Technical Report: TR-96-08, April 16, 1996

Machine independent Analytical models for cost evaluation of template–based programs

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. The power of this approach stands in the possibility to build an optimizing template–based compiler using low time complexity algorithms. In order to optimize the code, this compiler needs formulas that describe the performance of language constructs over the target architecture. We propose a set of parameters able to describe current parallel systems and build deterministic analytical models for basic forms of parallelism. The analytical model describes construct performance in a parametric way.This can be done by knowing that the compiler exploits a template–based support and giving template implementors guidelines to follow to make actual implementation perform as predicted.

ACM–CR Subject Classification:

Keyword and phrases: Skeletons, performance modeling, parallel languages, template–based compilers

Machine independent Analytical models for cost evaluation of template–based programs

Davide Pasetto and Marco Vanneschi

Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy,
tel +39 50 887228, fax +39 50 887226
e-mail: {davide,vannesch}@di.unipi.it

April 16, 1996

Abstract

Structured parallel programming is one of the possible solutions to exploit Programmability, Portability and Performance in the parallel programming world. The power of this approach stands in the possibility to build an optimizing template–based compiler using low time complexity algorithms. In order to optimize the code, this compiler needs formulas that describe the performance of language constructs over the target architecture. We propose a set of parameters able to describe current parallel systems and build deterministic analytical models for basic forms of parallelism. The analytical model describes construct performance in a parametric way. This can be done by knowing that the compiler exploits a template–based support and giving template implementors guidelines to follow to make actual implementation perform as predicted.

Keywords: Skeletons, performance modeling, parallel languages, template–based compilers.

1 Introduction

Parallel programming is a difficult task because the programmer should take into account a great number of problems, such as mapping, load balancing, resource usage and lots more. 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 writing efficient and portable applications but unfortunately building a tool able to deal efficiently with general parallel programs is very hard due to the size and complexity of the problems involved [AP92]. Infact modeling a machine architecture in such a way to be able to predict and optimize parallel program performance requires the use of very complex analytical models and of time consuming algorithms. A possible solution to this problem is the skeleton approach [Kun89, Col89, DFH+93, DMO+92, BDO+94]: restrict the program structure to basic forms of parallelism, called skeletons, and compose and nest them to build a parallel application.

This approach is feasible because most parallel applications exhibit the same parallel structure exploiting the same *kind of parallelism*, such as data–parallel, pipeline and so on. Each skeleton refers to a specific form of parallelism. Having a restricted set of parallel process interaction graphs we can devise good implementation templates, one or more for each form of parallelism, which take into account mapping, load balancing and so on, and solve the resulting problem for a given architecture. It is also simpler to develop formulas which describe the performance of these templates for a given machine and use these formulas to optimize both efficiency and performance [Pel93].

The skeleton computational model tries to obtain all the usual objectives:

Programmability can be obtained because the model permits *only* high–level programming of an application: the user cannot choose arbitrarily the parallel interaction structure, he cannot insert in his program low level communication and/or synchronization primitives. He can only use the available basic forms of parallelism, which have a simple and precise semantic like sequential language constructs.

Portability is possible because the number of basic forms of parallelism is *limited*. Knowing this we can choose an implementation for each form of parallelism on each parallel machine; obviously the **implementation templates** must be defined in such a way to allow composition, nesting and tuning of internal parameters. We must also choose a mapping algorithm able to layout the resulting parallel program on each target machine, but we can use low complexity heuristics

Performance can be obtained building a precise analytical model, able to predict construct performance on each target machine, and applying several optimizations in order to restructure the program and obtain better performance by tuning it to a specific machine hardware.

Several problems must still be addressed in order to make skeleton based approach general and to be able to build powerful compilation tools. In this article we address following questions:

- How can we model template performance for a large set of parallel architectures? Which kind of construct optimizations are we able to support?
- How should we define implementation templates for a given system architecture for each form of parallelism in order to allow template composition and nesting?

First of all, in section 2 we briefly describe the basic forms of parallelism we are interested in. Along with these forms, constructs of our structured parallel language, we describe a the high level implementation strategy we use to test our analytical model. Then, in section 3, we describe the inner working of a template–based compiler for a structured parallel language and, in the following section, we introduce our machine architecture analytical model parameters. The number of parameters and functions is high but this model has been studied to be used by a compiler and not directly by users. In section 5 we develop an analytical description for the performance of each basic forms of parallelism when implemented using the previously introduced high level template. Obviously other implementation templates can be defined and they can also be modeled using the proposed parameters. In section 6 we show how to physically implement the high level templates previously described onto two target architectures: 3D cubes and dual fat-tree based machines. In section 7, we show how our analytical model can accurately predict parallel program performance and section 8 contains some concluding remarks.

2 Basic forms of parallelism

The parallel computation model we are interested in is **skeleton–based**: users are not allowed to use any parallel interaction structure but must restrict themselves to a limited set of basic forms of parallelism [Col89, DMO+92]. Applying this approach we defined a structured parallel language, called P^3L, in which each construct denotes a specific form of parallelism, that is a single parallel interaction structure. Each construct can be customized with the desidered behavior using a limited set of parameters, such as input data list, output data list, body element, computed function, etc. The user is then able to build a complex parallel application just nesting and composing basic parallel structures, which is like defining and calling subroutines in a sequential language.

Structured parallel programs can be described using the **construct tree**, which shows how the programmer choosed to compose and nest the basic forms of parallelism used. Each tree node is labeled as a parallel construct, tree leaves are sequential code blocks and tree arcs show the types of the data that should flow between the constructs. The language has a **macro dataflow semantic**: each construct is a pure functional module and data flow between constructs like a **stream of tasks**; this way if we find a suitable performance model we are able to transform program construct tree in order to obtain another parallel structure with better performance. In this document we state that data exchanged between construct are defined statically at compile time so the compiler can compute their size and communication time: we do not allow dynamically sized structures to move between constructs.

