Joint Structured/Unstructured Parallelism Exploitation in muskel*

M. Danelutto[1,4] and P. Dazzi[2,3,4]

[1] Dept. Computer Science, University of Pisa, Italy
[2] ISTI/CNR, Pisa, Italy
[3] IMT – Institute for Advanced Studies, Lucca, Italy
[4] CoreGRID Institute on Programming model

Abstract. Structured parallel programming promises to raise the level of abstraction perceived by programmers when implementing parallel applications. In the meanwhile, however, it restricts the freedom of programmers to implement arbitrary parallelism exploitation patterns. In this work we discuss a data flow implementation methodology for skeleton based structured parallel programming environments that easily integrates arbitrary, user-defined parallelism exploitation patterns while preserving most of the benefits typical of structured parallel programming models.

1 Introduction

Structured parallel programming models provide the user/programmers with native high level parallelism exploitation patterns that can be instantiated, possibly in a nested way, to implement a wide range of applications [6, 10, 3, 2]. In particular, those 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 all these parallelism exploitation related details. As an example, to implement an embarrassingly parallel application processing all the data items in an input stream or file, the programmers simply instantiates a "task farm" skeleton by providing the code necessary to (sequentially) process each input task item. The system, either a compiler and run time tool based implementation or the library based one, takes care of devising the proper distributed resources to be used, to schedule proper tasks on the resources and to distribute input tasks and gather output results according to the process mapping used. In contrast, when using a traditional system, the programmers have usually to explicitly program code for distributing and scheduling the processes on the available resources and for moving input and output data between the

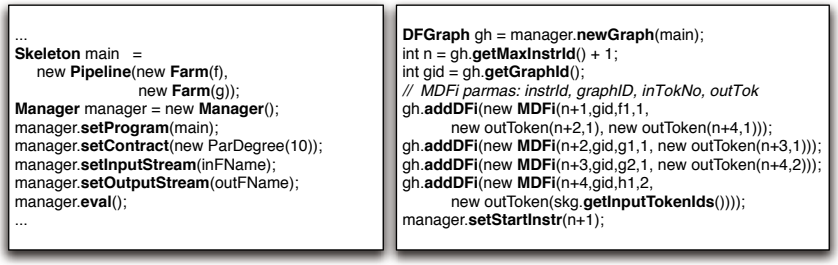
* This work has been partially supported by Italian national FIRB project no. RBNE01KNFP GRID.it and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part II, LNCS 3992, pp. 937–944, 2006.

involved processing elements. The cost of this appealing 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 only use the ones provided by the system, usually including all those reusable patterns that happen to have efficient distributed implementations available. This is mainly aimed at avoiding the possibly for the programmers to write code that can potentially impairs 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 [6]. In this work we discuss a methodology that can be used to implement mixed parallel programming environments providing the user with both structured and unstructured ways of expressing parallelism while preserving most of the benefits typical of structured parallel programming models. The methodology is based on data flow. Structured parallel exploitation patterns are implemented 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 huge computations). Unstructured parallelism exploitation can be achieved by explicitly programming data flow graphs. The system provides suitable ways to interact with the data flows graphs derived from structured pattern compilation in such a way that mixed structured and unstructured parallelism exploitation patterns can be used within the same application.

2 Data Flow Implementation of Structured Programming Environments

Muskel is a full Java skeleton programming environment derived from Lithium [2]. Currently, it only provides the stream parallel skeletons of Lithium, namely stateless task farm and pipeline. These skeletons can be arbitrary nested, to program pipelines with farm stages, as an example, and they process a single stream of input tasks to produce a single stream of output tasks. Muskel implements skeletons exploiting data flow technology and Java RMI facilities. The programmer using muskel can express parallel computations simply using the provided `Pipeline` and `Farm` classes. As an example, to express a parallel computation structured as a two-stage pipeline with a farm in each one of the stages, the user should write a code such as the one of Figure 1 left. `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. In order to execute the program, the programmer first sets up a `Manager` object, tells it which is the program to execute, which is the performance contract required (in this case, the parallelism degree required for the execution), the filename for the input file hosting the input stream items and the filename of the output file that will eventually host the result stream. Then he can ask parallel program execution simply issuing an `eval` call to the manager. When the call terminates, the output file has been produced.



```
...
Skeleton main  =
    new Pipeline(new Farm(f),
                 new Farm(g));
Manager manager = new Manager();
manager.setProgram(main);
manager.setContract(new ParDegree(10));
manager.setInputStream(inFName);
manager.setOutputStream(outFName);
manager.eval();
...
```

