Skeletons for Data Parallelism in p3l

Marco Danelutto, Fabrizio Pasqualetti, Susanna Pelagatti *

Dipartimento di Informatica, Corso Italia, 40 PISA Italy

Abstract. This paper addresses the application of a skeleton/template compiling strategy to structured data parallel computations. In particular, we discuss how data parallelism is expressed and implemented in p3l, a structured parallel language based on skeletons. In the paper, we describe the new set of p3l data parallel skeletons, outline the implementation templates used to compile data parallel computations, and discuss the template based compiling process and the optimizations that can be carried on. Finally, we give some preliminary implementation results.

1 Introduction

In the past years, a growing interest has arisen for *restricted* parallel computational models, that is models in which parallel computations must be of certain limited forms [12]. Restricted models are appealing as most of the problems to be solved when implementing a parallel application, such as it is efficient, are intractable if the application is of arbitrary form [4, 11]. Encouraging results have been derived for restricted models, and in particular for skeleton based models, in which parallel computation must be expressed as instances of a fixed set of patterns (the *skeletons*) possibly nested [7, 4, 11].

In our previous work, we proposed a structured parallel language, p3l, based on skeletons [5, 1] and we discussed an implementation strategy able to take advantage of the limitations imposed on the parallel program structure. This strategy is based on a *library of implementation templates*. An implementation template is a parameterized network of processes implementing all the instances of a given skeleton onto a given target parallel architecture. The network is associated to an analytic performance model able to predict the performance of a network instance on the basis of user defined code and architectural costs. The implementation of a p3l program is built by the compiler by composing templates in the library according to the skeleton nesting specified by the user. Performance models are consulted to assign resources to each template instance and to tune program structure in an optimal way.

p3l includes *task parallel* skeletons and *data parallel* skeletons. Task parallel skeletons abstract common forms of parallelism used in the computation of unrelated tasks, such as pipeline computations or task farm computations. Data

* We would like to thank the IFPC at the Imperial College in London for the access to the Fujitsu AP1000 machine and the assistance given. Fabrizio Pasqualetti was supported by a grant from the CPR Pisa. The research has been partially supported by the Italian MURST.

parallel skeletons abstract common patterns of parallelism used when computing in parallel parts of the same task. In [5, 1], we focused on task parallel skeletons and proposed some simple, non nestable data parallel skeletons (`map`, `reduce` and `geometric`). Moreover, the `geometric` skeleton modeled data parallel computations involving arbitrary data exchanges between sub tasks. It was a sort of 'escape' skeleton, in which a programmer could express in a non restricted way data parallel computations not expressible by `map` and `reduce`. This lead to problems designing an optimized template for this construct as it incurred the same intractable problems encountered for efficient implementation of non-restricted, universal, models.

In this paper, we describe the result of our work in redesigning the skeletons, the implementation templates and the compiler strategies dealing with the data parallel part of p3l.

2 Data parallel p3l skeletons

p3l is a structured parallel language in that it provides the programmer with a set of primitive skeletons (the *parallel constructs*) that can be nested to build complex parallel structures [5, 1]. p3l is built around a host sequential programming language (the **host language**) from which it borrows concrete types implementing the p3l data types, identifiers and "functions" which are sequential portions of code describing a function f to be computed on a single input datum by an instance of a sequential construct. Currently, p3l uses C as the host language [1]. Skeletons included in p3l are the following: the **sequential skeleton** defining a sequential module computing a function f on a stream of input values; the **farm skeleton** defining a pool of worker modules, each one able to compute a given function f; the **pipeline skeleton** defining a parallel module composed of a sequence of stages in cascade; the **loop skeleton** allowing the execution of a p3l module to be iterated and the **data parallel skeletons**.

Data parallel skeletons abstract typical patterns of data parallel computation. They identify a small set of data parallel patterns that can be composed to build more complex structures in the spirit of the skeleton methodology. There are three skeletons currently available: the `map` skeleton modeling independent data parallel computation over arrays, the `reduce` skeleton exploiting parallelism in the reduction of array elements using an associative and commutative binary operator and the `comp` skeleton allowing different independent phases of a data parallel computation to be composed. All three skeletons can be freely nested.

