Università di Pisa

Dipartimento di Informatica

Technical Report : TR-99-15

Dynamic Run Time Support for Skeletons

Marco Danelutto

July 15, 1999

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

Dynamic run time support for skeletons

M. Danelutto

Department of Computer Science – University of Pisa
Corso Italia, 40 – I-56125 PISA – Italy
marcod@di.unipi.it

Abstract

We propose a new implementation model for structured parallel programming models based on skeletons. The model is aimed at overcoming some of the problems observed in traditional, template based implementations of skeleton languages, while preserving all the positive features of structured parallel programming models. The proposed implementation model does not rely on static process networks like the template based implementation model for skeletons. It rather implements skeleton programs by properly scheduling macro data-flow instructions (corresponding to the sequential portions of code appearing in the skeleton code) to a set of macro data-flow interpreters running on the processing elements of the target architecture. We discuss some preliminary experimental results demonstrating the feasibility and effectiveness of our approach.

1 Introduction

Parallel, structured programming languages based on the skeleton concept [6] have been studied to provide the programmer with suitable tools to develop efficient and portable parallel applications in a reasonable time [1, 3, 4, 7, 13]. Basically, a skeleton is a known, reusable pattern of parallelism exploitation, and structured parallel programming languages based on the skeleton concept allow the programmer to code parallel applications by using proper (compositions of) skeletons. As the skeletons hide all the details involved in parallelism exploitation, such as process setup and scheduling, communication handling, load balancing, etc., the programmer can concentrate on the qualitative aspects of parallelism exploitation rather than on low-level, error-prone implementation details. Skeleton programs are usually compiled by generating a process network out of a known process network template for each of the skeletons appearing in the application code [12]. The process networks are then dimensioned (i.e. the parallelism degree of the network is computed) and mapped onto the processing elements (PEs) of the target architecture using heuristics derived from the information associated with the network templates. Once the mapping of processes to PEs has been devised, each PE in the target architecture becomes a specialized executor of some code belonging to the skeleton application.

[0]Work supported by the Italian MURST MOSAICO project and by the PQE2000 project

However, this implementation strategy presents some problems. At compile time, the PEs of the parallel target architecture are dedicated to execute a particular part of the skeleton program and therefore they cannot be used for other purposes even when they stay idle (e.g. during pipeline startup, the PEs dedicated to the execution of the final stages of the pipeline stay idle even if they could be used to perform the computations of the first pipeline stages). Furthermore, process networks are dimensioned statically. Any rearranging of the network at run time is impossible, even if the dimensioning turns out to be wrong, maybe due to the presence of irregular computations or in case the compiler performed an over and under-estimation of the parallelism degree of the skeleton.

In this work, we discuss a new implementation technique for skeleton based parallel programming languages aimed at solving the problems arising from a number of choices performed at compile time, such as the ones stated above.

2 The skeleton framework

We assume to use the skeleton parallel language `P3L` [2]. This language provides both *stream* parallel skeletons (including `pipe` and `farm` skeletons) and *data* parallel skeletons[1] (including a `map` skeleton). The `pipe` and `farm` skeletons model usual pipeline and task farm parallelism exploitation patterns. The `map` skeleton models data parallel computations where a set of (possibly overlapping) data partitions are obtained from each input data item, then a set of results is obtained (by applying the inner skeleton on each data partition) and eventually a single result is computed combining the results in the set. `P3L` also includes a `seq` skeleton which is used to encapsulate sequential portions of code in such a way that they can be used as skeleton parameters (i.e. as a `pipe` stage). Using this language, we can write programs as the following one:

```
seq f in(TASK1 x) out(float y[N][M], float z[N][M], BIGITEM a)
${ ... /* C code here computing y,z,a out of x */ }$ end seq
seq g in(float x, float y[N], BIGITEM z) out(float res)
${ ... /* C code here computing res out of x,y,z */ }$ end seq
seq h in(float x[N][M]) out(TASK2 y)
${ ... /* C code here computing y out of x */ }$ end seq
farm stage1 in(TASK1 x) out(float y[N][M], float z[N][M], BIGITEM a)
  f in(x) out(y,z,a)
end farm
map stage2 in(float x[N][M], float y[N][M], BIGITEM z) out(float res[N][M])
  g in(x[*i][*j], y[*i][], z) out(res)
end map
farm stage3 in(float x[N][M]) out(TASK2 y)
  h in(x) out(y)
end farm
pipe main in(TASK1 x) out(TASK2 y)
```