Now we describe the set of basic forms of parallelism, constructs of our structured parallel language, we refer in this document. Each skeleton can be implemented using several techniques and/or strategies which define **high-level templates**; in the following description we report one of these high level templates. We'll analyze the following forms:

Sequential construct represents the leaves of the construct nesting tree and contains the basic code blocks of the application. This construct is a pure functional module: it receives input data from its input channel, executes its own sequential function over the received data and outputs the results to the output channel. A single function call is named **task**. Both input and output data, relative to successive tasks, can be grouped into a **token** to reduce the overheads for sending and receiving messages and enhance program performance. The number of tasks contained into a single token is called **task granularity**. Each sequential construct is implemented on a distinct processing element: no multitasking is allowed.

Pipe construct expresses pipeline parallelism. Each stage of the pipe is implemented on his own; the mapper can use the locality information given by stage sequence to map stages in order to reduce long distance communications.

Farm construct expresses the stream parallelism based on the independence of activation of pure functional modules on different input data. We implement this construct mapping the emitter, the collector and all the workers on different processing element.

Loop construct is used to model parallel iterative computation of each element of the input stream over a loop body. To implement this form of parallelism we use, apart from the loop body implementation, two more PEs: the first is called **loop–input** and receives tasks from the loop input channel and from a feedback channel and transmits them to the body. The second one is called **loop–output** and receives the body results, evaluates the exit condition and sends the results through the construct output channel or back to loop–input through the feedback channel.

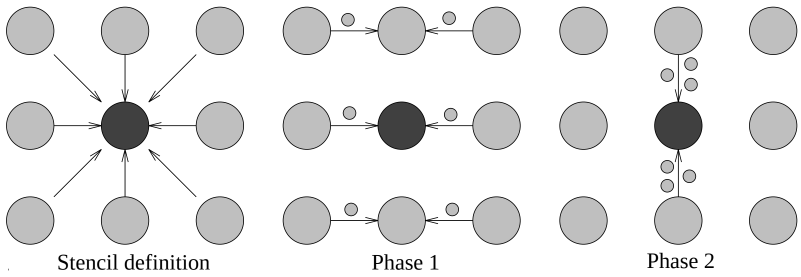

Figure 1: A stencil example and its implementation.

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 [OV93, For93]. The output of this construct is an array with same dimensions. The input token can also contain data items that should be used in the computation of each new component of the array; these data items are called **broadcasted data**. The computation phase consists in a loop of steps; each step begins with a data gathering from a set of neighboring processors, called the **stencil**, and is followed by a local computation. Both the number of iterations and the data access stencil must be known at compile time (it is possible to add support for a *reduce* operation every k steps to check for the convolution termination instead of statically setting the iteration number). Actually we allow this construct to have only a sequential body.

A possible implementation of this form of parallelism is based on **virtual processors** and **virtual topology**: a virtual processor is defined as a sequential module that performs the computation of a single array element. The virtual topology is the topological placement of virtual processors, in this case an n dimensional cube like the input and the output arrays. **Virtual processor granularity** stands for the amount of virtual processors mapped on a single processing elements, i.e. the size of the sub–cube we place on each PE. For example, if we must operate on a 400x400 array then we have 160000 virtual processors arranged as a 400x400 grid; we can choose a virtual processor granularity of 10x10, mapping 100 virtual processors on each PE, and obtaining a 40x40 PE grid to map on our machine hardware.

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.

3 Building an optimizing compiler for a structured parallel language

We developed an optimizing compiler for our structured parallel language. The compiler is based on libraries of implementation templates and on performance models for these templates for specific machines. Both the library and the model are replicated for each target machine. The compilation algorithm used in our compiler is able to optimize both program performance and resource usage [BDP94].

The compilation proceeds through the following stages:

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. 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.

All the algorithms needed to implement such approach have a polynomial time complexity. Obviously the key point of this approach is to be able to predict program performance with an high approximation. If we devise an analytical model able to describe the performance of basic forms of parallelism over all parallel machine hardware using a limited set of parameters, we will be able to port such a compiler onto different architecture just writing the physical implementation templates and adding the new machine description in the architectural model. In the following sections we show how to build and use such a model for a **message passing style** implementation of the template library; we are studying other techniques too, like shared memory, collective operations and mixed implementations.

4 The architectural model

We are interested in MIMD architectures with distributed memory. A MIMD system is composed by a set of processing elements (PEs). Each PE is build out of one or more processors and some memory banks and is connected to one or more interconnection networks through a network interface. While the node processors are usually standard RISCs (as they come from the high speed workstation world), the network and its interface change from machine to machine and make modeling these machines using a limited set of parameters a difficult task. In this paper we model MIMD architecture by looking at their basic performance figures and we select these figures as the building blocks of our formulas.

This approach has previously been used in other models, such as the LogP model [CKP+93]. Unfortunately this model is too restricted to predict performance with an high approximation: for example it does not include a support for special block transfer hardware (although this has been added in a successive revision), it cannot model important contention–free communication pattern, it's difficult to model situations with parallel input, output and computation, and has other severe limitation. These limits arise from the fact that LogP model has been studied to help *humans* to write applications, so it is simple enough to be used by hand, while the analytical model we need is used by the *compiling tools*, so can be much more complicated.

The reference abstract architecture is quite standard: a set of computing elements, each with a number of network connections, and a global communication network that supports message routing. To model the performance we use the following parameters:

Processor speed. In order to analyze PE performance figures we must take into account node processor speed: the processor must execute the sequential code that is part of the application as well as the run time support (RTS). We include processor speed between our parameters simply by benchmarking or computing the time spent in executing both user and RTS code.

