "oil" filters (available at `http://jiu.sourceforge.net`) from the Java Imaging Utilities in sequence as two stages of a pipeline. Note that these are *area filter operations*, i. e. the computation of each pixel's color does not only impact its direct neighbours, but also an adjustable area of neighboring pixels. By choosing five neighboring pixels in each direction as filter workspaces, we made the application more complex and enforced several iterations over the input data within each pipeline stage, which makes our filtering example a good representative of a compute intensive application [5].

Two types of parallel platforms are used for experimentation. The first is a dedicated Linux cluster at the University of Pisa. The cluster hosts 24 nodes: one node devoted to cluster administration and 18 nodes (P3@800MHz) exclusively devoted to parallel program execution. The second is a grid-like environment, including two organizations: the University of Pisa *(di.unipi.it)* and an institute of the Italian National Research Council in Pisa *(isti.cnr.it)*. The server set is composed of several different Intel Pentium and Apple PowerPC computers, running Linux and Mac OS X respectively (the detailed configuration is shown in Figure 5.1 left). In this case traditional measures like efficiency and speedup versus number of machines cannot be used due to the machines' power heterogeneity. To take the varying computing power of different machines into account, the performance increase is documented by means of the *BogoPower* measure, defined as the sum of individual BogoPower contributions of machines participating in the application run. The BogoPower of each machine is measured in terms of tasks/s the sequential version of the application can compute on the machine. BogoPower enables the comparison between an application's actual parallel performance and the application's ideal performance for each run [5].

Figure 4.6 summarizes the typical performance results of the enhanced interpreter. The left plot is relative to runs of SyntApp on the homogeneous cluster. The experiment shows that, in case of low grain, `muskel` rapidly loses efficiency with the number of machines involved in the computation. When the grain is high

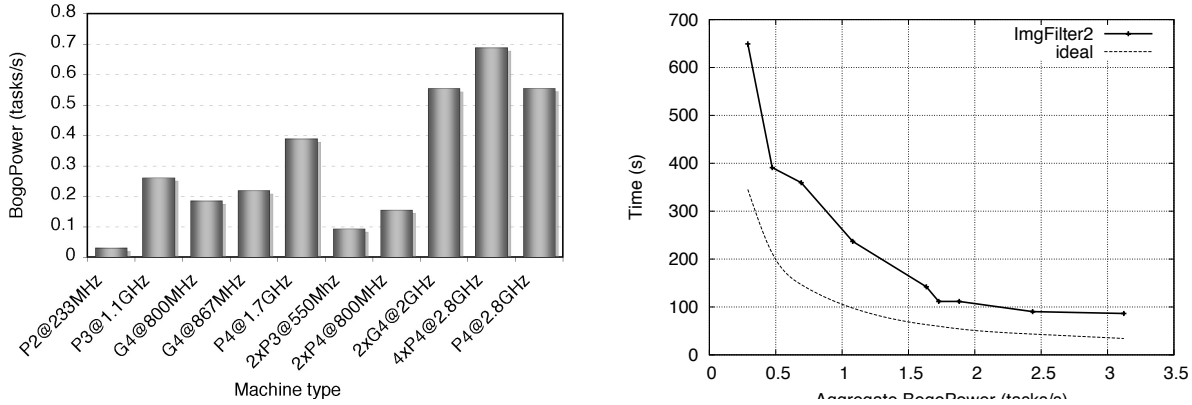

FIG. 5.1. *muskel performance on a grid-like testing environment. Left) Description of platforms in the testing environment (machine-type, Bogopower). Right) Completion time of ImgFilter2*

enough ($G = 200$ or more) the efficiency is definitely close to the ideal one. The right plot shows the speedup of ImgFilter2 on the same homogeneous cluster, instead.

Figure 5.1 right plots the completion time of ImgFilter2 executed on an heterogeneous network of Linux/Pentium and MacOsX PowerPC machines, whose relative performance is shown in the left part of the same Figure. The measured completion times show the same shape as the theoretical ones, confirming that the `muskel` run time efficiently and automatically balances the load when different (with respect to computing power) resources are used to allocate the `muskel` distributed macro data flow interpreter.