[1]stream parallel skeletons exploit parallelism between computations performed on different data items appearing on some kind of input stream (ordered sequence of data items of the same type, possibly available at different times), while data parallel skeletons exploit parallelism between computations relative to (possibly overlapping) partitions of a single data item belonging to a given input data stream

```
  stage1 in(x) out(float y[N][M], float z[N][M], BIGITEM a)
  stage2 in(y,z,a) out(float res[N][M])
  stage3 in(res) out(y)
end pipe
```

All the skeletons in the program compute the data items appearing in the `out` list out of the data items in the `in` list. The program processes a stream of `TASK1` data items producing a stream of `TASK2` data items. The outer parallel structure is a three stage pipeline. The first stage exploits farm parallelism in computing a triple holding two `float` matrixes and a `BIGITEM` data item. The second stage exploits data parallelism in computing a new matrix starting from each triple produced by the first stage. The new matrix is computed as follows: first, a set of $N \times M$ overlapping data sets is produced such that each set holds the i, j item of the first matrix, the whole i-th row of the second matrix and the whole `BIGITEM` data item ($\forall i \in [0, N-1], \forall j \in [0, M-1]$). Then, function g is computed over all the data sets. Eventually, a matrix is produced out of all the results of the computations of the function g, such that item i, j of the result matrix is the result of the computation of g over the data set including item i, j of the first matrix. The third stage exploits farm parallelism again, while computing a `TASK2` result out of each `float` matrix produced by the second, data parallel stage.

P3L provides a template based set of compiling tools [2, 5] that compile skeleton source code to a set of communicating processes. At compile time, the parallelism degree (i.e. the number of worker processes in the farms and the number of processes in the `map`) of all the skeletons is either computed or taken from user code.

P3L includes more skeletons than those used in the example, but `farm`, `pipe`line and `map` skeletons constitute a suitable, small set of skeletons that can be used to discuss and demonstrate the topics covered by this paper. It's worthwhile to point out that other skeletons based languages/programming models include skeletons similar to those included in P3L [1, 13, 4].

The computation of each result appearing on the output stream proceeds through five distinct steps: 1) an input task is processed by computing the sequential code relative to the first stage; 2) the results of such computation are split into subsets according to the parameters of the `map` skeleton; 3) each set is processed by computing the sequential code appearing as a parameter of the `map` skeleton; 4) a result is built from all the results deriving from step 3; 5) this result is used as input to compute the sequential code appearing in the last pipeline stage.

Therefore we can state that the computation of a skeleton program on a single input data item can be implemented as the coordinated execution of the sequential code appearing in the program in the order the `seq` skeletons appear in the normal visit of the application skeleton tree. Furthermore, we can state that execution of sequential portions of code embedded in the `seq` skeletons is accomplished according to the plain data flow model: as soon as a new data set is ready, to be computed by using some `seq` code, a new process computing that code can be started.