Network ports. Each processing element is connected to the communication network through an interface which uses one or more ports to inject messages into the network. The number of these ports pose an upper limit to the number of communications a PE can perform in parallel.

Node architecture. We must also take into account the special features allowed by a communication processor or by a specific node architecture. To obtain the better performance we wish to overlap communication and computation whenever is possible. We model node performance by using a function $\mathcal{P}$ which express the service time of a module that reads an input token, executes a local computation upon the received data and finally outputs a result token. If the architecture supports parallel input, output and computing, the implementation of this procedure should exploit pipeline parallelism in order to obtain better bandwidth. The function is defined as:

$$\mathcal{P}(T_{in}^{com}, T_{in}^{ov}, T_{out}^{com}, T_{out}^{ov}, T_c)$$

where T_c is the local computation running time, T^{com} is the data communication time and T^{ov} is the communication overhead, that is the time needed to setup the network interface. To simplify the exposition we consider only networks of processes where each process that can receive data from a single channel and send data to a single channel at a time. The model can be extended to use a family of functions indexed by the number of "simultaneous" input and output communications required. We use this function to model specific node architectures: for example CM–5 nodes [KTR93] cannot do computation and communication in parallel, on such a machine the definition of $\mathcal{P}$ is simply the sum of all the parameters. Transputer–based systems instead [Hey89, Cok91] have a direct memory access engine for each communication channel (4 input and 4 output channels); this leads to the following formula:

$$\max\{T_c + T_{in}^{ov} + T_{out}^{ov}, T_{in}^{com}, T_{out}^{com}\}$$

Network and interface speed. Network and interface performance can be modeled using the following parameters:

$$\begin{array}{lcl} T_{snd}(len) & : & \text{send comm. time.} \\ T_{rcv}(len) & : & \text{receive comm. time.} \\ T_{snd}^{ov}(len) & : & \text{send overhead.} \\ T_{rcv}^{ov}(len) & : & \text{receive overhead.} \end{array}$$

Grid mapping. Several basic forms of parallelism, especially data–parallel one like geometric or map constructs, operate on multi-dimensional vectors, so we must map a grid computation onto the target machine. To express the overheads of this operation we sum–up the relevant info from network type, topology, routing and flow control in two parameters. First of all we need to exchange data between logically neighboring processors one dimension at a time and (possibly) in both directions in parallel. To handle this we define the parameter g_d:

$$g_d = \begin{cases} 2 & \text{if must do one direction at a time} \\ 1 & \text{if can do both directions in parallel} \end{cases}$$

We must also know how many communications used to implement this data movement can be pipelined through the interconnection structure; to express this we define two functions, called **pipelinization factors**, the first relative to unidirectional communications and the second to bidirectional one:

$$g_p^u(\overline{D}, i)$$

$$g_p^b(\overline{D}, i)$$

where $\overline{D}$ is a vector of n elements (D_j is the number of PEs in dimension j) and i is the dimension through which we move data; this parameter heavily depends on network flow control and routing algorithm.

We don't explicit include contention in the network performance model because recent studies and experiments[BYA89, Dal90, AmKV93, KTR93, SGC93] show that, if the communication pattern does not exhibit hot spots, message latency raises very slowly until a specific load level (dependent on routing and topological solutions) is reached. If the load increases above the threshold the network quickly saturates. This means that, if we can avoid hot spots at compile time, it is possible to use a network below his saturation point and so consider latency as a (small) constant. For example the CM–5 network [LA+92] is over–dimensioned with respect to its processing node speed, so each processor can inject messages in the network as fast as he can without saturating it. KSR–1 network instead is slower [WB+92, RS+93, SGC93] when using level 1 ring for intra–cluster communications, so in this system we should limit message injection rates to avoid congestion.

A structured parallel language compiler can use the available analytical models, which describe the performance of each implementation template, to balance the communication and computation load of each processor and so completely avoid hot spots at compile time. Moreover template implementors have a great amount of locality informations given by the construct communication pattern and nesting tree, so they can use local communication whenever is possible when defining template shapes and can eventually limit injection rates by artificially rising the values of data send and receive times.

Please note that the parameters here introduced can be used to model the process networks we are interested in, that is those produced by the template based support of a structured parallel program, and do not represent a general parallel computation model.

5 Construct implementation models

Now we devise an analytical model which describes the performance of each high level template described along with each language construct. This model defines the **mean service time** of each construct parametric with respect to the input task granularity (that is the number of task per input token). The inputs required by the analytical model are the mean service time of the user and system sequential code and the sizes of the exchanged data. Basic performance figures for some of the constructs change if the running time distribution has a large variance and we are developing a probabilistic extension to the the formulas reported here to be able to use running time variance informations to better predict service times. It's also possible to define the construct competition time, that is the time spent by the program in executing a computation on a stream of given length, including the effect of start–up and end–up periods.

Along with each analytical model we briefly comment how the template must be implemented in order to perform as predicted and report some of the optimizations the compiler is able to perform.

Throughout the formulas we'll use the following parameters:

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.

Sequential. The performance of a sequential code module derives directly from the node architecture (recall that each of them is mapped onto a different PE):

$$\mathcal{T}(k) = \mathcal{P}(T_{rcv}(ks_{in}), T^{ov}_{rcv}(ks_{in}), T_{snd}(ks_{out}), T^{ov}_{snd}(ks_{out}), kT_c)$$

Obviously if the machine architecture allows parallel computation and communication the runtime support should exploit it in order to enhance performance.