The **map skeleton** models independent data parallel computations on array elements. It defines a set of "virtual processors" organized as an n dimensional array and models a data parallel computation in three phases: distribution and partition of data to the virtual processors (*distribution phase*), independent virtual processor computation (*computation phase*) and collection of results (*collection phase*). A generic map instance is of the form

```
map mod-name in(in-list) out(out-list)
  body in(body-in-list) out(body-out-list)
end map
```

where `map` and `end map` are p3l delimiters, `mod-name` is the name of the map instance, `in-list` (`out-list`) defines types and the names of the input (output) data, `body` is the name of the module defining the computation to be performed by each virtual processor, `body-in-list` defines how an input datum is partitioned among the virtual processors, and `body-out-list` defines how the virtual processor results are collected to build the array(s) in output. As an example, consider the following map instance

```
map foo in(int a[n][m], int b[n], int c[n][m]) out(int d[n][m])
  body in(a[*i][*j], b[], c[][*j], *j) out(d[*i][*j])
end map
```

Variables prefixed by `*` (`*i` and `*j`) are called *free variables* and define the virtual processors and the data partition. Virtual processors are defined by the free variables in the body output list, in the example above we have $n \times m$ virtual processors. Virtual processor (i, j) computes the array element `d[i][j]`.

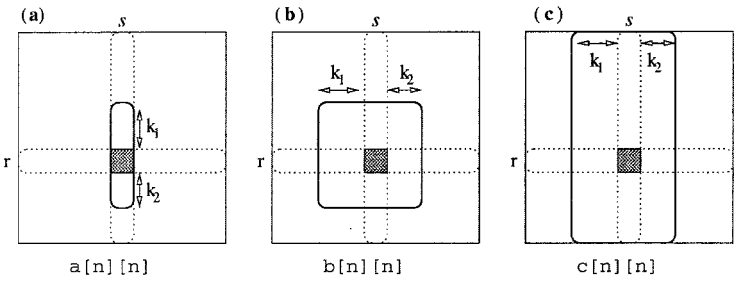

Fig. 1. Different overlapping multicast: a column slice of `a` (a); a square slice of `b` (b) and a group of columns of `c` (c).

Distribution operations are defined by the position of the free variables in the body in-list. In practice, if a free variable `*j` appears within an array reference, all the virtual processors for which `*j` has value k will receive the array element for which `*j` has value k. For instance, in the example above `a[*i][*j]` scatters the array `a` distributing `a[i][j]` to virtual processor (i, j); `b` is broadcasted to all the virtual processors as no free variable appears in it; and, `c[][*j]` specifies a multicast operation where all the virtual processors $(1, i) \ldots (n, i)$ get the ith column of `c` (`c[][i]`). More complex multicast operations can be expressed in a `map` instance by using constant expressions involving free variables. For instance, `a[*i-k1 .. *i+k2][*j]` specifies that each virtual processor (r, s) gets `a[r-k1][s]`...`a[r+k2][s]` (Fig. 1a); `b[*i-k1 .. *i+k2][*j-k1 .. *j+k2]` specifies for each (r, s) a rectangular slice centred on `a[r][s]` (Fig. 1b); and, `c[][*j-k1 .. *j+k2]` specifies a slice of $k_1 + k_2 + 1$ columns (Fig. 1c).