(a) Sample muskel code

```
DFGraph gh = manager.newGraph(main);
int n = gh.getMaxInstrId() + 1;
int gid = gh.getGraphId();
// MDFi parmas: instrId, graphID, inTokNo, outTok
gh.addDFi(new MDFi(n+1,gid,f1,1,
        new outToken(n+2,1), new outToken(n+4,1)));
gh.addDFi(new MDFi(n+2,gid,g1,1, new outToken(n+3,1)));
gh.addDFi(new MDFi(n+3,gid,g2,1, new outToken(n+4,2)));
gh.addDFi(new MDFi(n+4,gid,h1,2,
        new outToken(skg.getInputTokenIds())));
manager.setStartInstr(n+1);
```

(b) Adding custom MDF graph to skeleton MDF graph

Fig. 1. Sample muskel code (left) and custom/standard MDF graph integration (right)

Actually, the `eval` method execution happens in steps. First, the application manager looks for available processing elements using a simplified, multicast based peer2peer discovery protocol, and recruits the required remote processing elements. Each remote processing element runs a data flow interpreter, actually. Then the skeleton program (the `main` of the example) is compiled into a macro data flow graph (actually capitalizing on normal form results shown in [1, 2]) and a thread is forked for each one of the remote processing elements recruited. Then the input file 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 is initialized by being sent the serialized code of the data flow instructions once and forall 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 proper "next" data flow instruction(s) in the task pool, or it directly writes the result to the output file. 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, in such a way this bottleneck will eventually disappear. 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 [9]. 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. In case it is not possible to completely satisfy the user requests, the library accomplishes to establish 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. In case 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 usage of a constant number of remote interpreters in the execution of the program. The prototype has been thought 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 line code such as `manager.setContract(new ServiceTime(500))`, the user may ask to deliver one result every half a second (time is in millisecs, as usual in Java). We do not enter in more detail in the implementation of the distributed data flow interpreter here. The interested reader can refer to [8, 9]. Instead, we will try to give a better insight into 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 is a tuple: $\langle id, gid, opcode, \mathcal{I}^n, \mathcal{O}^k \rangle$ where id is the instruction identifier, gid is the graph identifier (both are either integers or the special *NoId* identifier), *opcode* is the name of the `Skeleton` class providing the code to compute the instruction (i.e. computing the output tokens out of the input ones) and $\mathcal{I}$ and $\mathcal{O}$ are the input tokens and the output token destinations, respectively. An input token is a pair $\langle value, presenceBit \rangle$ and an output token destination is a pair $\langle destInstructionId, destTokenNumber \rangle$. With this 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 will be delivered to this instruction, either coming 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 $PC[]$ as

$$\begin{aligned}
&PC[\mathsf{seq}(\mathtt{f})]_{gid} = \lambda i.\{\langle newId(), gid, \mathtt{f}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle i, NoId\rangle\rangle\rangle\} \\
&PC[\mathsf{farm}(\mathsf{P})]_{gid} = C[\mathsf{P}]_{gid} \\
&PC[\mathsf{pipe}(\mathsf{P}_1, \mathsf{P}_2)]_{gid} = \lambda i.\{C[\mathsf{P}_1]_{gid}(getId(C[\mathsf{P}_2]_{gid})), C[\mathsf{P}_2]_{gid}(i)\}
\end{aligned}$$

where $\lambda x.T$ it the usual function representation $((\lambda x.T)(y) = T|_{x=y})$ and $getID()$ 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, and a compile function $C[]$ such as

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

where $newId()$ and $newGid()$ are state full 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 `NoId`. As an example, the compilation of the `main` program pipe(farm(seq(f)), farm(seq(g))) produces the data flow graph

$\{\langle 1, 1, \texttt{f}, \langle\langle \texttt{null}, \texttt{false}\rangle\rangle, \langle\langle 2, 1\rangle\rangle\ , \langle 2, 1, \texttt{g}, \langle\langle \texttt{null}, \texttt{false}\rangle\rangle, \langle\langle \texttt{NoId}, \texttt{NoId}\rangle\rangle\rangle\}$
(assuming that identifiers and token positions start from 1).