Pipe. In an h stage pipeline we denote with $\mathcal{T}^i(k)$ the performance of stage i when used with input granularity k; the whole construct exhibits the following service time:

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

Template implementors can use the locality information to map stages in such a way to minimize long distance communications and reduce network load. The optimization strategy must balance the load between stages to enhance resource usage; to do this it is possible to merge adjacent sequential stages that are too fast with respect to the others, to farm–out a slow stage and to insert a specific number of resources in a parallel stage to obtain a given performance. It's also possible to change task granularity (that is the number of tasks per token) between two adjacent stages in such a way to obtain the optimal granularity required to balance communication and computation load of each module.

Farm. Recall that the particular farm template we refer in this document has an emitter, a collector and some workers that run in parallel; this means that the emitter and the collector are mapped onto different processing nodes. The emitter exploits the load balancing strategy best suited for the machine architecture (for example if the system allows only local communications it should implement some kind of distribution bus, otherwise workers can use non–local communication to ask for a new task) [DPV94]. We call h the task granularity internally used by the construct (this granularity can be different from input task granularity) and $\mathcal{T}_w(h)$ the farm body service time. The performance of this high level template can be expressed as follows:

$$\begin{aligned}
\mathcal{T}(k) &= \max\{\tfrac{T_{emit}(k,h)}{h}, \tfrac{T_{coll}(k,h)}{h}, \tfrac{\mathcal{T}_w(h)}{n}\} \\
T_{emit}(k,h) &= \mathcal{P}(hT_{rcv}(ks_{in}), hT^{ov}_{rcv}(ks_{in}), kT_{snd}(hs_{in}), kT^{ov}_{snd}(hs_{in}), T_{ovr}) \\
T_{coll}(k,h) &= \mathcal{P}(kT_{rcv}(hs_{out}), kT^{ov}_{rcv}(hs_{out}), hT_{snd}(ks_{out}), hT^{ov}_{snd}(ks_{out}), T_{ovr})
\end{aligned}$$

The optimization process, minimizing this formula, can find the maximum number of workers (n) to insert, still maintaining a good resource usage, and the optimum task granularity (h).

Loop construct implementation considered here uses an input process, which reads data from the input and the feedback channels and sends them to the loop body, and an output process, which computes the exit condition for each token. These processes, and also the body, are mapped onto different PEs. Knowing that input and feedback tokens have the same size, construct performance can be defined in terms of the mean number of iterations (l) and construct internal task granularity (h):

$$
\begin{aligned}
\mathcal{T}(k) &= l \max\{\frac{T_{in}(k,h)}{h}, \frac{T_{out}(k,h)}{h}, \mathcal{T}_{body}(h)\} \\
T_{in}(k,h) &= \mathcal{P}(hT_{rcv}(ks_{in}), hT^{ov}_{rcv}(ks_{in}), kT_{snd}(hs_{in}), kT^{ov}_{snd}(hs_{in}), T_{ovr}) \\
T_{out}(k,h) &= \mathcal{P}(kT_{rcv}(hs_{out}), kT^{ov}_{rcv}(hs_{out}), \\
&\quad hT_{snd}(ks_{out}), hT^{ov}_{snd}(ks_{out}), T_{ovr} + T_{cond})
\end{aligned}
$$

The optimization process can find the optimal construct task granularity and must balance the load between input/output nodes and loop body. If the body is just a sequential construct the loop can be removed and transformed into a single sequential construct, obviously only if this gives better performance.

Geometric implementation template analyzed here uses an emitter, a collector and a set of workers logically arranged in a virtual topology like an n–dimensional grid. Data loading and collection is pipelined into the structure rows and columns. This template maps each array element on a single virtual processor in order to obtain the maximum degree of parallelism; having a bounded number of physical processors and non zero communication time, we probably need to map several virtual processors onto each physical processor to balance communication and computation loads. This grouping is represented by parameter d_i (data granularity for dimension i): if the i dimension of the grid has cardinality N_i, we allocate a grid with $\frac{N_i}{d_i}$ PEs in dimension i. The construct receives k requests within a token and sequentially executes them by distributing data to the available workers, computing the function in parallel on each worker and collecting the output data. Construct performance is:

$$
\mathcal{T}(k) = k(T_{dis}() + T_{calc} + T_{col})
$$

Data loading strategy examined here implements a pipelined distribution tree and proceeds one dimension at a time: suppose that dimension i has cardinality n_i, the emitter builds n_1 packets each consisting in a $n_2 \times n_3 \times \ldots \times n_n$ elements plus one copy of broadcasted data.

These packets are loaded in the first row using a pipelined technique; then each worker in the first row builds n_2 packets splitting the data he received and adding one copy of broadcasted data and sends them down in pipeline along his column and so on until last dimension. Data collection works the reversed way. The performance of this data loading and collection strategy can be expressed as follows:

$$
\begin{aligned}
n_i &= \frac{N_i}{d_i} \\
\overline{D} &= [n_1, \ldots, n_N]
\end{aligned}
$$

$$
\begin{aligned}
s_{in}^{i} &= b_{in} + d_{in} d_i \prod_{j=i+1}^{N} N_j \\
T_{dis} &= \mathcal{P}(T_{rcv}(s_{in}), T_{rcv}^{ov}(s_{in}), n_1 g_p^u(\overline{D}, 1) T_{snd}(s_{in}^1), n_1 T_{snd}^{ov}(s_{in}^1), T_{ovr}) + \\
&\quad \sum_{i=2}^{N} \mathcal{P}(g_p^u(\overline{D}, i-1) T_{rcv}(s_{in}^{i-1}), T_{rcv}^{ov}(s_{in}^{i-1}), \\
&\quad g_p^u(\overline{D}, i) n_i T_{snd}(s_{in}^i), n_i T_{snd}^{ov}(s_{in}^i), T_{ovr}) \\
s_{out}^{i} &= b_{out} + d_{out} d_i \prod_{j=i+1}^{N} N_j \\
T_{coll} &= \mathcal{P}(T_{rcv}(s_{out}), T_{rcv}^{ov}(s_{out}), n_1 g_p^u(\overline{D}, 1) T_{snd}(s_{out}^1), n_1 T_{snd}^{ov}(s_{out}^1), T_{ovr}) + \\
&\quad \sum_{i=2}^{N} \mathcal{P}(g_p^u(\overline{D}, i-1) T_{rcv}(s_{out}^{i-1}), T_{rcv}^{ov}(s_{out}^{i-1}), \\
&\quad g_p^u(\overline{D}, i) n_i T_{snd}(s_{out}^i), n_i T_{snd}^{ov}(s_{out}^i), T_{ovr})
\end{aligned}
$$

The pipelinization factor g_p^u is used to keep track of the conflicts that arise between different messages along the same dimension.

