ANACLETO: a template-based p3l compiler

S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi and S. Pelagatti
ciarpagl, marcod, folchi, manconi, susanna@di.unipi.it

Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa -Italy

Abstract

This paper gives a brief overview of p3l, a parallel coordination language, and describes the structure of a template based p3l compiler. The compiler produces optimized parallel code for a Fujitsu AP-1000 running MPI (ANU/CAP version).

1 Introduction

p3l is a parallel coordination language based on skeletons [3, 5, 6, 10]. Parallel models based on skeletons belong to the broader class of restricted models, in which the parallel structure of a computation belongs to of a certain limited set of forms.

Restricted parallel models have gained a growing interest in the past years for researchers interested in developing high level parallel programming systems. Developing such systems requires the automatic solution of problems such as mapping, scheduling, data distribution, load balancing, etc.

These problems are intractable in the case parallel models allowing any arbitrary parallel computation structure to be expressed. On the contrary, they can be made tractable when limiting the set of feasible parallel program structures [10, 12].

Among restricted models, skeleton-based models provide the programmer with a set of predefined forms (the *skeletons*) which capture common structural components of parallel applications. Processor farms, pipelines and process arrays are typical skeletons. Applications consist of a set of non-parallel modules, expressed in a *host* non-parallel language and coordinated using a suitable combination of the skeletons provided.

p3l uses C as the host language to express the sequential parts of an application and provides a small set of skeletons as primitive constructs in the language. Skeletons model both task and data parallelism and can be freely nested to define complex interaction patterns.

The main problem in skeleton based languages is devising an efficient implementation of programs composed from several skeleton instances. The parallel structure defined by skeletons needs to be translated into a well structured process graph. Then, the process graph needs to be mapped efficiently onto the target machine, balancing the load of the nodes and minimizing communication delays.

In the paper, we describe how this is achieved in the ANACLETO p3l compiler using *implementation templates*. An implementation template is a parametric network of processes recording a good strategy for implementing a given skeleton on a target architecture. Implementation templates are developed once and for all by the system implementor when porting p3l on a new machine. Each template comes with an analytic performance model predicting the performance of the process network on the basis of a small set of parameters depending on the target architecture and on the features of the user-defined sequential code.

The p3l compiler consults the template library and instantiates the templates by “filling” the parametric parts according to the user code. Template instances are assembled together according to the skeleton nesting specified by the user. Performance models are consulted to decide optimizations on the final process graph and resource allocation.

In the following, we briefly introduce p3l and, then, we describe ANACLETO which produces code for a Fujitsu AP-1000 running MPI (ANU/CAP version [11]). ANACLETO is still being tested and completed. The paper discusses some preliminary results.

2 Skeletal programming in p3l

p3l is built around a host sequential language (currently C) from which it borrows some data types and in which the sequential fragments to be "plugged in" the skeletons are expressed [2, 4]. Skeletons included in p3l model both *task parallelism* and *data parallelism*. Task parallel skeletons are: the **farm** skeleton defining a pool of identical worker modules and the **pipeline** skeleton defining the application of a cascade of functions to a stream of incoming tasks. Data parallel skeletons are: the **map** skeleton modeling independent data parallel computations over arrays; the **reduce** skeleton, modeling the reduction of the elements of an array using a binary associative and commutative operator; and the **comp** skeleton modeling plain functional composition.

The skeleton set is completed by two "control" skeletons: the **sequential** skeleton, which encapsulates sequential portions of C code implementing the non-parallel parts of the application and the **loop** skeleton, which allows the iteration of both task and and data parallel skeletons arbitrarily composed.

A simple example of a p3l program computing the Mandelbrot set on a square of the complex plane is shown in Fig 1. Here, each row of the output picture is computed in parallel. The program is a three-stage pipeline. The first stage `start` and the last stage `stop` are sequential stages not shown in the picture. `start` generates a stream of rows to be computed. Each row is identified by the initial/final complex point (`init`/`end`), by its progressive number (`k`) and by the `resolution` to be applied. The computation of the stream of rows is done in parallel using a pool of farm workers (`mandelbrot`). Each worker (`pixel`) computes a row using a fragment of plain C code and produces the color of each pixel in the integer array in output (`color`). The last stage `stop` displays the rows on the screen as soon as they are produced.

