Technical Report: TR-96-06, March 27, 1996

Design and evaluation of parallel applications using a structured parallel language

Pasetto,Davide; Vanneschi,Marco

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy

Abstract

Structured parallel programming is one of the possible solutions to exploit Programmability, Portability and Performance in the parallel programming world. Programming using high level parallel constructs permits the programmer to focus on the development of the parallel algorithms rather than on their low level implementation. The power of this approach stands in the possibility of modeling the performance of the high level constructs and building optimizing template-based compilers. These compilers use formulas describing the performance of language constructs over the target architectures and tune their parametric implementation to achieve high performance figures. In this paper, we describe a set of applications developed using such a programming methodology and show the results obtained.

ACM–CR Subject Classification:

Keyword and phrases: parallel programming skeletons languages

Design and evaluation of parallel applications using a structured parallel language

Davide Pasetto Marco Vanneschi

March 27, 1996

Abstract

Structured parallel programming is one of the possible solutions to exploit Programmability, Portability and Performance in the parallel programming world. Programming using high level parallel constructs permits the programmer to focus on the development of the parallel algorithms rather than on their low level implementation. The power of this approach stands in the possibility of modeling the performance of the high level constructs and building optimizing template-based compilers. These compilers use formulas describing the performance of language constructs over the target architectures and tune their parametric implementation to achieve high performance figures. In this paper, we describe a set of applications developed using such a programming methodology and show the results obtained.

1 Introduction

The main problem that makes parallel architectures used for just a limited number of applications is that programming such a machine is very difficult, time consuming and error prone. Moreover the state of the art in parallel languages is still not able to obtain both programmability and performance of the programs. The major differences between existing parallel machine hardware make efficient programming a parallel application in a portable way a task even harder and often impossible. Tools should help programmers write efficient and portable applications, but unfortunately building such tools is very hard if programs are written accordingly to a general parallel computation model in which processes can interact accordingly to any arbitrary communication or synchronization pattern [Ski91]. When dealing with such a general computation model, the main problem is modeling and predicting program performance in such a way to be able to operate on the program and to optimize it. The performance model must be concise and precise at the same time: concise to allow the tool–writer to use it in a real compiling tool, and precise to permit the tool to do the right compiling choices in order to obtain optimal performance figures. The large size of the usual performance models and the NP–hard complexity of the algorithms that operate on them [AP92] make developing an effective tool an almost impossible job.

Between the proposed solutions for this problem, an attractive one is to adopt a restricted parallel computation modem, that is a framework in which the parallel structure of the computation is build by composing a restricted set of basic forms of parallelism. Adopting such methodology allows us to decrease the size and the complexity of the program performance model and to use simpler heuristic algorithm to effectively operate on the program structure. This approach is feasible because most parallel applications exhibit the same parallel structure exploiting the same *kind of parallelism*, such as data–parallel, functional, pipeline and so on [Kun89]. Moreover, having a restricted set of parallel interaction structures, we can devise good implementation templates for each form of parallelism on each different parallel machine. An implementation template is a parametric network of processes able to take into account mapping, load balancing and so on in such a way to obtain a tuned implementation of a single basic form of parallelism on a specific architecture [Col89, DMO+92, DFH+93].

It is also possible to develop concise formulas which describe the performance of these templates for each machine; the formulas can be used to optimize both efficiency and performance of programs [Pel93]. Doing so, we can also achieve parallel program portability ensuring a good performance over a large set of parallel systems. Infact the tool designer has a deep knowledge of each target machine and concentrates on the performance of basic forms of parallelism building the template library used by the compilers. The application programmer, on the other hand, can use an high level language independently from any specific architecture and can concentrate on algorithms rather than on their low–level implementation. This approach is used by just a few recently proposed libraries and languages; examples are Cole and Darlington's skeletons for parallel functional programming [Col89, DFH+93] and the P^3L language and compiling tools developed at the university of Pisa [DMO+92].

In this article we report some experiences gathered in the Pisa Parallel Programming Project during the testing phase of our compiling tools. We focus on parallel application programmability, portability and performance, showing how users can write structured parallel applications very quickly, usually starting from existing sequential code, and obtaining good performance figures over a number of parallel systems.