We define the actual data access stencil as the smallest nD subcube that contains the elements referenced by a given logical processor. Data gathering operation is implemented through iterated data shifts: suppose that the maximum data access distance for dimension i is l_i; first of all we do a shift along dimension 1 in both directions moving each data by l_1 positions, then we build in each processor a packet containing all the data it has received plus its local data. These packets are shifted along dimension 2 by l_2 positions in each direction, and so on until last dimension. With n data shift phases, phase i shifts data along dimension i doing $\lceil \frac{l_i}{d_i} \rceil$ communications in both directions between logically adjacent processing nodes. Each phase shift moves all the data moved by previous phases plus the local data. The number of vector elements exchanged in phase i (s_i) and the number of shifts to do in phase i ($l_i^{'}$) are:

$$
\begin{aligned}
l_i^{'} &= \lceil \tfrac{l_i}{d_i} \rceil \\
s_i &= \begin{cases} 1 & \text{if } i = 1 \\ s_{i-1}(2 l_{i-1}^{'} + 1) & \text{if } i > 1 \end{cases}
\end{aligned}
$$

Computation of gathering time of phase i (T_{gat}^i) depends heavily on node and network architecture: if the PE has at least 2 I/O network interfaces we should map grid computation in such a way to do shifts in both directions in parallel, if the network characteristics allow us to do so (recall that this is described by g_d parameter), to obtain a better performance. Also we must use the pipelinization factor for data shifts, called g_p^b, to know how many communications of a particular data shift can happen in parallel.

$$
\begin{aligned}
T_{gat}^i &= g_d l_i^{'} \mathcal{P}(g_p^b(\overline{D}, i) T_{rcv}(s_i d_{in}), \\
&\quad T_{rcv}^{ov}(s_i d_{in}), g_p^b(\overline{D}, i) T_{snd}(s_i d_{in}), T_{snd}^{ov}(s_i d_{in}), T_{ovr})
\end{aligned}
$$

$$T_{gat} = \sum_{i=1}^{n} T_{gat}^{i}$$

The overall construct computation time, for an l loop computation, is:

$$T_{calc} = l(T_{gat} + T_c \prod_{i=1}^{N} d_i)$$

The optimization process can devise the optimum data partitioning scheme, that is the vector $< d_1, \ldots, d_N >$ which leads to best performance by minimizing the service time function, which keeps track of data distribution, collection and iterative computation times. Obviously there are other implementation algorithms for the gather phase and they can also be modeled using the architectural parameters introduced here. The compiler should choose the algorithm which performs best for a given communication stencil. The proposed method is good if the stencil is very similar to a nD subcube centered around the requesting processor; if this is not the case other data gather strategy could move less redundant data than this one.

Map construct, as a subcase of geometric one, can be implemented on a logical nD grid of processors and using the same data distribution and collection strategies. What changes is just the computation time because we do not iterate and don't need a data gathering phase. This leads to the following performance formula:

$$\mathcal{T}(k) = k(T_{dis}() + T_c \textstyle\prod_{i=1}^{N} d_i + T_{col})$$

6 Template implementation onto real machines

Now we show how template implementors can use the high level implementation template to create process networks performing as predicted by the analytical model. Process interaction structures must be defined differently according to network type, topology and features. Processing node architecture does not necessarily change the shape of a template for a construct but changes its performance and its speedup figures. Having a restricted and statically defined set of process graph to implement, compiler developers can map processes and communication channels aiming at reducing network contention. In order to be able to implement construct composition and nesting using simple mapping techniques, each template is defined as a process graph whose shape is a subset of the topology it is studied for. Obviously we map the sequential construct on a single processing element.

6.1 Direct 3D mesh with e–cube routing

Several mapping strategies can be employed in 3D mesh based networks and most of them are just some kind of heuristic; in this work we report a methodology where each template is considered as a 3D cube with input and output nodes at opposite corners (see figure 2.a). The compiler should try out several different strategies in order obtain a better mapping of the program.

Pipe template is very simple and can be obtained just putting one stage after the other. This linear pipeline can be transformed in 3D by wrapping it two times.

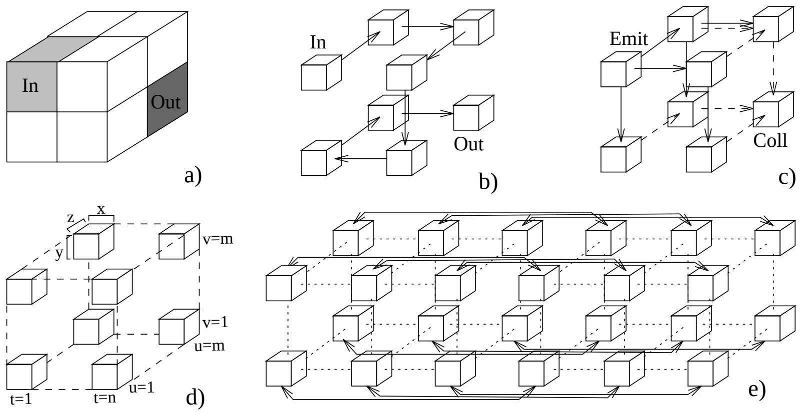

Figure 2: Templates for 3D mesh network with e–cube routing.