The **reduce skeleton** models the "reduction" of an array by means of a binary associative and commutative operator. A reduce instance specifies the dimensions of the array to be reduced and the p3l module specifying the binary associative operator to be applied. For example, the following reduce instance

```
logic_or in(bool a,b) out(bool c) ${ c = a || b; }$ end

step in(bool sten[3][3], int i, int j) out(bool z, changed) ${ ...}$ end

map life_step in(bool a[n][n]) out(bool b[n][n], bool ch[n][n])
  step in(a[*i-1..*i+1][*j-1..*j+1], *i, *j) out(b[*i][*j],ch[*i][*j])
end map

reduce is_life_moving in(bool ch[n][n]) out(bool cond)
  logic_or in(ch[*][*]) out(cond)
end reduce

comp game_step in(bool a[n][n]) out(bool b[n][n], bool cond)
  life_step in(a) out(b, bool c[n][n])
  is_life_moving in(c) out(cond)
end comp

loop game_of_life in(bool a[n][n]) out(bool b[n][n]) feedback(a=b)
  game_step in(a) out(b, bool cond)
until not(cond)
end loop
```

Fig. 2. Definition of a data parallel module computing the game of life.

```
reduce red_f_0 in(int x[10]) out(int y)
  f in(x[*]) out(y)
end reduce
```

specifies the reduction of all the elements in a vector `x` (`x[*]`) by an operator defined by the `f` module.

The **comp skeleton** allows data parallel modules to be composed. A `comp` instance specifies a list of calls to the data parallel phases to be executed in sequence. Among the rest, `comp` can be used to compose a `map` and a `reduce` instance, to model the well known Map&Reduce paradigm of parallelism [2].

3 An example

We illustrate the characteristics of the data parallel part of p3l with a simple example. Consider the problem of computing the classical *game of life* exploiting data parallelism in the update of each pixel. The world is modeled with a matrix of boolean values `a[n][n]`. Each `a[i][j]` is `true` if the corresponding world cell is alive and `false` if it is dead. Each world cell `a[i][j]` is updated according to the value of the neighbor 8 cells until a stable configuration is reached. All cells can be updated in parallel. Figure 2 sketches the corresponding p3l program. `logic_or` is a sequential module implementing the logic `or` operator, `step` is the sequential module implementing the update of a single cell in terms of its neighbors (recorded in the array `sten`). A single parallel update step is defined by the modules `life_step` and `is_life_moving`. Finally, the single update step is iterated using the `game_of_life` instance of the `loop` construct. At the beginning, the world matrix `a` is distributed to all the virtual processors with a multicast

operation giving the 8 neighbors to each one. Then, each virtual processor (i, j) computes the update of the world cell `a[i][j]` and sets `ch[i][j]` to true if a change has occurred in the cell state. The vector `ch` is then reduced with the logic OR operator to understand if at least one of the cells has changed. If no change has occurred, the configuration reached is stable and the computation terminates. This is stated by the `until` clause of the `loop` testing the logical or of all the `ch[i][j]`. In case a change has occurred, the virtual processors cooperate to gather the new neighbor values and a new iteration is computed.

4 Template based compilation of p3l

The p3l compiler is a *template-based* compiler [1, 11]. For each skeleton, an implementation template is defined which implements in a parametric way all the instances of that skeleton. An implementation template is composed of two parts: a *parametric process graph* and a *performance model*. The performance model allows the prediction of the performance of a given process network instance with a given amount of resources. All the templates are stored in libraries, which are consulted by the p3l compiler to build the implementation of a given program as an optimized composition of templates in the library. The compiler extensively uses the performance models related to each template to guide the optimization process and to decide the amount of resources to be assigned to each template. The p3l compiler is organized in three parts: *front-end*, *middle-end* and *back-end*.

The **front end** parses the source code, check types and produces the internal representation of a p3l program (the *construct tree*). The **middle end** processes the information contained in the construct tree. The tree is modified and annotated until a suitable optimized composition of the templates in the libraries is reached. The middle end produces an abstract representation of the final process graph implementing the program on the target architecture (the *abstract process graph*). In particular, the middle end analyzes the data parallel subtrees in order to generate an optimized instance of the data parallel templates. A *data parallel subtree* is a subtree including only instances of data parallel constructs, possibly iterated. The **back end** takes in input the abstract process graph and generates the actual code for the target machine. This is carried out by using canned process templates included in the process template library. More details on the general structure of the p3l compiler can be found in [1]. In the following two sections, we describe an optimized data parallel template (Sec. 4.1) and the algorithm used within the p3l compiler to choose the optimized template instance for a given data parallel subtree (Sec. 4.2). The template is defined according to a simple message passing model, which abstracts a distributed memory MIMD machine with nodes equipped with communication processors (such as commercially available machines like Cray T3D/T3E, Fujitsu AP1000, Meiko CS2).