First of all, in section 2, we define the basic forms of parallelism available in the P^3L language and recognized by our template–based compiler; along with a high level syntax we report operational semantic rules that describe how they work. Then, in section 3, we briefly show how our compiling tools work, describing the compilation process. In section 4 we describe a set of applications we developed in order to test our methodology. All the applications were developed by students, with low or none previous parallel programming experience, as their graduate thesis. They often started from an existing sequential application and spent a few months to develop several different parallel versions. Section 5 reports the results obtained in term of best performance on real machine hardware and predicted performance on other popular existing parallel machines. Section 6 contains some concluding remarks and finally appendix A reports a subset of the formulas internally used by the compiler.

2 High level parallel constructs

In this paper we refer to the P^3L language [DMO+92] developed by the $P4$ group at the University of Pisa. This language is a structured parallel language which provides high level constructs that express commonly used forms of parallelism. The user can build a parallel application instantiating these constructs and composing them to build complex parallel interaction structures. Other structured approaches comprise the one proposed in [DFH+93], which allows only the use of a single high–level skeleton in each application, and [Col89], which permits the user to nest parallel constructs.

Due to the fact that programmers can build their programs specifying the parallel interaction structure trough composition of basic parallel forms, each parallel program can be represented by its **construct tree**, which describes how the available forms of parallelism are instantiated and composed. Each tree node is labeled as a parallel construct, tree leaves are sequential code blocks and tree arcs are labeled with the type of the data that should flow between constructs. The language has a **macro dataflow semantics**: each construct behaves as a stateless pure functional module; this choice permits us to simply transform the construct tree in order to balance the load between the available processing nodes and obtain a process network that implements the algorithm showing a better performance. To transform the program we simply change how the basic forms of parallelism are used. Program input is a **stream of tasks**, eventually composed by a single input task, and the function computed by the program is applied to each element of this stream. The current implementation of the language is limited to static data structures: data exchanged between constructs must be defined statically at compile time so the compiler can compute their size and communication time. We are extending our compiler to allow constructs to exchange dynamically sized structures.

Now we introduce all the available language constructs, along with their operational meaning and a denotation using a functional style to write down program structures in a very compact way. In the functional syntax we use the symbol $\mathcal{S}_i$ in place for any kind of construct.

Sequential The SEQ construct represents the leaves of the construct tree and contains the sequential code blocks of the application. The construct works in a pure functional way applying each input data, which represents a task to be executed, to its sequential body and outputting a results. It is not allowed to keep an internal state. In this document, we denote sequential module i by the symbol S_i.

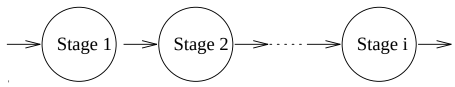

Figure 1: Abstract representation of an i stage pipeline construct.

Pipe construct expresses pipeline parallelism: if an algorithm can be statically divided into a sequence of operations, then we can execute in parallel different operations on different input stream elements. The construct contains i stages and each task of the input stream is passed to the first stage, then the result of the computation performed at this stage is passed to the second one while the first one starts computing over another task, and so on. A pipe construct with n stages is denoted by $\Pi(\mathcal{S}_1, \ldots, \mathcal{S}_n)$.

Farm construct expresses the parallelism based on the independence of activation of pure functional modules on different input data and is denoted by $\mathcal{F}(\mathcal{S})$.

Loop construct is used to model parallel iterative computation of each element of the input stream over a loop body. Each input task is passed to loop body module until the exit condition is satisfied. Then, the result is sent to the next construct. Its denotation is $\mathcal{L}(\mathcal{S}, C)$ where $\mathcal{S}$ is the loop body and C a sequential module that computes the condition value.

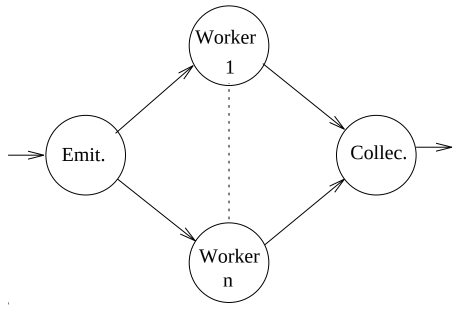

Figure 2: Abstract representation of a farm construct.

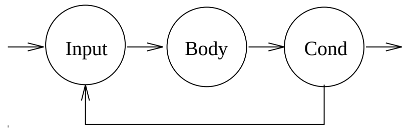

Figure 3: Abstract representation of a loop construct.