In figure 2.b we show a wrapping strategy that reduces link conflicts at minimum and works well when stages have about the same size.

Farm template is obtained by arranging all the workers plus an emitter process and a collector process in a 3D grid as shown in figure 2.c. Emitter and collector are positioned at opposite corners while workers run on every other node inside the cube; the figures show also how conflicts for link usage between emission and collection communications are reduced to a minimum.

Map Mapping a nD grid into a 3D grid is quite simple: if n is 1 (or 2) we just make a "segment" (or a plane) and wrap it up to obtain a 3D structure; if n is more than 3 we divide it in a number of 3D subcubes and put them one after the other (see figure 2.d). If $n > 3$ we have conflicts when doing parallel movements of data between different 3D subcubes. As we can see in figure 2.e for a $3 \times 2 \times 2 \times 2$ structure, when shifting data along the fourth dimension we have link conflicts. This conflicts, whose number is equal to dimension cardinality, can result in a total communications serialization when pure wormhole flow control is used (if flow control is packet–switched or virtual cut–through this does not happen). The pipelinization factor for distribution with wormhole flow control is then equal to:

$$g_p^u(\overline{D}, i) = \begin{cases} 1 & \text{if } i < 4 \\ D_i & \text{otherwise} \end{cases}$$

while is always 1 when another flow control strategy is used.

Geometric template is the same as the map one. With this kind of topology data shifts can occur in parallel in both directions (obviously if links are bidirectional) so $g_d = 1$. As in map template wormhole flow control can reduce performance when $n > 3$ due to serialization of the communications used to implement data distribution and data shifts. In such a machine we will map the structure in such a way that the last 3 shift phases, which move the largest amount of data, do not exhibit contention in the network. The pipelinization factor for data shifts on a wormhole network is then equal to:

$$g_p^b(\overline{D}, i) = \begin{cases} 1 & \text{if } \|\overline{D}\| - i < 3 \\ D_i & \text{otherwise} \end{cases}$$

while is always 1 with another flow control strategy.

6.2 Indirect dual generalized Fat–Tree

Generalized Fat–Tree networks are used by some commercial machine such as CM–5 and CS–2; in figure 3.a we can see the network topology for a generalized fat–tree build with 4 way by 4 way switches (these are the most popular building blocks for such a network). Usually over these networks an adaptive routing strategy or a random[1] one is implemented and the existence of multiple paths between each pair of node makes them very powerful [Lei85]. The mapping strategy for the kind of systems we choosed is very simple: the process network shape is just a "segment" of processors with input from the leftmost one and output from the rightmost. *It is possible to prove that using the templates proposed here we are able to plan message routing at compile time in such a way to have no network conflicts [Pas94]*; a machine like CS–2, where random routing is implemented through the use of tables in the message source node, can use the network as a reconfigurable structure and avoid every conflict between messages.

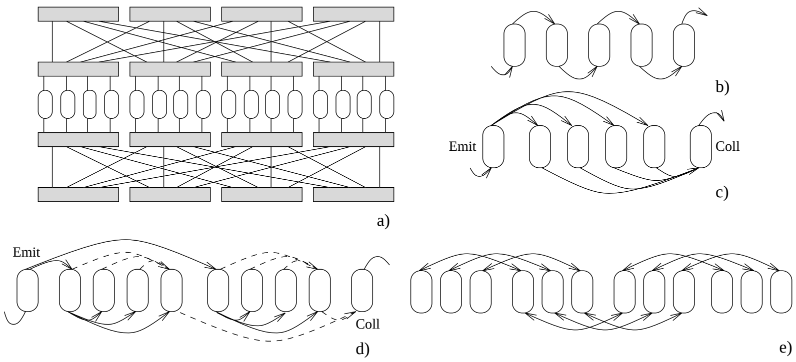

Figure 3: Templates for dual generalized Fat–Tree network.

Pipe mapping strategy is very simple: stages are putted one after the other and communication happens alternatively through the upper and the lower network. We can see the resulting shape in figure 3.b.

Farm template consists of a processing node that implements the emitter, one that implements the collector and a set of workers in the middle of these two processing nodes. Emission and collection communications happen on two different networks to avoid link conflicts between messages of different types (see figure 3.c).

[1]For example it's possible to use source randomization of message path, as in the CS–2, or random choice of the exit port at each switch until a common ancestor of source and destination is reached, as in the CM–5.

Map implementation consists in a hierarchy of farm–like structure, one for each dimension. The resulting template behaves exactly as predicted by analytical formulas (see figure 3.d for an example of 2D map). It is also simple to show that data distribution always happens in pipeline and so $g_p^b(\overline{D}, i) = 1$

Geometric workers are laid out one row after the other, one dimension after the other; during data loading phase communication happen exactly like in map template while data shifts, used in stencil gather implementation, work as in the pipeline template with communication placed alternatively on the upper and on the lower net (see figure 3.e for a 2D geometric horizontal shift). With this kind of interconnection structure is possible to do data shifts in parallel in both directions, so g_d is 1, and is simple to show that different messages do not conflict due to network redundancy of paths and especially for the availability of two networks so also $g_p^u(\overline{D}, i)$ is 1.