In addition to the evaluation of the scalability of the `muskel` prototype, we also have taken into account the possibility of using different mechanisms to support distributed data flow interpreter execution. We implemented several versions of `muskel` on top of ProActive [29], each exploiting different mechanisms, primitive to the ProActive library, to deploy and run remote macro data flow interpreter instances. In particular, we used ProActive XML deployment descriptors as well as RMI `ssh` tunnelling. When possible, we exploited the option to pre-allocate JVMs running the remote interpreter instances on the remote processing elements, to speed up program startup. The results showed that in the case where the remote JVMs are preallocated, the performance is definitely comparable to the performance of plain `muskel`. In the case of use of RMI tunelling through `ssh`, however, larger grain macro data flow instructions (close to 10 times larger grain) are needed to achieve almost perfect speedup.

As discussed in §3, appropriate security mechanisms, defined using Java 1.5 annotations, should be used to guarantee that data and code moved to and from the remote data flow interpreter instances are kept confidential and that intruders cannot use remote data flow interpreter instances to execute non-authorized macro data flow code. We conducted some experiments to evaluate the effectiveness of introducing selective security annotations in the code. We prepared a stripped `muskel` prototype version, using `ssl` to secure interaction between the main code running on the user machine and the remote data flow interpreter instances. With the `muskel` prototype exploiting `ssl` [34], we managed to measure the scalability penalty paid to introduce security. We verified that "secure" `muskel` scales close to ideal values when using up to 32 nodes for the remote macro data flow interpreter instances, similarly to plain `muskel`. However, due to the encrypting/decrypting activity taking place at the sending/receiving nodes, larger (i. e.more compute intensive) macro data flow instructions are required to achieve ideal scalability (see Figure 5.2 left). Also, we measured the load distribution in runs involving half "secure" and half "non-secure" remote interpreter instances. Communications with the secure interpreter instances are performed using plain TCP/IP, while communications with the non-secure ones are performed using SSL. With higher and higher amounts of data transferred to and from the remote interpreters more and more computation is performed on the secure nodes. This is due to the auto scheduling strategy of `muskel` that always dispatches computations to the free remote interpreter instances. As more data is transmitted, more time is spent securing communications through SSL and more time is spent computing a single MDFi. Therefore less MDFi are actually executed at the non-secure nodes (see Figure 5.2 right). The results shown are perfectly in line with what is stated in §4: securing `muskel` communications is quite costly and therefore it

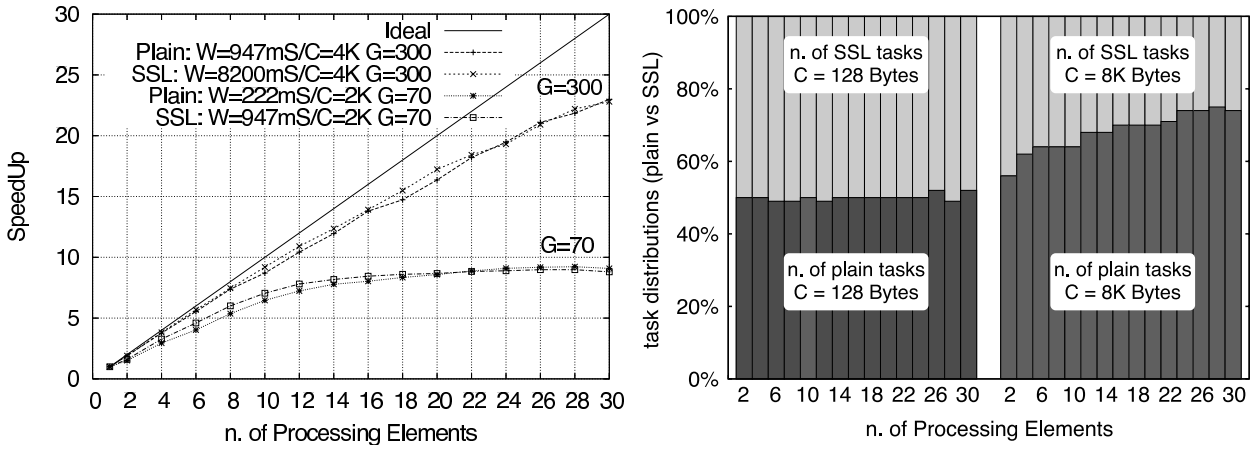

FIG. 5.2. *Effect of providing security in the distributed data flow interpreter: scalability of the* `muskel` *prototype using plain TCP/IP sockets vs. the one using SSL for different computational grains. W represent the average time spent computing a single MDFi, C the average amount of data sent/received to/from remote processing elements to compute the single MDFi*

is better to avoid securing communications not involving sensitive data and/or code. And this can be done by exploiting the annotation mechanisms just outlined in §4. Further details concerning security issues in `muskel` are discussed in [4].