Geometric form of parallelism is used when each input task is composed by an n dimensional array and we want to apply a convolution operator on each single array element [For93, OV94]. The input task can also contain data items that should be used in the computation of each new component of the array (these are called **broadcasted data**). The output of this construct is also an n dimensional array. The computation phase consists in a loop of steps; each step begins with a data gathering from a set of neighboring processors (the communication pattern is called the **stencil**) and is followed by a local computation. Actually both the number of iterations and the data access stencil must be known at compile time (we are improving this construct adding the support for a *reduce* operation every k steps to check for the convolution termination instead of statically setting the iteration number).

This construct is denoted by $\mathcal{G}(l, \overline{N}, \overline{S}, S)$ where l is the number of iterations, $\overline{N}$ is the set of input and output array dimensions, $\overline{S}$ is the data access stencil subcube definition and S is the local computation body which is a sequential construct.

Map form of parallelism is a special case of geometric one in which the computation phase consists in a single function applied to each input array element: we do not need a data access stencil and we do not iterate; this can lead to simpler implementation templates, language constructs and analytical models. Construct denotation is $\mathcal{M}(\overline{N}, S)$, $\overline{N}$ and S having the same meaning as in the geometric case.

Reduce construct implements a binary tree-like computation using an associative operator. The input data item is an array and the output data item is a single array element. Construct denotation is $\mathcal{R}(S)$, where S is the module

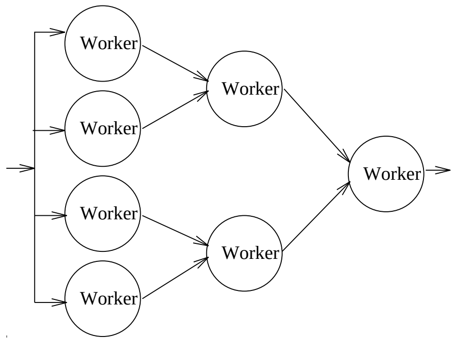

Figure 4: Abstract representation of a reduce construct.

that implements the operator.

3 An optimizing compiler for a structured parallel language

Along with the definition of the language constructs, we developed a set of machine specific implementation templates, an architecture independent analytical model able to predict the performance of the templates on a wide range of architectures and a template-based compiler for the $P3L$ language. Each machine specific template is a process network parametric with respect to the contained construct, the parallelism degree (if applicable) and other internal parameters such as structure decomposition rules, etc. The analytical model uses a set of machine specific parameters, usually obtained using micro–benchmarking techniques, to evaluate the performance of a given template over a specific machine. The model is able to determine template scalability (that is the maximum feasible parallelism degree) and to optimize internal template parameters (such as grain size) given a specific parallelism degree.

The compiler starts from the construct tree, which describes the program, and from a machine specific template library (actually available for T800 mesh-based systems, PVM and Cray T3D), which contains the customizable code and the basic parameters used by our performance model, and produces an executable program highly optimized for the target architecture and that fits into the available resources [BDO+95]. The compilation process proceeds through the following steps:

1. Source program is analyzed and the construct nesting tree is build.

2. An unlimited resources compilation is done: the compiler examines each construct used and selects an implementation template between those available for that basic form of parallelism. The selection is done computing template's maximum theoretical performance and choosing the one that obtains the best values. To compute the maximum theoretical performance we use the template analytical model to analyze the scalability and determine the implementation saturation point after which more resources are not used effectively.

3. The compiler tries to enhance program performance applying a set of rewriting

rules studied to increase the parallelism exploited. All these rewriting rules maintain program semantic but modify its parallel structure in order to exploit more parallelism and obtain better performance. This phase can insert new constructs (for example farming–out a slow module) and can select different implementation templates that optimize specific construct compositions (for example a map followed by a reduce over the same dataset can be implemented using a *compact* template).

4. The compiler balances the load on all the pipelines of the program applying another set of rewriting rules, for example reducing construct performance reducing the number of workers, joining sequential stages, and so on. This phase uses the analytical model to estimate the performance of each rewrite of the program: the system heuristically searches a good solution and at each step tries several rules. To test a rewriting rule the compiler builds the construct tree resulting from the application of the rule and the overall program performance is evaluated. A rule is selected only if the performance vs resource ratio leads to a better solution.