7 The analytical model validated

To validate the precision of this analytical model we measured the parameters of some commercial parallel architectures (see appendix A) and wrote a set of benchmarks composed of a single template and tested them extensively comparing measured and predicted times. Figure 4 reports the values obtained in one of these tests for the 2 dimensional map template on a Cray T3D. The error values are quite high because the template does not balance the computation load and the test program is quite unbalanced.

map size in PE	predicted performance	measured performance	error
2x2	8.18288	9.5	13%
3x3	3.76883	4.3	13%
4x4	2.2246	2.6	14%
5x5	1.51041	1.8	16%
6x6	1.12294	1.3	13%
7x7	0.88972	1.0	12%

Figure 4: Comparison between predicted and measures service time for 2D map on Cray T3D.

If we examine a similar table relative to a load–balancing template (see figure 5) we can see that all the error values are limited.

To validate the correctness of the model with construct composition and nesting we wrote a number of testing applications[PV96] and compiled them with our compiler for a Meiko CS-1 transputer based system. The test programs are:

DPR Data parallel ray-tracer, a simple parallelization of a sequential algorithm obtained mapping it onto every image pixel. The testing scene size is 400 by 400 pixel with 20 objects and 10 lights.

DFR Data flow ray-tracer, another parallel version of the previous program obtained by a data flow analysis of the sequential code. Same testing image.

number of workers	predicted performance	measured performance	error
2	15239	14945	1%
4	7619	7373	3%
6	5079	4946	2%
10	3047	3070	0.7%
15	2435	2642	7%
20	2435	2658	7%

Figure 5: Comparison between predicted and measures service time for farm construct on Cray T3D.

program name	predicted performance	measured performance	error
DPR	23.5	26.5	11%
DFR	0.0012	0.0013	7%
CFL	0.068	0.07	2%
CAT	23.5	25	6%
DEL	19.2	20	4%

Figure 6: Comparison between predicted and measured service time for test applications (all times are in seconds).

CFL Circuit fault list analysis, about 1000 gate circuit with 1600 possible faults.

CAT Computer aided thermography image analysis, 32 by 32 pixel images.

TDE Three dimensional delaunay triangulation, 1024 point dataset.

Figure 6 reports the results obtained from runs of the test applications and compares them with predicted performance values. The values are relative to process network service time.

The absolute performance of all these programs is usually very high, for example to compute each image DPR takes 140 seconds of CPU time on an HP 9000/735 workstation with 64Mbytes of RAM and just 27 seconds with 40 T800 (recall that Transputers are 10 times slower that HPPA processor). As we can see, the error in the analytical prediction of the service time is very low.

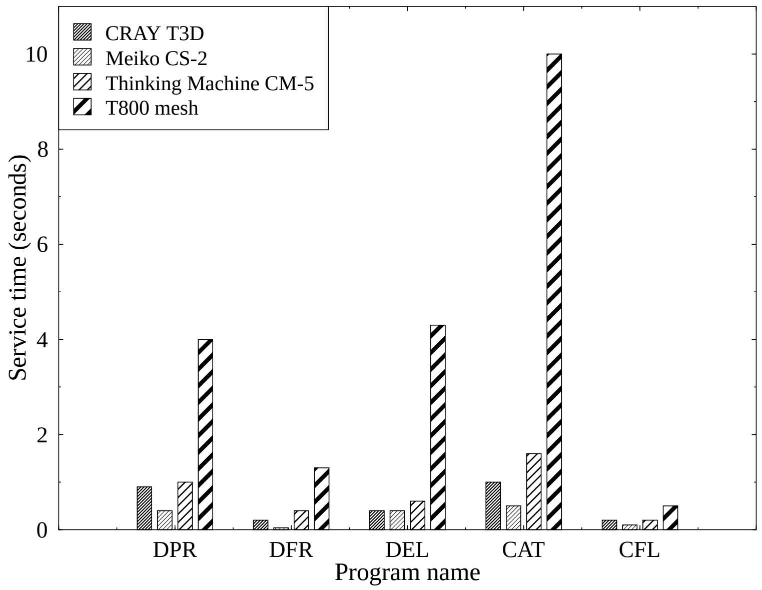

Figure 7: Testsuit programs minimum service time over different parallel architectures.

A comparison between the predicted service time of the various programs over some parallel architectures is reported in figure 7; note that circuit fault list service time is multiplied by 100 and dataflow ray-tracer by 1000 to make the graph visible.

8 Conclusions

In this paper we show an architectural model that can be used during the development of a template–based compiler to build formulas that describe the performance of parallel construct in an architectural independent way.

First of all we described the basic forms of parallelism we are interested in; along with each form we introduced an high level implementation strategy. Then we modeled our target machines using a (quite large) set of parameters and functions aimed at accurately predict performance and studied to be used by a compiler and not directly by the user. We wrote the analytical performance of each form of parallelism when implemented using the high level template previously introduced. Along with each formula we reported the optimizations the compiler is able to do just operating onto the template analytical model. Obviously other implementation strategies can be studied and added to the template library available to the compiler, provided that their performance is modeled using the proposed parameters and the possible optimizations are formalized. We showed also the inner working of the template–based compiler we developed as part of the P^3L project at the University of Pisa and finally we validated the model proposed here using a set of test applications and several different parallel architectures.

This model is quite complicated and covers several aspects of machine hardware, such as network speed, coprocessor hardware, internal processing node parallelism, etc.. This model is not intended to be used directly by users, instead it has been studied to be inserted inside a template–based compiler for a structured parallel language. Its complexity, much higher that other performance models like LogP, can achieve a high degree of precision in predicting parallel structures performance and allows for several program optimizations. Starting from the high level implementation strategies described here, tool writers can study other high level implementation template for the same basic forms of parallelism and let the compiler choose the best one for a given application on a given architecture.