4.1 A template for data parallel computations

The process network of the data parallel template is composed of p worker processes each one emulating a subset of virtual processors (the *VP-set*). We assume

`broadcast(w0,x)`	broadcasts `x` from worker `w0` to all the others;
`scatter(w0,x,distr)`	scatters `x` from `w0` with scattering strategy `distr`
`multicast(w0,x,distr)`	multicasts `x` from `w0` with multicast strategy `distr`;
`stencil-gather(x, y, stencil)`	gathers some of the elements of `x` in `y` according to the pattern `stencil`;
`gather(x, w0)`	gathers all the results produced in the array `x` in `w0`
`gather-broadcast(x, w0)`	gathers `x` on `w0` and then broadcasts it
`gather-scatter(w0,x,distr)`	gathers `x` on `w0` and scatters it according to `distr`;
`reduce(opn, x, x1, w0)`	reduces `x` using `opn` and puts the result in `x1` on `w0`
`reduce-broadcast(opn,x,x1,w0)`	reduces `x` on `w0` and then broadcasts it
`call proc in(var1) out(var2)`	calls the sequential procedure `proc` on each virtual processor with input `var1` and output `var2`

Table 1. Collective operations in the data parallel template

that there is only one process (the `root` process) interacting with the external environment to get the input stream and to produce the output stream. Processes in the template are able to execute a set of *elementary collective operations* each one implementing a simple step of a data parallel computation. The collective operations available are shown in Table 1.

An **instance** of the data-parallel template is obtained by fixing a number of workers, fixing the VP-set to be emulated by each worker and fixing a sequence of collective elementary operations to be executed. The workers follow an SPMD pattern of computation executing the collective operations in order and terminate at the end of the operation sequence. A p3l data parallel subtree is implemented using a suitable sequence of collective operations chosen by the compiler. The number of workers to be included (as well as the distribution strategy for the virtual processors) is chosen by the compiler using a performance model predicting the overall template performance. For instance, the map `life_step` in Figure 2 can be implemented by an instance of the template with n^2 workers, each one emulating a single virtual processor, and executing the following collective steps

1. a `scatter(a,root,[*i][*j])` operation to distribute each `a[i][j]` to the worker emulating virtual processor (i, j);
2. `stencil-gather(a,a,[-1..+1][-1..+1])` to read the neighbor values;
3. `call step in(a) out(b,c)` to call the code implementing a virtual processor;
4. `gather(b,root)` to gather the results in the `root` node.

The collective operations in Table 1 are quite standard a set of primitives, and similar ones are included in many parallel libraries (such as CVL [3]) and in the emerging `MPI` standard [10]. However, our operations are supplied with analytical performance models that can be combined to obtain the expected performance of a template instance onto a given machine. The cost of a given data parallel

template instance will be then derived combining the costs of the collective operations used in the instance.

In our prototype implementation, we have developed a library of collective operations written in C plus `MPI` on a Fujitsu AP1000 [9]. The complete models can be found in [6]. The cost of a communication among two processes is given by the sum of a startup time t_s accounting for the overhead of starting up a send operation, a receive time t_r accounting for the overhead of receiving a message and a per-byte transmission cost t_1 which is the time required to send one byte once the communication has been started. A performance model for a template instance is derived combining performance models for different template steps and instantiating all the parameters according to the case at hand.

The scheduling strategy of virtual processors to the workers is static and is decided by the compiler on the basis of the estimated variance of worker computation among the usual *block* and *cyclic* strategies [8]. The compiler goal is to optimize the sequence of collective operations and to choose the number p of workers and the distribution in order to optimize the response time of the module and obtain the best resource allocation.

4.2 Compilation of data-parallel subtrees

Each instance of a data parallel construct is compiled using a suitable sequence of operations. Generally, the behaviour of the resulting SPMD program is as follows. First, the `root` process cooperates with the workers to distribute a new input datum to be computed; then, all the processes execute a sequence of collective operations; and, finally, the result of the computation is collected in the `root` process to be output.