6. Related work. Macro data flow implementation for the algorithmical skeleton programming environments was introduced by the authors in the late 90's [15] and subsequently has been used in other contexts related to skeleton programming environments [31]. Cole suggested in [13] that "we must construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well defined way", and that structured parallel programming environments should "accommodate diversity", that is "we must be careful to draw a balance between our desire for abstract simplicity and the pragmatic need for flexibility". Actually, his eSkel [9, 14] MPI skeleton library addresses these problems by allowing programmers to program their own peculiar MPI code within each process in the skeleton tree. Programmers can ask to have a stage of a pipeline or a worker in a farm running on k processors. Then, the programmer may use the k process communicators returned by the library for the stage/worker to implement its own parallel pipeline stage/worker process. As far as we know, this is the only other attempt to integrate ad hoc, unstructured parallelism exploitation in a structured parallel programming environment. The implementation of eSkel, however, is based on process templates, rather than on data flow. Other skeleton libraries, such as Muesli [23, 24, 26], provide programmers with quite extensive flexibility in skeleton programming following a different approach. They provide a number of data parallel data structures along with elementary, collective data parallel operations that can be arbitrarily nested to get more and more complex data parallel skeletons. However, this flexibility is restricted to the data parallel part, and it is, in any case, limited by the available collective operations.

CO2P3S [25] is a design pattern based parallel programming environment written in Java and targeting symmetric multiprocessors. In CO2P3S, programmers are allowed to program their own parallel design patterns (skeletons) by interacting with the intermediate implementation level [10]. Again, this environment does not use data flow technology but implements design patterns using proper process network templates.

JaSkel [21] provides a skeleton library implementing the same skeleton set as `muskel`. In JaSkel, however, skeletons look much more like implementation templates, according to the terminology used in §2. However, it appears that the user can exploit the full OO programming methodology to specialize the skeletons to his own needs. As the user is involved in the management of support code too (e.g. he has to specify the master process/thread of a task farm skeletons) JaSkel can be classified as a kind of "low level, extensible" skeleton system, although it is not clear from the paper whether entirely new skeletons can be easily added to the system (actually, it looks like it is not possible at all).

There are several works proposing aspect-oriented techniques for parallel programming. [22] discusses an approach using AOP to separate concerns in scientific code. In [33, 32] a use of AOP is proposed aimed at separating the concerns of partitioning and distributing data and performing concurrent computations. This is

far from the usage we think to make of AOP techniques in this work, however, in that it requires a much more "template oriented" approach with respect to the one followed in `muskel`.

7. Conclusions. We discussed `muskel`, a full Java, parallel programming library providing users with the possibility to use skeletons to structure their parallel applications and exploiting macro data flow implementation technology. We discussed how `muskel` supports expandability of the skeleton set, as advocated by Cole in his "manifesto" paper [13]. In particular, we discussed how `muskel` supports both the introduction of new skeletons, modeling parallelism exploitation patterns not originally covered by the primitive `muskel` skeletons, and the introduction of non-functional features, i. e. features related to parallel program execution but not directly related to the functional computation of the application results. The former possibility is supported by allowing users to define new skeletons providing the arbitrary data flow graph executed in the skeleton and by allowing `muskel` to seamlessly integrate such new skeletons with the primitive ones. The latter possibility is supported by exploiting more innovative programming techniques such as annotations and aspect-oriented programming. This second part is under development, while the first is already available in the `muskel` prototype.