5. At this stage the resulting program is well balanced and gets the best performance the optimizer is able to obtain on the current target system Now the compiler starts removing resources from the program in order to obtain a number of processors small enough to run on a specific target machine hardware. The resource removal process keeps the communication and computation load balanced between PEs and can completely remove redundant parallel construct (such as a farm with a single worker) or collapse them into a single sequential stage (such as a map with one worker). This process uses several heuristics and ends in polynomial time. [BDP94] The analytical model is used to compare the performance of different solutions, in order to reduce resources still having good performance figures.

6. When the number of the nodes required by the program fits into the target system, program executable is produced.

Note that all the algorithms used in the compilation process have a polynomial time complexity. Currently our compiler can produce code for our Meiko CS–1 transputer based system and for a network of heterogeneous workstations running PVM. In the last case no optimization rule is applied due to the fact that it is very hard to predict the performance of a network of multiuser workstation, nevertheless this version is useful in the developing phase of an application. A Cray T3D template library and some of the compiler phases are ready but still a part of the compilation process must be done by hand.

4 Parallel programming in a structured environment

We trust the fact that our approach can achieve both programmability, portability and performance.

To check this we developed a set of applications using undergraduate students, with low or none previous parallel programming experience. The applications we developed[1] are described in the following sections.

[1] You can download the code of all these applications via World Wide Web at P^3L home page: `http://www.di.unipi.it/~susanna/p3l.html`

4.1 Circuit fault list analysis

Circuit fault list analysis packages (CFL) are used by electronic chip producers: when a circuit is printed onto a wafer it is possible that the building process produces defects. Due to the complexity of current chip generation, it can be very time consuming to test all the chip functionalities to search for these defects. To overcome this difficulties producers introduce a set of testing points in the chip itself and, using these points, they search only for the more probable faults [BCGR93]. A possibly small set of test patterns must then be designed, along with the expected output from each pattern, to obtain a fast testing process.

The CFL application is aimed at producing these test patterns: input data represent both the circuit layout and the list of more probable faults the integrated circuit can have due to errors or defects in wafer preparation. Usual approaches involve the use of an automatic test pattern generator (ATPG), which produces a pattern that recognizes one of the faults still to be tested, and a simulator (SIM) that checks whether this pattern also detects other faults. Using this approach a small set of patterns can be found, but the process is very long due to the fact that both the ATPG and the SIM algorithms are very complex and time consuming.

We developed several different parallel version of this application; one of the parallelization strategy selected is to exploit stream parallelism. The structure of the program is the following:

$$\Pi(S_1, \mathcal{L}(\Pi(S_4, \mathcal{F}(\Pi(S_5, S_6))), S_2), S_3)$$

The first module (S_1) partitions the input fault list into a stream of sublists; each sublist enters the loop where it remains until the process generated a number of input sequences able to detect every fault in the list. The pattern generation process starts with a standard sequential automatic test pattern generator (S_5), which computes a pattern for the currently selected fault, and proceeds through a standard sequential simulator (S_6), which checks whether the generated pattern covers also other faults in the current sublist. The other modules handle candidate fault selection (S_4), termination checking (S_2) and fault list output (S_3).

4.2 Radiological image analysis

CAT image analysis software is fundamental to examine radiological images such as those produced by CAT machines. The application gets an image where each pixel shows the energy of a part of the body, and joins zones with similar energies to produce a simpler and understandable image [KMaNMR92]. Several different parallelization strategies have been tested by modifying the parallel structure (i.e. changing construct types and composition) of the starting solution. One of the tested structure is the following:

$$\Pi(S_1, \mathcal{L}(\Pi(\mathcal{M}([X,Y], S_2), S_3, \mathcal{M}([X,Y], S_2)), S_5), S_6)$$

where X is the image width and Y the image height.

This program does not exploit stream parallelism, because we examine one image at a time. Instead it uses two map computations: the first one is done over each pixel of the input image and builds a list of the neighboring pixels of each region. A map form of parallelism is used because the same operation must be done for each pixel[2].

[2] Note that our map is not a SIMD operator.

After building these lists, the sequential stage S_3 decides which neighbor glue with each region and transmits these informations to the second map, which is done over each image region and simply merges each region with the selected neighbor. The process is iterated until no more regions can be glued.