The generation of the sequence of collective operations for a given data-parallel subtree goes through three steps: First, a virtual processor allocation strategy is fixed and each module in the data-parallel subtree is expanded in a sequence of collective operations. Then, the resulting global sequence of operations is optimized consulting the operation cost models to minimize communications and data movement. Finally, the performance model is consulted to choose the best number p of workers

The data parallel p3l skeletons translation is rather straightforward. For instance, sequential modules are translated using a suitable `call` operation and map modules are translated in a sequence of distribution collective operations, followed by a sequence of operations implementing the body module and ended by some `gather` operations to collect the arrays of results.

As an example, consider the game of life program in Figure 2. Analyzing the program, we find out that the virtual processor set is a two dimensional array $n \times n$. The computational effort is expected to be uniform, while interaction among neighbor processors is needed, thus a (block, block) distribution of the virtual processors is chosen by the support. The result of the first translation step is the following.

```
/* variable declarations and initialization */
1.  repeat {
2.    scatter(root, a, [*i][*j])          /* map life_step */
3.    stencil_gather (a,a,[*i-1][*i+1])
4.    call Simple_step in(a) out(b,cond)
5.    gather(b, root)
6.    gather(cond, root)
7.    scatter(root, cond, [*i][*j])          /* reduce is_life ...*/
8.    reduce(Logic_or,cond,c,root)
9.    broadcast(root, c)
10.   call Not in(cond) out(ncond)
11.   if (not(ncond)) copy in(b) out(a)
12.   } until (ncond)
/* termination management  */
```

Here, the map `life_step` has been translated in the operations already discussed in the previous section. First, the input matrix `a` is scattered according to the distribution pattern (L2), then the neighbors are collected (L3), the sequential code is called (L4) and the results gathered (L5,L6). Lines L7-L8 implement the reduce module by first scattering the array `cond` to be reduced and then reducing it according to the `Logic_or` operation. The subsequent lines implement the loop control in a distributed way on all the workers. In particular, Line L9 broadcasts the boolean condition to be tested. Then, (line L10) each worker computes the termination condition. Line L11 copies `b` in `a` to start a new iteration in case the termination condition is not met.

This initial translation is rather inefficient, as many data movement are useless. The second step of compilation deals exactly with the optimization of data movement and rearranges the collective operations in order to have a more efficient template execution. Typical program optimizations are: eliminating `gather` and `scatter` of the same data with the same distribution strategy (as they move data around to come back to the initial state); substituting pairs of operations with more efficient combined ones, (for instance, the pair `reduce` and `broadcast` can be substituted with a more efficient `reduce-broadcast` which requires less synchronization) and eliminating unnecessary `copy` operations.

Optimizing our example we obtain the following sequence of operations

```
/* variable declarations and initialization */
  scatter(root, a, [*i][*j])   /* map translation */
  stencil_gather (a,a,[*i-1][*i+1])}
  repeat {
    if (non_first_it){
     stencil_gather (b,a,[*i-1][*i+1])
    }
    call Simple_step in(a) out(b)
    reduce-broadcast(Logic_or,cond,c,controller)
    call Not in(cond) out(ncond)
  } until (ncond)
  gather(b, controller)
/* termination management  */
```

The scatter operations on line L7 and the gather operation on line L6 have been removed as they leave the `cond` array distributed in the same way. Moreover, if we unroll one time the `repeat` loop we see that the array `b` is first gathered (line L5) then copied to `a` (line L11) and then scattered (line L2). Thus, the intermediate loop iterations can be optimized by eliminating the matching scatter and