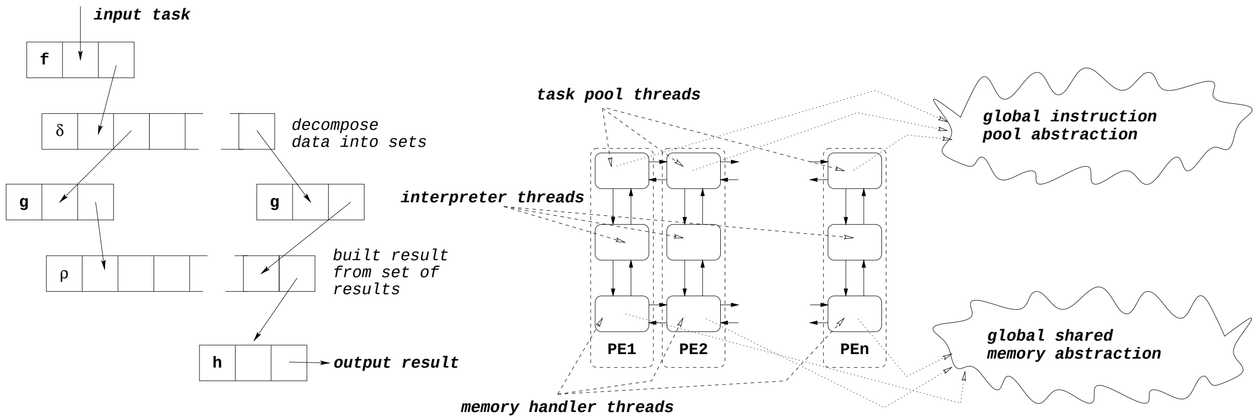

Figure 1: Macro Data Flow code for the example program of Section 2 (left) Target architecture abstraction (right)

3 A data flow implementation model for skeletons

The new model we propose to implement skeleton parallel programs is based on macro data flow, and can be summarized as follows: skeleton code is translated in macro data flow code, first. Then the macro data flow code is evaluated on a parallel architecture by a distributed interpreter of data flow code. The distributed interpreter, is an enhanced version of a plain macro data flow interpreter. The enhancements basically consist in advanced memory management and macro data flow instruction pool management. They are possible just because of the exploitation of the knowledge deriving from the skeleton structure of the source code.

Skeleton programs are translated into data flow code by generating a macro data flow instruction from each `seq` skeleton appearing in the program. Such instructions wait for input data, compute the code embedded in the `seq` skeleton as soon as the input data are available and eventually deliver the output data to some other instruction. The input and output arcs to each instruction (i.e. the places where input data is taken from and output data is sent to) are determined by the skeleton structure of the program. Figure 1 shows the macro data flow code corresponding to the program presented in Section 2.[2]

A collection of instances of an "interpreter" process is run on the target architecture, one process per PE. Each process has three distinct threads: a thread (the *task pool* thread) participates in managing a global macro data flow instruction pool, which is the union of all the instruction pools local to the different interpreter processes. A second thread (the *interpreter* thread) fetches fireable instructions (i.e. instructions with all the input data available) from the task pool thread, computes the instructions and deliver the results to the task pool thread. A third thread (the *memory handler*) manages a local data cache (Figure 1 right). In addition to the interpreter processes, two further processes are run on the target architecture: a first one delivering data items from the program input stream to the instructions that must eventually consume them, according to the skeleton structure of the program, and a second one, fetching results from the final instructions and delivering

[2]Functions δ and ρ correspond to the decomposition of input data of the `map` skeleton into sets and to the recomposition of sets of results into the final result of the `map` skeleton.

them onto the program output stream. The execution of a skeleton program compiled into macro data flow code proceeds with interpreter threads fetching fireable instructions from the task pool threads and delivering results to other instructions stored in the task pool.

We enhance this simple interpreter with some features exploiting the knowledge derived from skeleton structure of the program, in order to achieve better efficiency during skeleton program execution.

Data delivered as the result of the evaluation of an instruction, that may be reused in the evaluation of other instructions, is marked as `reusable`. As an example, all the rows of the second matrix in input to the `map` skeleton of our sample program will be marked as reusable, as they will be used to compute different items of the final matrix result. Each interpreter stores received data marked reusable in a local cache handled by the memory manager thread. Further instructions referring to the cached data are scheduled with the name of the cached data in place of the input parameters. This mechanism requires some kind of centralized knowledge in the task pool managers to understand where pointers can be inserted in the instructions instead of real data. The mechanisms we want to exploit to this purpose, make use of "affinity scheduling" techniques, such as those used in SMP to enhance thread scheduling taking caches into account [14, 9]. Furthermore, reusable data known to be used by a large number of data flow instructions, is broadcast to all the interpreters and stored in local caches before actually starting the scheduling of macro data flow instructions processing the data, in all those cases where broadcast communications are known to be more efficient than collections of point-to-point communications. Then, instructions using this data are scheduled to the interpreter processes with references in place of data.