4.3 3D Delaunay Triangulation

Three dimensional delaunay triangulation (TDE) is an application aimed at producing a volumetric visualization of sparse data sets [PDMT92]. It starts from a set of points randomly distributed in a 3D space and must compute a set of tetraedros whose vertex are three different points in space. The intersection between tetraedros in the set must always be null or limited to one or two sides. Tetraedros set must cover all the points of the starting dataset. We started from the InCoDe algorithm and developed different parallelization strategies[COS95]; one of them has the following structure:

$$\Pi(S_1, \mathcal{M}([X,Y], S_2), S_3, \mathcal{M}([Y], S_4), S_5)$$

The stage S_1 is the input stage, the first map partitions the input data space using a set of walls and computes all tetraedros that cross these walls. These walls divide the input data space in equally sized sections. After a data reorganization, done in stage S_3, the data structure proceed towards the second map, which computes the tetraedros that cover the points inside the walls.

4.4 A simple ray-tracer

This application implements a simple ray-tracer; scenes are build combining objects like sphere, planes, cones and polygons and using several light sources. Object bodies can have a color or a texture and a separate reflection and refraction coefficients. A sequential ray-tracer program was developed quickly without any optimization: simply for each pixel of the screen we send light rays and handle up to 5 ray reflection and refraction. Then we tried two different parallelization strategies.

4.4.1 Data-parallel ray-tracer

Data-parallel ray-tracer (DPR) parallelizes the ray-tracing algorithm by mapping the sequential algorithm over the set of pixels. The structure of the program is:

$$\Pi(S_1, \mathcal{M}([W,H], S_2), S_3)$$

Where stage S_1 is the input stage, which reads the description of the scene and sends it out; the map construct partitions pixel coordinates and broadcasts scene description along with each pixel. Map output is a pixel color matrix that contains computed picture. Map's worker (module S_2) is the complete sequential ray-tracing algorithm. The final stage (S_3) receives the computed image and writes it to a file.

4.4.2 Dataflow ray-tracer

Dataflow ray-tracer (DFR) is an alternative implementation of the same algorithm that should exploit fine grain parallelism. To program this we did a high level dataflow analysis of the program, cutting it into *macro-blocks* and re-building its structure using the available parallel structure. Using this strategy we wish to exploit the stream parallelism of the computation of different pixel onto different code blocks. Program structure is quite complex:

$$\Pi(S_1, \mathcal{L}(\Pi(\mathcal{F}(S_6), S_4, \mathcal{F}(\mathcal{L}(\Pi(S_8, S_9, S_{10}), S_7)), S_5), S_3), S_2)$$

The stage S_1 reads the scene description and outputs a stream of tasks, one for each pixel (160000 for a 400 by 400 pixel image); each task contains current pixel coordinates, scene description and a pixel state, which describes current pixel color and what we are doing now (i.e. if we are computing a 2^{nd} ray reflection or whatever). When a pixel leaves stage one its state shows that we must compute first ray directed color. Stage S_2 gathers the various pixels, which are now connected to their definitive color, and finally outputs the image. The main loop computes iteratively direct ray color, reflected ray color and refracted ray color; the algorithm is recursive and stops at depth 5 (i.e. we reflect a ray up to 5 times). The inner modules implement the basic code blocks:

- S_6 computes where each object intersects the current ray.
- S_4 checks object is first hit by the ray and sets ray status ready for the light source loop.
- S_8 prepares ray status to check for current light source color contribution.
- S_9 computes the current light source contribution to the ray color.
- S_{10} adds light source contribution to ray color.
- S_5 handles the recursive process: checks if we can have one more reflection or a refraction and eventually makes the pixel do another turn in the extern loop.

5 Experimental results

To validate the approach and the analytical models we developed for our run time systems, and to check if the predicted performance matches with real runs, we used our optimizing compiler to run all these programs onto our Meiko CS–1 transputer based system. In figure 5 we report the results obtained from real runs of the programs described in section 4 on our parallel system and we compare them with predicted values [3]. The **parallelism degree** column reports the size our compiler assigns to the parallel construct of the program, either to obtain a good balance between the load of each PE or due to resource reduction algorithm in order to fit the program into our small (50 nodes) system.

DPR and DFR runs are relative to a 400 by 400 pixel image with 20 objects and 10 lights, CFL has a 2800 items input fault list and partitions it into 95 sublists, CAT examines a 32 by 32 pixel image and DEL a 1024 point dataset. Absolute performance of the test programs is usually very high; for example the sequential ray-tracer application, used as a base for both parallel versions, takes 140 seconds of CPU time on an HP 9000/735 workstation with 64 Mbytes of ram and just 26.5 using 38 T800. A program that resulted in a bad parallelization strategy is the dataflow version of the ray-tracer, which is too fine grained for this kind of machine, but our compiler recognizes this and does not allocate unused resources.