Each skeleton instance is seen as a parallel module which reads an input stream, performs some computation and produces a result. The data composing one element of the stream are defined by an `in()` list and the produced results are defined by an `out()` list. The computation can be either the call to a fragment of the host code (encapsulated between `${...}$` in a sequential module), or the call to one or more p3l skeletons. The farm `mandelbrot` calls `pixel` to perform the computation on a single stream element. The pipe `main` defines the starting point of the whole program and calls the sequence of stages to be executed. Temporary variables between a stage and another are declared "in line" in the pipeline `out` lists.

3 Template-based compilation

p3l is compiled using a library of implementation templates. An implementation template implements a skeleton on a target architecture and is composed of two parts: a *parametric process graph* and a *performance model*. The process graph encapsulates strategies for data decomposition, scheduling, mapping, etc., which have been optimized for the corresponding pair skeleton/target architecture. The process graph is parametric both with respect to the user defined code (e.g. input and output types, sequential C fragments) and with respect to the resources used (e.g. number of worker process activated in a farm implementation). The performance model predicts the performance of an instance of the process graph. A single skeleton can have several templates, each recording a different "good" implementation strategy suitable for a class of architectures and/or for a given subset of skeleton instances [10, 2]. It is up to the compiler devising which are the best templates to be used to implement the skeleton instances of a given application.

Given a p3l program, the compiler is responsible of the instantiation and combination of the implementation templates in the library. The aim is optimizing performance figures and resource utilization of the final process graph to be run on the target. In the following section,

```
seq pixel in (float init[2], float end[2], int k, int resolution)
          out(int color[400])
  ${ /* C fragment computing the Mandelbrot set on a row */ }$
end seq

farm mandelbrot in (float init[2], float end[2], int k, int resolution)
                out(int color [400])
  pixel in(init, end, k, resolution) out(color)
end farm

pipe main in() out()
  start in() out(float init[2], float end[2], int k, int res)
  mandelbrot in(init, end, k, res) out(int color[400])
  stop in(color) out()
end pipe
```

Figure 1: A p3l program defining a three-stage pipeline.

we describe the implementation templates developed for the Fujitsu AP-1000. Then, section 5 details the global compilation strategy.

4 Templates on AP-1000

A preliminary work to be done when deriving templates for a new target architecture is defining a simple abstract model for the architecture. The model should provide a set of operations which is expressive enough to make easy the definition of typical interaction structures in the support. Each operation should be efficiently implementable on the machine and have a stable behavior which can be modeled by analytic cost functions. Operation costs are needed to be able to develop sensible performance models for the templates.

In our case, all the requirements are met using a simple abstract model in which the machine is seen as a set of nodes completely interconnected, and processes interact using a small subset of MPI [8, 11]. MPI primitives are easy to use and their implementation is optimized against the hardware facilities of the AP-1000. Moreover, when the network usage is below the saturation point both point-to-point and collective operations present a stable behaviour and their costs can be modeled using simple functions. For instance, a point-to-point send can be modeled

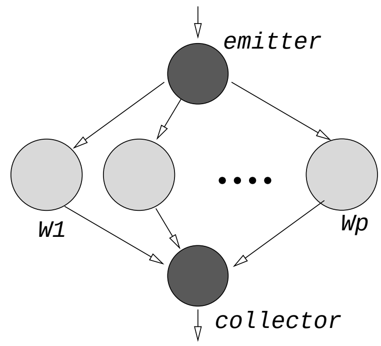

Figure 2: A farm template process graph.

using a a fixed startup cost t_0 to set-up the communication and a transmission cost t_1 for each byte transmitted. Thus, sending a message of d bytes costs $t_0 + dt_1$ independently of the two nodes involved. The same holds for collective operations such as MPI_bcast, with different values for t_0 and t_1.

Using this subset of MPI and these costs, we have developed templates for p3l skeletons. The templates developed so far show a communication load far below the saturation point and the same holds for their combination. We describe a template for the farm skeleton to exemplify the issues involved.

Consider an instance of the farm skeleton with sequential workers. A simple strategy to implement this instance in parallel is to use the process graph of of Figure 2 in which the *emitter*

```
<Data Structures/MPI initialization>
switch(my_rank){
case 0  :{ /* emitter */
           <emitter code>
         }
case p+1:{ /* collector*/
           <collector code>
         }
default :{ /* worker */
           ....
         }
}
< MPI finalize/ termination>
```

Figure 3: Global structure of a SPMD program for a farm template instance.

process accepts the incoming tasks and schedules them to the workers, the *collector* manages the delivery of task results and processes *W1* to *Wp* are *worker* processes performing actual task computation. We can express this structure using a SPMD program whose structure is sketched in Fig. 3. Here, after an initialization phase each activated process specializes according to its rank.