The interpreter performance is also enhanced by a proper management of local task pools. Each local task pool has a low and a high water mark, representing the minimum and the maximum (suitable) number of data flow instructions that is to be stored in the local task pool. When the number of instructions falls below the low mark, new instructions are required to the other task pool threads. When the number of instructions exceeds the high mark, requests from other nodes are considered and consequently local instructions are delivered to other task pool threads. This mechanism allows dynamic load balancing to be achieved during program execution.

We organized things in such a way that instructions relative to the same input task are preferably kept local, even in case they become candidate for moving to other task pool threads. As a consequence, stream parallel computations relative to the single input data task happen to be performed onto the same PE. This allows communication overheads to be minimized.

Instructions deriving from data parallel `map` skeletons are scheduled to task pools by exploiting techniques derived from autoscheduling [11] in such a way that load balancing *and* communication optimization is achieved.

Last but not least, in our system we do not store in the task pools the complete data flow graph relative to the computation of any input data task (i.e. we do not store graphs such as the one of Figure 1, with fireable and non-fireable instructions). The first instruction only (the one fireable with the data coming from the input stream) is initially stored in the task pool. Then, interpreter threads know exactly the "continuation" of each data flow

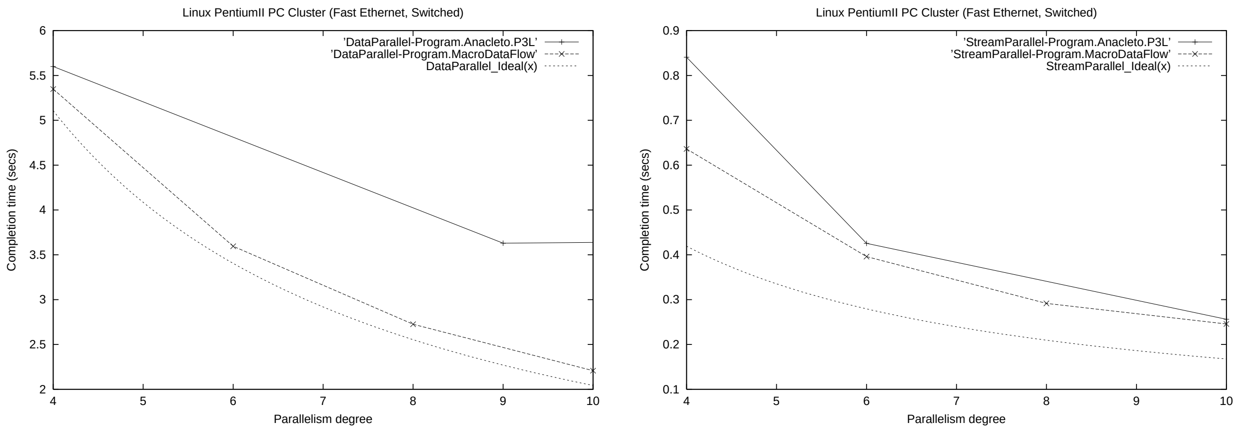

Figure 2: Performance results on a PC cluster

instruction[3] and each time they compute a data flow instruction they send to the task pool thread not only the result data but the result data embedded in the data flow instruction that will consume that data[4].