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 replace the data field of the lower id 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:

$\{\langle 1, i, \texttt{f}, \langle\langle \texttt{task}_i, \texttt{true}\rangle\rangle, \langle\langle 2, 1\rangle\rangle\ , \langle 2, i, \texttt{g}, \langle\langle \texttt{null}, \texttt{false}\rangle\rangle, \langle\langle \texttt{NoId}, \texttt{NoId}\rangle\rangle\rangle\}$
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 directly delivered 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 proper instructions in the task pool that will eventually become fireable.

3 Joint Structured/Unstructured Parallelism Exploitation

We now take into account how unstructured parallelism exploitation mechanisms are provided to the muskel user. First, we provide the programmer with primitives to access the fundamental parameters of the data flow graph generated out of the compilation of a skeleton program. We provide methods to deliver data to and retrieve data from this graph. We provide the programmers with the ability to instantiate a new graph $C[\mathsf{P}]$ in the task pool by providing the input task token and to redirect the output token of $C[\mathsf{P}]$ to an arbitrary data flow instruction in the pool. This possibility is given through a new specific method of the `Manager` class, a `MdfGraph newGraph(Skeleton p)` method returning a `MdfGraph` handle for the new data flow graph generated out of the skeleton program `p`, and by two methods of the `MdfGraph` class, one (`IdPair getInputTokenId()`) returning the id identifying the first instruction (to be used to redirect a data token to this instruction) and the other one (`setOutputTokenId(int instrId, int graphId)`) writing the destination *id*s in the output token of the graph, used to redirect the output of macro data flow instruction to first instruction of the skeleton program. Second, we provide the programmer with direct access to the definition of data flow graphs, in such a way he can describe his particular parallelism exploitation patterns that cannot be efficiently implemented with the

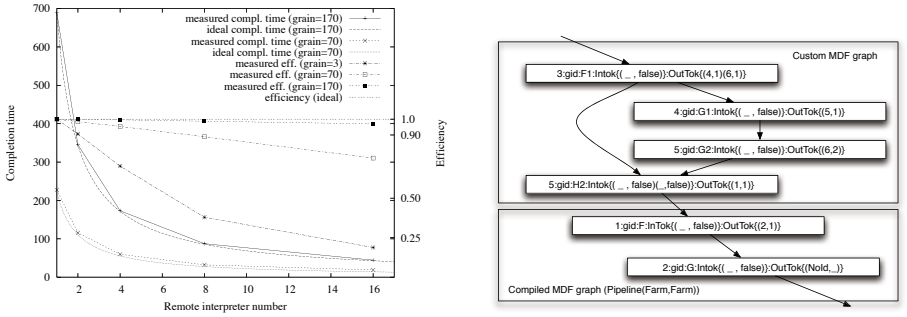


Fig. 2. Mixed sample MDF graph (right), scalability and effect of computation grain (left)

available skeletons. In particular, the programmer can instantiate new data flow instructions using the `MDFi` class constructors and accessory methods.

These two mechanisms can be jointly used to program all those parts of the application that cannot be easily and efficiently implemented using the skeletons subsystem. Let us suppose we want to pre-process the input tasks in such a way that for each task t_i a new task $t'_i = h_1(f_1(t_i), g_2(g_1(f_1(t_i))))$ is produced. This computation cannot be programmed using the stream parallel skeletons currently provided by muskel. Then we want to process the preprocessed tasks through the `main` two-stage pipeline described above, in order to produce the final result. In this case the programmer can set up a new graph using the code shown in Figure 1 right. The resulting MDF graph eventually executed by the muskel interpreter is outlined in Figure 2 (right), where the parts derived from the `main` and the custom parts provided by the user are outlined.

Making good usage of these mechanisms, the programmer can arrange to express computations with arbitrary mixes of arbitrary 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 as the execution of any other data flow graph derived from the compilation of a skeleton program.