The same template structure can be used when each worker is in turn parallel. In this case, given a set of processes $p_1, \ldots, p_k$ implementing a single worker, we can replicate this subnet p times, provided that the subnet has an input node that interacts with the emitter and an output node that interacts with the collector.

The behavior of this template can be described very simply in an analytic way. For instance, if we have N tasks to be computed in the farm input stream, the completion time of the whole stream can be expressed as

$$T(N) = L + (N-1) * T_s$$

where L is the time taken to output the results of the first task (*latency*) and T_s is the average time taken to compute a task at the steady state (*service time*). In turn, L and T_s can be expressed as

$$L = l_e + L_w + l_c \quad T_s = \max\{t_e, T_w/p, t_c\}$$

where p is the number of workers, l_e, L_w, l_c are the latency of emitter, worker and collector respectively, and t_e, T_w, t_c are the corresponding service times.

The compiler can predict the behavior of a template instance using these formulae along with the communication costs on the target machine and an estimation of the cost of the user defined code. The estimation of the user code is derived by profiling it automatically on a sample input set provided by the user. This prediction can be used to fix the best number of workers for a given template instance or the best granularity of task computation.

In order to use a template within the compiler, both the process graph and the performance model need to be recorder in some way. Performance models are recorded as simple functions in a *model library*.

On the other hand, the structure of the process graph is recorded in "chunks" that can be used to build different template instances depending on the input/output types specified by the user, on the structure of the nested worker modules and on the number of workers included.

The structure of "control" processes, such as the *emitter* and the *collector*, as well as the structure of the code needed to handle data streams by means of MPI primitives are recorded in files (the *process templates*). Parametric parts such as the type of the tasks to be processed and the ranks of the worker processes are left unspecified and will be filled by the compiler on the basis of global program structure.

The interconnection between the control processes, the nested submodules and the template input/output channels is recorded in "wiring" procedures used by the compiler to generate the final process graph. The generation of the final process graph proceeds starting with the inner (sequential) instances to the outer modules. In the farm case, it produces the implementation of the worker first. Then, the net implementing the worker is replicated p times and wired to the appropriate emitter and collector instances.

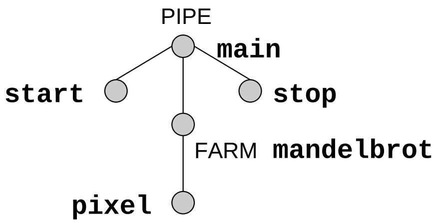

Figure 4: Construct tree for Mandelbrot.

5 Compiler structure

The result of the compilation of a p3l program is a single SPMD program (written in C plus MPI), which encodes the final process graph. The final program has a structure similar to the one shown in Figure 3 in which after a standard initialization phase each process is specialized according to its rank and executes one process of the final process graph.

The translation process is organized in the following steps:

- first, the *front-end* parses the source code, performs type checking and produces the internal representation of the program, the *construct tree*. The construct tree includes a node for each skeleton instance in the program. The root is the `main` skeleton and the children of each node are the called skeletons. The construct tree of the Mandelbrot program in Figure 1 is shown in Figure 4, where the sequential skeleton instances are the leaf nodes.

- then, there is a *template/resource tuning phase* in which the construct tree is traversed and modified/annotated in order to assign a template to each skeleton and to optimize the resources devoted to each template taking the global structure into account. This can include a *profiling* phase in which the user code specified by the sequential modules is profiled on a sample set of inputs to derive the costs to be inserted in the performance models. The profiling is usually done on demand when the sequential part changes and recorded in files for subsequent compilations.

- finally, there is a *code generation phase*. Starting from the construct tree annotated with the optimal resource assignment and with the templates chosen, the final SPMD C+MPI program is produced. This requires the computation of the rank of all the processes to be included, the generation of the specialized code for both the sequential and the control processes of each template instance, and the computation of the exact process ranks for the partners of point-to-point communications. For instance, the Mandelbrot program can be compiled in a process graph with the structure shown in Figure 5. This graph be coded in a C+MPI fragment with the same structure of Figure 3. In this case, we have two more processes for the `start` and `stop` stages and the number of workers is fixed by the compiler. In the code generation phase, we also generate scripts and makefiles to compile the program using the right options and to execute it properly.