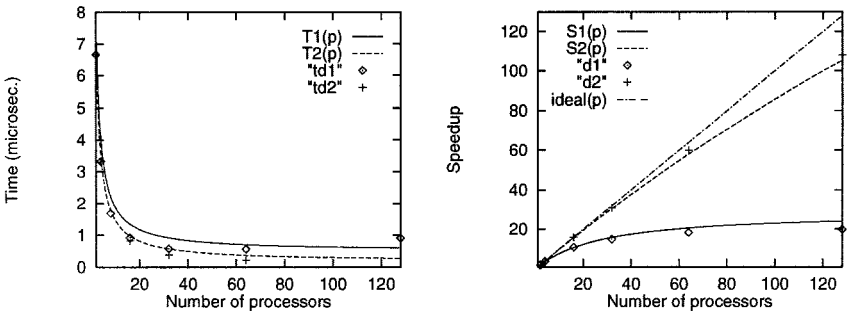

Fig. 3. Completion time of the game of life (right) and speedup (left).

gather and leaving the copy on the local variables. The first scatter of `a` and the corresponding `stencil-gather` is moved outside the loop and is performed *only* in the first iteration. In iterations following the first one the `stencil-gather` is executed reading directly from `b`. Moving also the `stencil-gather` outside the loop, we can eliminate the final copy reading directly from the output vector `b`.

5 Experimental results

In Figure 3, we show the completion time (right) and the speedup (left) achieved by two different versions of the game of life produced using the implementation template and the compilation algorithm in a partially automated way on the Fujitsu AP1000. In both pictures, we consider a world matrix of 256×256 and execute 16 iterations. The solid lines represent the time predicted with the analytical models and the spots represent measured runs on the machine. T1(p) and td1 represent the predicted and measured completion times for the first unoptimized translation of the game of life discussed in Sec. 4.2. T2(p) and td2 are the predicted and measured times of the final optimized version. The corresponding speedups are S1(p) and d1 (corresponding to T1) and S2(p) and d2 (corresponding to T2). From the picture we can see that the performance prediction achieved with the analytical models is very close to the measured time. Moreover, the optimization technique appears rather effective, as the optimized and non-optimized versions discussed in Sec. 4.2 achieve radically different speedup figures.

6 Related Work and conclusions

In the paper, we have discussed a skeleton based approach to the implementation of data parallel computations. The work presented here is related to the research track based on skeletons [4, 7] and to the research track on implementation of data parallel languages [3, 8].

There are two key points that distinguish our approach from the ones in the literature. The first regards the abstraction of the skeletons/construct provided.

p3l data parallel skeletons present a high abstraction regarding data distribution, alignment and virtual processor allocation. For instance in spf [7] and in HPF [8] the programmer is explicitly responsible of data distribution and alignment. On the contrary, a p3l programmer only states abstract dependencies among the parallel activities to be executed and the data present in the program. The number of processes used, the allocation strategy and the data distribution are completely decided by the compiler using the information related to the pattern used. This allows the compiler to make different choices when moving to a different target, making code (and performance) portability easier without a program re-tuning played by the programmer.

The second point regards the combination of the idea of a data parallel support based on library of collective operations, which is largely used in the literature (for instance [3, 10]), with the idea of precise analytic performance models describing the performance of each collective operation. Using the two ideas together, we derive accurate prediction models for our implementations and are able to take sharp optimization decisions. The preliminary results obtained are encouraging as they show high speedup and high levels of performance prediction.

References

1. 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.
2. R. S. Bird. An introduction to the Theory of Lists. In *Logic of programming and calculi of discrete design*, p. 5–42. Springer-Verlag, 1987.
3. G.E. Blelloch, et al. Implementation of a portable nested data-parallel language. *JPDS*, 1(21):4–14, April 1994.
4. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, Mass., 1989.
5. M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8:205–220, August 1992.
6. M. Danelutto, F. Pasqualetti, and S. Pelagatti. Data parallelism in p3l. Draft, Dipartimento di Informatica, Università di Pisa, December 1996.
7. 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 PPoPP*, 1995.
8. High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1), June 1993.
9. H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. CAP-II Architecture. In *First Fujitsu-ANU CAP Workshop*, November 1990.
10. Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.
11. S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, Dep. of Computer Science, Pisa, Mar. 1993.
12. D. B. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2):133–158, April 1991.