A Model parameters for real machine hardware

We studied and benchmarked the following machines[2]:

Cray T3D [Cra93] system with 3D torus interconnection network, each PE contains a 150MHz alpha microprocessor and its network interface implements a physically distributed logically shared memory. System performance can be described as follows:

$$\begin{aligned}
\mathcal{P} = &\quad T_c + T_{in}^{com} + T_{in}^{ov} + T_{out}^{com} + T_{out}^{ov} \\
T_{snd}(l) = &\quad 0.0187l \\
T_{rcv}(l) = &\quad 0.0375l \\
T_{snd}^{ov}(l) = &\quad 70 \\
T_{rcv}^{ov}(l) = &\quad 70
\end{aligned}$$

Meiko CS–2 [Mei93] PEs contain a 40MHz SPARC microprocessor connected to a dual generalized fat–tree network; network interface is a programmable communication processor able of doing parallel input and output communications. The coprocessor can behave like a direct memory access engine so inter processor communications can be implemented through the use of memory mapped channels and direct message writing in destination nodes.

$$\begin{aligned}
\mathcal{P} = &\quad \max\{T_c, T_{in}^{com}, T_{out}^{com}, T_{in}^{ov} + T_{out}^{ov}\} \\
T_{snd}(l) = &\quad 0.011l \\
T_{rcv}(l) = &\quad 0
\end{aligned}$$

[2] All times are in microseconds.

$$T^{ov}_{snd}(l) = \quad 35$$
$$T^{ov}_{rcv}(l) = \quad 25$$

Connection Machine 5 [LA+92, KTR93] has 33MHz SPARC–based PEs and uses a dual generalized fat–tree network. Its network interface consists in a memory mapped FIFO queues; packets up to 16 bytes of data can be sent in about 2000 nS.

$$\begin{aligned}
\mathcal{P} &= T_c + T^{com}_{in} + T^{ov}_{in} + T^{com}_{out} + T^{ov}_{out} \\
T_{snd}(l) &= (\lceil \tfrac{l}{16} \rceil + 1)2 \\
T_{rcv}(l) &= (\lceil \tfrac{l}{16} \rceil + 1)2 \\
T^{ov}_{snd}(l) &= 50 \\
T^{ov}_{rcv}(l) &= 50
\end{aligned}$$

References

[AmKV93] Vikram S. Adve and mary K. Vernon. Performance Analysis of Mesh Interconnection Networks with Deterministic Routing. Technical Report 1001b, University of Wisconsin–Madison, Computer Science Department, July 1993.

[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.

[BDO+94] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. The P^4 approach to massively parallel programming. In *Proc. of COMPAR 94–VAPP VI*, September 1994.

[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.

[BYA89] Laxmi N. Bhuyan, Qing Yang, and Dharma P. Agrawal. Performance of Multiprocessor Interconnection Networks. *IEEE Computer*, pages 25–37, February 1989.

[CKP+93] David E. Culler, Richard Karp, David Patterson, et al. Logp: Towards a realistic model of parallel computation. In *Proc. of Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[Cok91] R. S. Cok. *Parallel Programs for the Transputer.* Prentice–Hall, Englewood Cliffs, New Jersey, 1991.

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

[Cra93] Cray Research Inc. *Cray T3D System Architecture Overview*, September 1993.

[Dal90] Willian J. Dally. Performance Analysis of k–ary n–cube Interconnection Networks. *IEEE Transactions on Computers*, 39(6):775–785, June 1990.

[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.

[DPV94] M. Danelutto, S. Pelagatti, and M. Vanneschi. An analytic study of the processor farm paradigm. Technical Report TR–3/132, Progetto finalizzato C.N.R. "Sistemi Informatici e Calcolo Parallelo", sottoprogetto "Architetture Parallele", May 1994.

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

[Hey89] A. J. G. Hey. Reconfigurable transputer networks: practical concurrent computation. In R. J. Elliott and C. A. R. Hoare, editors, *Scientific Applications of Multiprocessors*, pages 39–54. Prentice Hall, 1989.

[KTR93] Thomas T. Kwan, Brian K. Tatty, and Daniel A. Reed. Communication and Computation Performance of the CM–5. In *Supercomputing '93*, pages 192–201, November 1993.

[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.

[LA+92] Charles E. Leiserson, Zahi S. Abuhamdeh, et al. The Network Architecture of the Connection Machine CM-5. Extended Abstract, July 1992.

[Lei85] Charles E. Leierson. Fat–Trees: Universal Networks for Hardware–Efficient Supercomputing. *IEEE Transactions on Computers*, C–34(10):892–901, October 1985.

[Mei93] Meiko World Incorporated. *Computing Surface 2 reference manuals*, preliminary edition, 1993.

[OV93] S. Orlando and M. Vanneschi. Compiling Data-Parallel Paradigms through Basic Communication Mechanisms. Technical Report HPL-PSC-93-56, Hewlett Packard Laboratories, Pisa Science Center (Italy), September 1993.

[Pas94] Davide Pasetto. Sviluppo di una Libreria di Template per Architetture ad Elevato Parallelismo. Tesi di Laurea, Dipartimento di Informatica, Università di Pisa, Italy (in italian), July 1994.

[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.

[PV96] Davide Pasetto and Marco Vanneschi. Design and evaluation of parallel applications using a structured parallel language. Technical Report TR-96-06, Department of Computer Science, University of Pisa (Italy), March 1996.

[RS+93] E. Rosti, E. Smirni, et al. The KSR1: Experimentation and Modelling of Poststore. Technical report, Dept. of Computer Science of Vanderbilt University, 1993.

[SGC93] Rafael H. Saavedra, R. Stockton Gaines, and Michael J. Carlton. Micro Benchmark Analysis of the KSR1. In *Supercomputing '93*, pages 202–213, November 1993.

[WB+92] Daniel Windheiser, Eric L. Boyd, et al. KSR1 Multiprocessor: Analysis of Latency Hiding Techniques in a Sparse Solver. Technical Report UCB/CSD92/#675, Computer Science Division, University of California, Berkeley, March 1992.