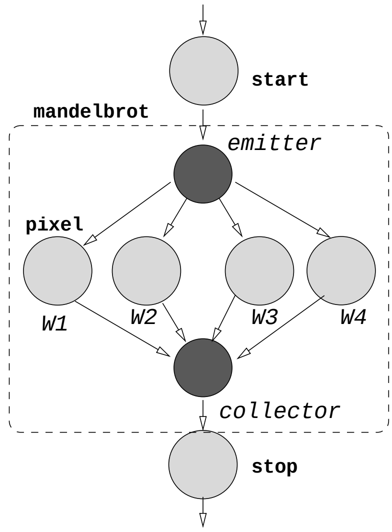

Figure 5: Process graph for Mandelbrot.

At the moment, ANACLETO is still being tested and completed. It currently accepts all the p3l parallel skeletons and optimizes the task parallel ones automatically while for data parallel skeletons some compilation steps are still to be performed by hand. Moreover, most skeletons have a single implementation template.

The optimizations implemented in the template/resource tuning phase include pipe balancing, optimization of the communication grain and computation of the optimal number of workers for farm instances. Pipe balancing ensures that resources devoted to each stage are tuned in order to have the maximum bandwidth achievable on the whole pipeline and the best balance between the speed of different stages. This allows resource saving on non-bottleneck stages. Farm worker tuning ensures that the number of workers balances the speed of the emitter and the collector. Finally, grain optimization computes the best number of tasks delivered within a single communication step between different processes in order to achieve a good balance between communication and computation in the whole program. Details on the optimizations applied for data parallel skeletons can be found in [4].

6 Conclusion

While developing ANACLETO the Fujitsu AP-1000, we have been able to obtain very interesting results. First of all, we have checked that very simple analytic performance models can be used to predict the template behaviour. This is due to the fact the AP-1000, routing network and algorithms show a very regular behavior when the load is below the network saturation point, which is always the case with the templates we have developed. The models derived are rather similar to the ones developed for Transputer networks [10], but much simpler due to the presence of routing co-processors and to the complete interconnection network abstracted by MPI primitives.

After that, implementation templates have been derived for the skeletons in p3l. The scalability of the templates and the level of prediction ensured by the performance models have been evaluated and they are rather high.

In the future, we plan to complete ANACLETO automating the parts that are still manual and testing it against much larger applications starting from the application set we used for the previous p3l compiler [1]. We also plan to port the back-end of the compiler on other parallel platforms running MPI.

Further research topics we intend to investigate are: the extension of p3l with more dynamic data structures (only static n-dimensional arrays are currently supported); the study of the problems involved in the combination of the task parallel subset of p3l skeletons with other data parallel standard languages (such as HPF [7]); and the study of the impact of primitives for shared memory (such as those present in MPI-2 [9]) for template development.

Further information on the whole p3l project (and eventually, the experimental ANACLETO version) is available on WWW at

`http://www/di.unipi.it/~susanna/p3l.html`

References

[1] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An environment for structured parallel programming. In L. Grandinetti, M. Kowalick, and M. Vaitersic, editors, *Advances in High Performance Computing*, pages 219–234. Kluwier, Dordrecht, The Netherlands, 1997.

[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] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, Mass., 1989.

[4] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in p3l. In *Proc. of EURO-PAR '97, Passau, Germany*, August 1997. To be published by Springer, Berlin on LNCS series.

[5] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, R.L. While, and Q. Wu. Parallel programming using skeleton functions. In A. Bode M. Reeve and G. Wolf, editors, *Proc. of PARLE'93*, volume 694 of *LNCS*, pages 146–160, Berlin, 1993. Springer.

[6] J. Darlington, Y.K. Guo, H. W. To, and Y. Jing. Skeletons for structured parallel composition. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[7] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1), June 1993. Also available as Technical Report Computer Science Dept., Rice University, Houston Texas, 1993.

[8] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994. Also available as Technical Report CS-94-230, Computer Science Dept., University of Tennessee, Knoxville, TN, 1994.

[9] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. Draft, presented at Supercomputing '96, Computer Science Dept., University of Tennessee, Knoxville, TN, November 1996.

[10] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, TD-11/93, Department of Computer Science, University of Pisa (Italy), March 1993.

[11] D. Sitsky, D. Walsh, and C. Johnson. Implementation and performance of the MPI Message Passing Interface on the Fujitsu AP1000 multiprocessor. Technical report, Department of Computer Science, Australian National University, Canberra,Australia, 1994.

[12] D. B. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2):133–158, April 1991.