Commission of the European Communities

****

ESPRIT III

PROJECT NB 6756

****

CAMAS

COMPUTER AIDED MIGRATION OF APPLICATIONS SYSTEM

****

CAMAS-TR-2.1.1.5
Technical Report
Presentation on the Status of Parasol

(output month 24)

****

Date: November 1994 — Review 4.0

ACE - Univ. of Amsterdam - ESI SA - ESI GmbH - FEGS - PARSYTEC - Univ. of Southampton

University of Amsterdam

A. de Mes	mes@fwi.uva.nl
P. M. A. Sloot	peterslo@fwi.uva.nl

Contents

List of Figures

List of Tables

Chapter 1

Goals for Parasol I

1.1 Execution Time Estimation for Parallel Programs

Quoting for Parasol I Technical Report 2.1.1.4:

> The goal of the Parasol project is make a rough estimate of the execution time of a program, given the program source code and the machine that will execute the program. This estimate must be given in a very short time, short enough to call the estimators response 'interactive'.

Using a step by step trace through the parallel code, whilst simulating the state and reactions of the parallel computer will give the most accurate description of the loads and result in very exact estimate of the execution time needed for real execution. However, this type of simulation is too time consuming for practical purposes. If a programmer is to use a tool during the porting process, useable results will need to be presented fast. To determine the optimal configuration for the parallel computer, the programmer might want to change parameters that describe the computer. The tool should also provide that capability.

To analyse a parallel program, the program is split into three basic levels. The lowest level consists of basic blocks. These are groups of sequential data without jumps, loops or entry points. One level higher, the flow control is introduced. With this second level, complete sequential programs can be expressed. The third and highest level is the communications level. At this level processors can synchronize, send, receive, and broadcast.

The application on which the tool is being applied needs to be analysed and split into these three levels. The lowest level has a straight forward time behaviour. The time needed to execute such a basic block is the sum of the time needed for each of the instructions in the block. The second level introduces more complexity; the repetitions of a loop depend on input values to the program and on operations done with the input from the basic blocks. The third level increases complexity even more by introducing communication networks with delays, limited throughput, startup time for messages, and saturation.

An enormous speedup in estimating the execution time of a sequential program for various data input sets can be achieved by rewriting the lowest level (basic blocks) into expressions where instructions are represented by the time they need to execute. The times needed for those instructions are collected in a machine model. A few queries to this machine model will suffice to make a reasonable estimate of the time needed to execute these basic blocks, created in the previous step.

The second level can be represented by supplying multipliers around the first level expressions. For the type of applications that need to be modelled by PARASOL, the values for these multipliers depend upon the trace of a few variables rather than the trace of all the variables. Here another speedup is accomplished against simulation.

The third level is more of a problem, as network behaviour can be the rise or fall of a parallel application. Routing plays a crucial role in determining the load of a link. Once information on the state of the network in all phases of the program is available, combining it with the expressions derived in the first and second level will produce the final estimate of the execution time of the parallelized application.