[3] All times are in seconds.

program	pred.	meas.	error	parallelism degree
DPR	23.5	26.5	11%	main map: 6x6 workers
DFR	0.0012	0.0013	7%	intersect farm: 1 worker light farm: 2 workers
CFL	0.068	0.07	2%	main farm: 8 workers
CAT	23.5	25	6%	neighbor map: 4x4 workers glue map: 2x2 workers
DEL	19.2	20	4%	over-walls map: 2x2 workers intra-walls map: 2x2 workers

Figure 5: Comparison between predicted and measures service time; all times are in seconds.

These tests proved that we can provide both programmability and performance. Infact all these programs where written in a short period of time mostly by people with low or none previous parallel programming experience. They were able, often starting from standard sequential code or writing a sequential version of the application, to produce several different parallel versions in a few months. Some of these versions at first exhibited a poor performance or scalability, but simple code reorganization allowed for a big speedup.

Finally there's the portability question. We developed a PVM backend, especially to debug new code, and all these applications compile and run straightforward on a network of workstation. Unfortunately our network is severely loaded and we could not measure their effective performance.

We developed templates for several other parallel architectures and, thanks to our structured approach which allows us to predict performance in a simple and precise way, we can estimate the running time of these programs over a large set of machines. Figure 6 shows the predicted performance of all these programs onto various popular parallel architecture. Please note that circuit fault list service time is multiplied by 100 and dataflow ray-tracer by 1000 to make the graphs visible.

Values reported represent the best performance the program can reach on such a machine if enough resources are available; the resources needed are summarized in figure 7. This table also gives a measure of the scalability limit of each application when more resources are available; infact our compiler is able to allocate the "best" number of resources to any program, i.e. that inserting more processing elements does not lead to a performance increase.

6 Conclusions and future work

In this paper we describe a structured parallel programming methodology along with its language, called P^3L, and compiling tools. This kind of methodology is able to give a good solution to the basic problems of parallel programming. We are able to obtain:

- Programmability: we have a structured language in which the only way to express parallelism is to use an high level construct. The programmer does not need to write explicit inter–process communication and synchronization primitives but can concentrate only on the overall algorithm structure.

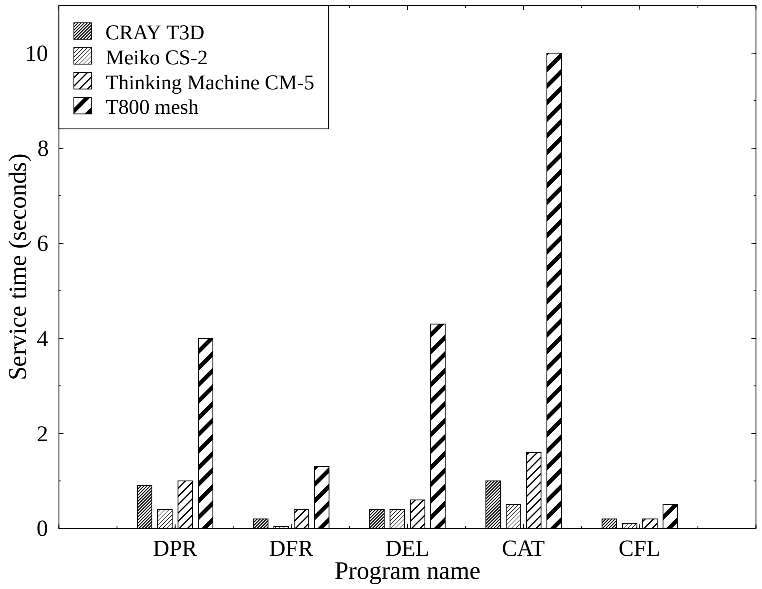

Figure 6: Testsuit programs minimum service time over different parallel architectures.

- Portability: the availability of a library of machine specific implementation templates allows us to compile the program on a wide range of target architectures without any changes in the source code. Finally the resource reduction algorithm allows us to run the program on machines of any size.
- Performance: our compiler uses analytical models, that describe construct performance, to optimize the program and transform its parallel structure in order to obtain good performance figures.