4 Experimental results

We evaluated our new implementation model for skeletons by "hand" compiling some sample programs and evaluating their performance. We considered simple "synthetic" programs instead of real applications, which are easy to be compiled according to our model, but having parameters that allowed us to measure different features of interest. We used P3L source code for the experiments, in such a way that the results achieved could be compared with the results achieved by using Anacleto [5], a prototype P3L compiler running on Linux PC clusters. Both the hand compiled code and the Anacleto object code were C programs with MPI calls [10]. We performed the tests on a cluster of Linux PCs (10 PentiumII PCs with a dedicated, switched Fast Ethernet interconnection network) and on a Cray T3E.

We discuss here the results of three different experiments. The first experiment was aimed at assessing the effectiveness of the reusable data caching mechanism. We wrote a program with a single data parallel skeleton inside. The `map` skeleton divided input data items into data sets in such a way that some of the input data were present only in a single data set (scattered data), some in more than one data set (multicast data) and some other data is simply included in all the data sets (broadcast data). The reusable data caching mechanism demonstrated its effectiveness on our PC cluster, showing good speedup values, and also showed better absolute performance than Anacleto in all those cases where the amount of reusable data is significant (Figure 2 left).

The second experiment was aimed at evaluating the effectiveness of the low/high water marks in conjunction with the continuation based generation of data flow instructions. We run a plain stream parallel program (a pipeline with four stages: two sequential stages, the

[3]such information can be easily derived (compiled) from the skeleton source code

[4]when evaluating instructions relative to a `map` skeleton, however, both the set of instructions computing the partial results and the instruction used to gather these partial results into the final `map` result are delivered to the task pool, in such a way that the partial results can be stored in the proper place.

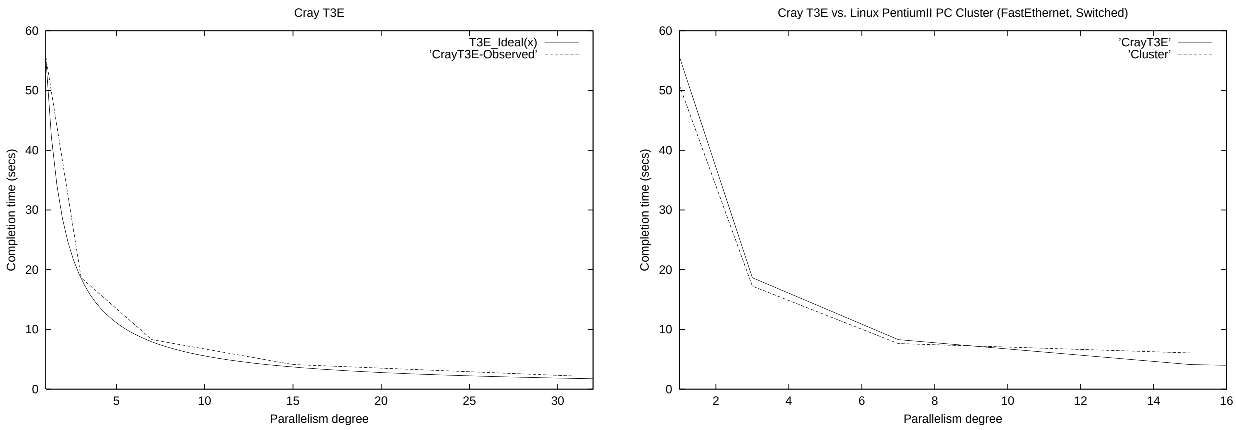

Figure 3: Performance results on a Cray T3E

first and the fourth one, and two `farm` stages, the second and the third one). Again the performance measures turned out to be good and slightly better that the ones obtained by using Anacleto (Figure 2 right).

The third experiment was aimed at assessing the scalability of our code on massively parallel architectures with a communication support much better than the support provided by the PC cluster. Therefore we run skeleton programs compiled according to our model on a Cray T3E. The typical results achieved are shown in Figure 3 (left): the performance achieved is very close to the ideal one. Although for small machine configurations the PC cluster demonstrated to be more efficient[5] , as communication to computation ratio grows the Cray T3E performs better than the PC cluster, as expected (Figure 3 right). In the right plot of Figure 3 the parallelism degree of the cluster runs exceeds the number of machines in the cluster. Our code is designed to exploit a limited amount of node multiprogramming, by having a couple of interpreters competing for the usage of the same Linux machine.