We also presented experimental results validating the whole `muskel` approach to expandability and customizability of its skeleton set. As far as we know, this is the most significant effort in the skeleton community to tackle problems deriving from a fixed skeleton set. Only Schaeffer and his group at the University of Alberta implemented a system were users can, in controlled ways, insert new parallelism exploitation patterns in the system [10], although the approach followed there is a bit different, in that users are encouraged to intervene directly in the run time support implementation, to introduce new skeletons, while in `muskel` new skeletons may be introduced using the intermediate macro data flow language as the skeleton "assembly" language.

Finally, we discussed how relatively new programming techniques, including annotations and AOP, can be usefully exploited in `muskel` to support details and features related to parallel program execution.

Preliminary versions of `muskel` have been released under GPL and are currently available on the muskel web site at `htpp://www.di.unipi.it/~marcod/muskel`. The new version, supporting the features discussed in this paper, is currently being developed. The support for new skeletons is already completed (and it is available, as a beta release, on the web site) and the other features will be released soon.

Acknowledgements. We wish to thank Daniele Licari, who implemented the graphic interface supporting user friendly design of new `muskel` skeletons. We wish also to thank Peter Kilpatrick for all the very useful and inspirational discussions we had on `muskel`, and for helping us proofread this paper.

REFERENCES

[1] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo, *A framework for experimenting with structured parallel programming environment design*, in Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, eds., vol. 13 of Advances in Parallel Computing, Dresden, Germany, 2004, Elsevier, pp. 617–624.

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

[3] M. Aldinucci and M. Danelutto, *Stream parallel skeleton optimization*, in Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems, Cambridge, Massachusetts, USA, Nov. 1999, IASTED, ACTA press, pp. 955–962.

[4] ———, *The cost of security in skeletal systems*, in Euromicro PDP 2007: Parallel Distributed and network-based Processing, IEEE, February 2007. Naples, Italy.

[5] M. Aldinucci, M. Danelutto, J. Dünnweber, and S. Gorlatch, *Optimization techniques for skeletons on grid*, in Grid Computing and New Frontiers of High Performance Processing, L. Grandinetti, ed., vol. 14 of Advances in Parallel Computing, Elsevier, Oct. 2005, ch. 2, pp. 255–273.

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

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

[8] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi, *SkIE: A heterogeneous environment for HPC applications*, Parallel Computing, 25 (1999), pp. 1827–1852.

[9] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, *Flexible skeletal programming with eSkel*, in Proc. of 11th Intl. Euro-Par 2005 Parallel Processing, J. C. Cunha and P. D. Medeiros, eds., vol. 3648 of LNCS, Lisboa, Portugal, Aug. 2005, Springer, pp. 761–770.