We show that this is possible reporting the results obtained coding a set of real–world applications using our language and compiling tools. All the applications were coded in a short period of time, re-using a great amount of existing sequential C code, and all obtained good performance figures without the need for hand optimizations or a tuning phase.

Actually our compiling tools are not well suited to be used in solving highly dynamic problems due to the static data structure assumption, but we are extending them, along with the template libraries and analytical models, in order to deal with constructs able to exchange dynamically sized data structures. Dynamic structures would have been very useful in the DFR to reduce the overheads associated with the recursive control of the algorithm. Also more work needs to be done in the selection of the construct to be included in a structured parallel languages: they must be enough to permit the programmer to efficiently code any application, but their number must be limited to keep template libraries small and not confuse the user with too much possibilities.

Program	T3D workers	CS–2 workers	CM–5 workers	T800 mesh workers
DPR main map	7x7	10x10	10x10	25x25
DFR intersect farm	1	1	1	1
DFR light farm	1	1	1	2
DEL over-walls map	3x3	3x3	3x3	4x4
DEL intra-walls map	2x2	2x2	2x2	2x2
CAT neighbor map	3x3	5x5	4x4	7x7
CAT glue map	1x1	1x1	1x1	3x3
CFL main farm	8	20	13	100

Figure 7: Testsuit programs maximum parallelism degree over different parallel architectures.

Acknowledgments

The research reported here would have been impossible without all the students who did the work of coding and debugging the applications: Maria Gabriella Brodi, Domenica Barresi, Giacomo Giunti, Stefano Milana, Paolo Pesciullesi, Antonio Biso, Luca Bordin, Alessia Conserva, Paola Criscione, Gianni De Giorgi. A special thanks also to Bruno Bacci for his work on the P^3L compiler.

A A subset of the analytical model

Here we report a subset of the analytical model used inside our compiler; a bigger subset of the model is described, along with its validation experiments, in a forthcoming paper.

Machine specific parameters

$\mathcal{P}(T_{in}^{com}, T_{in}^{ov}, T_{out}^{com}, T_{out}^{ov}, T_c)$	:	node parallelism.
$T_{snd}(len)$	:	send comm. time.
$T_{rcv}(len)$	:	receive comm. time.
$T_{snd}^{ov}(len)$	:	send overhead.
$T_{rcv}^{ov}(len)$	:	receive overhead.
g_d	:	grid mapping par. direction.
g_p	:	grid mapping pipelinization.

Symbols

k	:	Input task granularity.	N	:	Number of array dim.
s_{in}	:	Size of input data.	N_i	:	Card. of dim. i.
s_{out}	:	Size of output data.	d_{in}	:	Size of input array elem.
T_c	:	Seq. computation time.	d_{out}	:	Size of output array elem.
T_{ovr}	:	RTS sequential code time.	b_{in}	:	Size of input broadcasted data.
n	:	Parallelism degree.	b_{out}	:	Size of output common data.

Seq

$$\mathcal{T} = \mathcal{P}(T_{rcv}(s_{in}), T^{ov}_{rcv}(s_{in}), T_{snd}(s_{out}), T^{ov}_{snd}(s_{out}), T_c)$$

Pipe

$$\mathcal{T} = \max_{i=1..h}\{\mathcal{T}^i\}$$

Farm

$$\begin{aligned}
\mathcal{T} &= \max\{T_{emit}, T_{coll}, \tfrac{\mathcal{T}_w}{n}\} \\
T_{emit} &= \mathcal{P}(T_{rcv}(s_{in}), T^{ov}_{rcv}(s_{in}), T_{snd}(s_{in}), T^{ov}_{snd}(s_{in}), T_{ovr}) \\
T_{coll} &= \mathcal{P}(T_{rcv}(s_{out}), T^{ov}_{rcv}(s_{out}), T_{snd}(s_{out}), T^{ov}_{snd}(s_{out}), T_{ovr})
\end{aligned}$$

Loop

$$\begin{aligned}
\mathcal{T} &= l\max\{T_{in}, T_{out}, \mathcal{T}_{body}\} \\
T_{in} &= \mathcal{P}(T_{rcv}(s_{in}), T^{ov}_{rcv}(s_{in}), T_{snd}(s_{in}), T^{ov}_{snd}(s_{in}), T_{ovr}) \\
T_{out} &= \mathcal{P}(T_{rcv}(s_{out}), T^{ov}_{rcv}(s_{out}), T_{snd}(s_{out}), T^{ov}_{snd}(s_{out}), T_{ovr} + T_{cond})
\end{aligned}$$