5 Related work & conclusions

Recently, Serot et al. presented a coordination language based on skeletons for real-time image processing applications [13]. In their SKiPPER environment data flow graphs are generated that are subsequently submitted to a specialized tool scheduling fireable instructions for execution in such a way that real time constraints are satisfied. Although the approach is quite different, and despite the fact they claim that their environment is aimed at targeting special purpose applications, there are many contact points with our work. The work by Kasahara et al. [16, 8] is also interesting, although it addressed compilation of FORTRAN code, because they also use a macro data flow execution model. Most of the works on skeleton systems describe very interesting and efficient optimization techniques for skeleton programs, that must be obviously taken into account in our work [1, 15].

Starting from these known results and exploiting our past experience in the design of the P3L compiling tools, we designed a new implementation model for skeleton based parallel languages. The model is based on the data flow execution model, but we extend the data flow interpreter to better exploit, at run time, the locality proper of skeleton parallel com-

[5] cluster sequential execution times are better than those of Cray T3E

putations. Overall the model presents some benefits with respect to the classical template based implementation models for skeletons: a better resource (PE) usage due to the reduced number of choices performed statically, better load balancing even in case of irregular computations, due to the dynamic run time support implemented, easier introduction of new skeletons in the language, as new skeletons should not require modifications to the macro data flow interpreter, and possibility of introducing shared memory features in the skeletons, exploiting the memory handler thread.

We also showed some preliminary experimental results that demonstrate that the approach is feasible. We are currently working to the design and implementation of a complete P3L compiler according to our model.

References

[1] P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Co-ordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Europar '96*, pages 601–614. Springer-Verlag, 1996.

[2] 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(3):225–255, May 1995.

[3] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. Parallel Computing, to appear, 1999.

[4] George Horatiu Botorog and Herbert Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196(1–2):71–107, April 1998.

[5] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based P3L compiler. In *Proceedings of the PCW'97*, 1997. Camberra, Australia.

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

[7] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel Functional Programming with Skeletons: the OCAMLP3L experiment. In *ACM Sigplan Workshop on ML*, pages 31–39, 1998.

[8] H. Kasahara, H. Honda, M. Iwata, and M. Hirota. A compilation scheme for macro-dataflow computation on hierarchical multiprocessor systems. In *Proceedings of the International Conference on Parallel Processing*, pages II–294–II–295, 1990.

[9] K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam. Design of Cache Memories for Multi-Threaded Dataflow Architecture. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 253–264, 1995.

[10] Argonne National Laboratory. *MPICH home page*. `http://www-unix.mcs.anl.gov/mpi/mpich/`, 1999.

[11] D. S. Nikolopoulos, E. D. Polychronopoulos, and T. S. Papatheodorou. Enhancing the Performance of Autoscheduling in Distributed Shared Memory Multiprocessors. In D. Pritchard and J. Reeve, editors, *Proceedings of Europar '98*. Springer Verlag, 1998. LNCS No. 1470.

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

[13] J. Serot, D. Ginhac, and J.P. Derutin. SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications. In *Proceedings of the 5th International Parallel Computing Technologies Conference (PaCT-99)*, September 1999. to appear.

[14] Srikant Subramaniam and Derek Eager. Affinity scheduling of unbalanced workloads. In *Supercomputing '94*. IEEE Computer Society, 1994.

[15] Mario Südholt, Christian Piepenbrock, Klaus Obermayer, and Peter Pepper. Solving large systems of differential equations using covers and skeletons. In *Proceedings of the 50th Working Conference on Algorithmic Languages and Calculi*. Chapman & Hall, February 1997.

[16] A. Yoshida, K. Koshizuka, and H. Kasahara. Data-Localization for FORTRAN Macro-Dataflow Computation Using Partial Static Task Assignment. In *Proceedings of the 1996 international conference on Supercomputing*, page 61, 1996.