[10] S. Bromling, *Generalising pattern-based parallel programming systems*, in Parallel Computing: Advances and Current Issues. Proc. of the Intl. Conference ParCo2001, G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, eds., Imperial College Press, 2002.
[11] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti, *ANACLETO: a template-based P3L compiler*, in Proc. of the Parallel Computing Workshop (PCW'97), 1997. Camberra, Australia.
[12] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computations*, Research Monographs in Parallel and Distributed Computing, Pitman, 1989.
[13] ———, *Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming*, Parallel Computing, 30 (2004), pp. 389–406.
[14] M. Cole and A. Benoit, *The eSkel home page*. `http://homepages.inf.ed.ac.uk/abenoit1/eSkel/`.
[15] M. Danelutto, *Dynamic run time support for skeletons*, in Proc. of Intl. PARCO 99: Parallel Computing, E. H. D'Hollander, G. R. Joubert, F. J. Peters, and H. J. Sips, eds., Parallel Computing Fundamentals & Applications, Imperial College Press, 1999, pp. 460–467.
[16] ———, *QoS in parallel programming through application managers*, in Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing, Lugano, Switzerland, Feb. 2005, IEEE, pp. 282–289.
[17] M. Danelutto, P. Dazzi, D. Laforenza, M. Pasin, L. Presti, and M. Vanneschi, *PAL: High level parallel programming with Java annotations*, in Proceedings of CoreGRID Integration Workshop (CIW 2006) Krakow, Poland, Academic Computer Centre CYFRONET AGH, Oct. 2006, pp. 189–200. ISBN 83-915141-6-1.
[18] J. Darlington, A. J. Field, P. Harrison, P. H. J. Kelly, D. W. N. Sharp, R. L. While, and Q. Wu, *Parallel programming using skeleton functions*, in Proc. of Parallel Architectures and Languages Europe (PARLE'93), vol. 694 of LNCS, Munich, Germany, June 1993, Springer, pp. 146–160.
[19] J. Darlington, M. Ghanem, and H. W. To, *Structured Parallel Programming*, in Programming Models for Massively Parallel Computers, Berlin, Germany, September 1993, IEEE Computer Society Press.
[20] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler, *Fortran-S: a uniform functional interface to parallel imperative languages*, in Third Parallel Computing Workshop (PCW'94), Fujitsu Laboratories Ltd., November 1994.
[21] J. F. Ferreira, J. L. Sobral, and A. J. Proença, *JaSkel: A Java skeleton-based framework for structured cluster and grid computing.*, in CCGRID, IEEE Computer Society, 2006, pp. 301–304.
[22] B. Harbulot and J. R. Gurd, *Using aspectj to separate concerns in parallel scientific java code.*, in AOSD, G. C. Murphy and K. J. Lieberherr, eds., ACM, 2004, pp. 122–131.
[23] H. Kuchen, *A skeleton library*, in Proc. of 8th Intl. Euro-Par 2002 Parallel Processing, B. Monien and R. Feldman, eds., vol. 2400 of LNCS, Paderborn, Germany, Aug. 2002, Springer, pp. 620–629.
[24] ———, *Optimizing sequences of skeleton calls*, in Domain-Specific Program Generation, D. Batory, C. Consel, C. Lengauer, and M. Odersky, eds., vol. 3016 of LNCS, Springer, 2004, pp. 254–273.
[25] S. McDonald, D. Szafron, J. Schaeffer, and S. Bromling, *Generating parallel program frameworks from parallel design patterns*, in Proc of. 6th Intl. Euro-Par 2000 Parallel Processing, A. Bode, T. Ludwing, W. Karl, and R. Wismüller, eds., vol. 1900 of LNCS, Springer, Aug. 2000, pp. 95–105.
[26] *The muesli home page*, 2006. `http://www-wi.uni-muenster.de/pi/personal/kuchen.php`.
[27] S. Pelagatti, *Structured Development of Parallel Programs*, Taylor & Francis, 1998.
[28] M. Poldner and H. Kuchen, *Scalable farms*, in Parallel Computing: Current & Future Issues of High-End Computing, Proc. of PARCO 2005, G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, eds., vol. 33 of NIC, Germany, Dec. 2005, Research Centre Jülich, pp. 795–802.
[29] *ProActive home page*, 2006. `http://www-sop.inria.fr/oasis/proactive/`.
[30] J. Serot, *Tagged-token data-flow for skeletons*, Parallel Processing Letters, 11 (2001), pp. 377–392.
[31] J. Serot and D. Ginhac, *Skeletons for parallel image processing: an overview of the SKIPPER project*, Parallel Computing, 28 (2002), pp. 1685–1708.
[32] J. L. Sobral, *Incrementally Developing Parallel Applications with AspectJ*, in Proc. of 20th Intl. Parallel & Distributed Processing Symposium (IPDPS), IEEE, April 2006.
[33] J. L. Sobral, M. P. Monteiro, and C. A. Cunha, *Aspect-oriented support for modular parallel computing*, in Proc. of the 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, Y. Coady, D. H. Lorenz, O. Spinczyk, and E. Wohlstadter, eds., Bonn, Germany, March 2006, Published as University of Virginia Computer Science Technical Report CS–2006–01, pp. 37–41.
[34] *JSSE for the Java 2 SDK*, 2006. `http://java.sun.com/products/jsse/index-14.html`.
[35] P. Teti, *Lithium: a Java skeleton environment*, Master's thesis, Dept. Computer Science, University of Pisa, October 2001. In Italian.
[36] M. Vanneschi, *The programming model of ASSIST, an environment for parallel and distributed portable applications*, Parallel Computing, 28 (2002), pp. 1709–1732.

Edited by: Anne Benoît and Frédéric Loulergue
Received: September, 2006
Accepted: March, 2007