Map 2D

$$\begin{aligned}
\mathcal{T} &= T_{dis}() + T_c d_1 d_2 + T_{col} \\
n_1 &= \frac{N_1}{d_1} \\
n_2 &= \frac{N_2}{d_2} \\
\overline{D} &= [n_1, n_2] \\
s^1_{in} &= b_{in} + d_{in} d_1 N_2 \\
s^2_{in} &= b_{in} + d_{in} d_2 \\
T_{dis} &= \mathcal{P}(T_{rcv}(s_{in}), T^{ov}_{rcv}(s_{in}), n_1 g^u_p(\overline{D}, 1) T_{snd}(s^1_{in}), n_1 T^{ov}_{snd}(s^1_{in}), T_{ovr}) + \\
&\quad \mathcal{P}(g^u_p(\overline{D}, 1) T_{rcv}(s^1_{in}), T^{ov}_{rcv}(s^1_{in}), g^u_p(\overline{D}, 2) n_2 T_{snd}(s^2_{in}), n_2 T^{ov}_{snd}(s^2_{in}), T_{ovr}) \\
s^1_{out} &= b_{out} + d_{out} d_1 N_2 \\
s^2_{out} &= b_{out} + d_{out} d_2 \\
T_{coll} &= \mathcal{P}(T_{rcv}(s_{out}), T^{ov}_{rcv}(s_{out}), n_1 g^u_p(\overline{D}, 1) T_{snd}(s^1_{out}), n_1 T^{ov}_{snd}(s^1_{out}), T_{ovr}) + \\
&\quad \mathcal{P}(g^u_p(\overline{D}, 1) T_{rcv}(s^1_{out}), T^{ov}_{rcv}(s^1_{out}), g^u_p(\overline{D}, 2) n_2 T_{snd}(s^2_{out}), n_2 T^{ov}_{snd}(s^2_{out}), T_{ovr})
\end{aligned}$$

References

[AP92] S. Antonelli and S. Pelagatti. On the complexity of the mapping problem for massively parallel architectures. *International Journal of Foundations of Computer Science*, 3(3):379–387, September 1992.

[BCGR93] G.P. Balboni, G.P. Cabodi, S. Gai, and M. Sonza Reorda. A parallel system for test pattern generation. In *Model and Tools for Massively Parallel Architectures*. Consiglio Nazionale delle Ricerche – Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Sottoprogetto Architetture Parallele, June 1993.

[BDO+95] 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.

[BDP94] B. Bacci, M. Danelutto, and S. Pelagatti. Resource optimization via structured parallel programming. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 13–25. Birkhäuser Verlag, Basel, Switzerland, 1994.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, Cambridge, Massachusetts, 1989.

[COS95] P. Criscione, S. Orlando, and S. Scopigno. Metodologia di programmazione parallela P^3L: un'applicazione a problemi di geometria computazionale. In *Proc. of AICA 95*, volume 1, pages 221–229, Chia (Cagliari), September 1995. In Italiano.

[DFH+93] 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. Springer-Verlag, 1993.

[DMO+92] 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.

[For93] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Version 0.1, May 1993.

[KMaNMR92] G. Koepfler, JM. Mores, S. Osher amd N.J. Mankovich, and L. Rudin. Application of a new pyramidial segmentation algorithm to medical images. *SPIE Medical Imaging*, 1992.

[Kun89] H.T. Kung. Computational models for parallel computers. In R. J. Elliott and C. A. R. Hoare, editors, *Scientific Applications of Multiprocessors*, pages 1–17. Prentice Hall, 1989.

[OV94] S. Orlando and M. Vanneschi. Compiling Geometric Paradigms through Local Communications. In *Proc. of IFIP International Conference on Applications in Parallel and Distributed Computing, Caracas, Venezuela*, April 1994.

[PDMT92] E. Puppo, L. Davis, D. De Menthon, and Y.A. Teng. Parallel Terrain Triangulation. In *Proc. of 5th International Symposium on Spatial Data Handling*, pages 632–641, August 1992.

[Pel93] 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.

[Ski91] D. B. Skillicorn. Models for Practical Parallel Computation. *International Journal of Parallel Programming*, 20(2):133–158, April 1991.