4 Experimental Results

A muskel prototype implementation supporting the extensions described in this work is currently being tested. The muskel interpreter engine has been left basically unchanged, whereas the part supporting parallelism exploitation pattern programming has been changed to support linking of custom MDF graphs to the code produced by the compiler out of plain muskel skeleton trees. Figure 2 summarizes the typical performance results of the enhanced interpreter. We run several synthetic programs using the custom MDF graph features introduced in muskel. We designed the programs in such a way the MDF instructions appearing in the graph had a precise "average grain" (i.e. average ration among the time

spent computing the instruction at the remote interpreter and the time spent communicating data to and from the remote interpreter). Each run processed 1K input tasks. When grain is small, muskel do not scale, even using a very small number of remote interpreter instances. When the grain is high enough (about 200 times the time spent in communications actually spent in computation of MDF instructions) the efficiency is definitely close to the ideal one. Despite the shown data refer to synthetic computations, actual computations (e.g. image processing ones) achieved very similar results. This because the automatic load balancing mechanism implemented in the muskel distributed interpreter through auto scheduling perfectly optimized the execution of variable grain MDF instructions. All the experiments have been performed on a Linux (kernel 2.4.22) RLX Pentium III blade architecture, with Fast Ethernet interconnection among the blades, equipped with Java 1.4.1_01 run time.

5 Related Work

Macro data flow implementation for algoritmical skeleton programming environment was introduced by the authors in late '90 [8] and then has been used in other contexts related to skeleton programming environments [13]. Cole suggested in [6] 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 [4, 7] 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 processes communicator 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 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 [10, 12], provide programmers with a quite large 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 arbitrary nested to get more and more complex data parallel skeletons. However, this flexibility is restricted to the data parallel part, and it is anyway limited by the available collective operations.

COPS [11] is a design pattern based parallel programming environment written in Java and targeting symmetric multiprocessors. In COPS, programmers are allowed to program their own parallel design patterns (skeletons) by interacting with the intermediate implementation level [5]. Again, this environment does not use data flow technology but implements design patterns using proper process network templates.

6 Conclusions

We have shown how the existing data flow implementation of muskel can be extended to allow programmers to mix arbitrary parallelism exploitation patterns and structured ones in the same parallel application. We discussed preliminary performance results showing that the approach is feasible and efficient. The current version of muskel does not allow completely arbitrary custom graph and skeleton mix. As an example, it still does not support the usage of arbitrary MDF graphs as parameters of the skeletons. We are currently working to provide also this possiblity, however. Once completed, the new version of the muskel parallel Java library that will eventually be released under GPL on the muskel web site `www.di.unipi.it/~marcod/muskel`.

References

1. M. Aldinucci and M. Danelutto. Stream parallel skeleton optimisations. In *Proc. of the IASTED International Conference Parallel and Distributed Computing and Systems*, pages 955–962. IASTED/ACTA Press, November 1999. Boston, USA.
2. M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *FGCS*, 19(5):611–626, 2003.
3. B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, Dec. 1999.
4. A. Benoit, M. Cole, J. Hillston, and S. Gilmore. Flexible Skeletal Programming with eSkel. In *Proceedings of EuroPar 2005*, LNCS. Springer Verlag, Sept. 2005.
5. S. Bromling. Generalising Pattern-based Parallel Programming Systems. In *Proceedings of Parco 2001*. Imperial College Press, 2002.
6. M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.
7. M. Cole and A. Benoit. The Edinburgh Skeleton Library home page, 2005. http://homepages.inf.ed.ac.uk/abenoit1/eSkel/.
8. M. Danelutto. Dynamic Run Time Support for Skeletons. In *Proc. of the International Conference ParCo99*, Parallel Computing Fundamentals & Applications, pages 460–467. Imperial College Press, 1999.
9. M. Danelutto. QoS in parallel programming through application managers. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based processing*. IEEE, 2005. Lugano (CH).
10. H. Kuchen. A skeleton library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. Springer Verlag, August 2002.
11. S. McDonald, D. Szafron, J. Schaeffer, and S. Bromling. Generating Parallel Program Frameworks from Parallel Design Patterns. In *Euro-Par 2000 Parallel Processing*, LNCS, No. 1900, pages 95–105. Springer Verlag, August/September 2000.
12. Muesli home, 2005. www.wi.uni-muenster.de/PI/forschung/Skeletons/ index.php
13. J. Serot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKIPPER project. *Parallel computing*, 28(12):1685–1708, 2002.