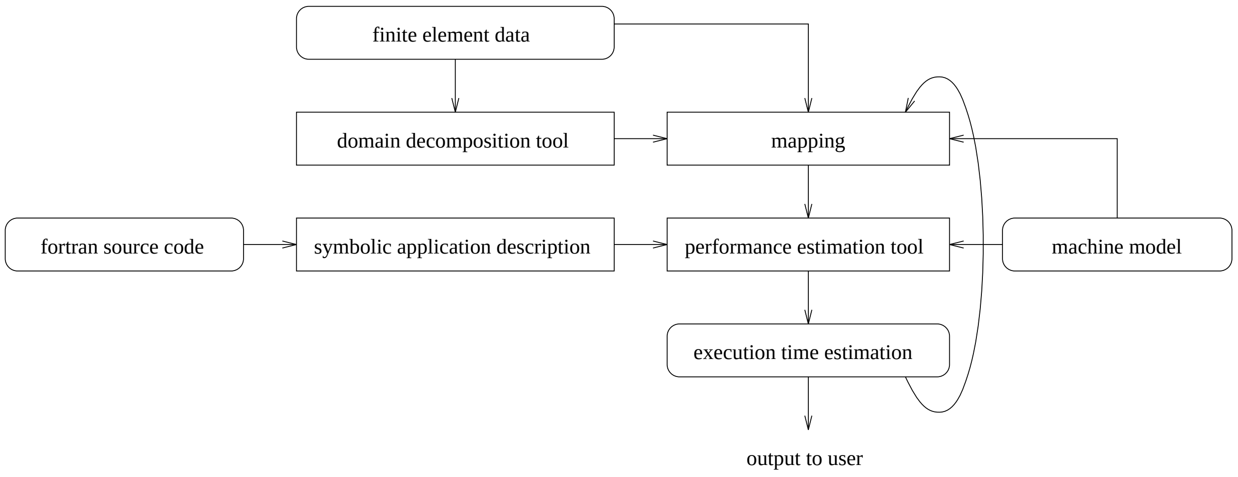

finite element data
domain decomposition tool
mapping
fortran source code
symbolic application description
performance estimation tool
machine model
execution time estimation
output to user

Figure 1.1: A graphical layout of the tools in the project

The basic setup of the different tools and their interdependencies are illustrated in Figure 1.1. The 'finite element data' is the input data for the Fortran program that is being analysed. The separate elements from the input data are gathered in x groups by the 'domain decomposition tool', where x is the number of processing nodes available. Using the topology information from the machine model and possibly the original finite element data, these x groups are mapped on the x processing nodes in the machine.

In the lower left part of the graph, the Fortran program is analysed and translated into a 'symbolic application description'. This SAD is fed into the 'performance estimation tool'. The PET combines information from MAP, SAD, and the 'machine model' (from here on PARASOL I) and estimates the execution time needed for the real program to run on the real machine with the given data. The part of the PARASOL project that will be referred to as PARASOL II is a combination of SAD and PET. SAD as an application description has three hierarchial layers: basic blocks, flow control, and communication behaviour.

1.2 Objectives

A brief summary of the objectives:

1. An abstract machine description is to be presented.

2. The machine description should be of use for interactive estimation of trends in execution time.

3. The model should be able to represent:

   (a) Virtual machines. These virtual machines can be derived by simulation of other virtual architectures, or by ‘what-if’ situations that result from the user changing parameters in a user interface.
   (b) Several real machines. Parameters for real machines are to be obtained using benchmarking routines. Two machines are to be represented as part of the project: the Parsytec GCel and the FWI csys SPARCstation cluster.

4. The benchmarking routines should be portable. They should be able to run on any machine, on which the application that is to be modelled, can run.

5. Since the applications that need to be modelled, are written in Fortran:

   (a) The machine will be represented by means of Fortran specific parameters. Due to this, the model becomes machine independent and the compilation phase is omitted.
   (b) Topologies will be modelled at the common level available to Fortran. Unfortunately, this is machines specific.
   (c) SPMD supporting run time systems will be modelled at their respective call levels. Their implementation and communication activities will be represented in terms of other machines parameters, also in the model.

1.3 This Document

This document describes the design issues, implementation issues, results and future work of the PARASOL I project. The integration with other tools such as PARASOL II is also discussed.

First the requirements of the machine model are discussed, followed by the model used for PARASOL I. Its advantages and disadvantages are presented and possible imperfections are discussed. In the next sections the implementation issues concerning the file format, graphical interface and inter-program interaction are brought forward. Problems and aspects of acquiring accurate timings needed for the model and the validation thereof are presented in the next section and this document ends with conclusions and suggestions for future work in area of research.

Decomposition of Fortran programs is not further detailed in this document, as it is beyond the scope of the assignment. For further reading regarding this topic, refer to [Halderen94].

Chapter 2

The Machine Model

2.1 Requirements of the Model

The machine model used for PARASOL is integrated into both PARASOL I and PARASOL II. Therefore the abstraction level of the machine model should be such that most parallel computers can be parameterized. It should not be necessary to have to rewrite code when specifying a new machine. The information stored in the model however should not hold the amount of detail that would imply execution traces or even microcode traces. It should be useable for estimating execution time of uncompiled Fortran code. Its elements should be related to Fortran expression syntax and functions.

2.1.1 Interactive Interface

One of the primary goals of the PARASOL project was to be able to use virtual – non existent – machines as input for performance estimation. The programmer would be able to try the ported application on 'what-if' machines. Changing a few parameters might lead to a totally new behaviour of the application.

A graphical interface presents the current machine to the programmer. It provides insight to the description of the current machine and simplifies modifying parameters in the model. As an extra, it is possible to change groups of elements. One group of elements, for instance, is the group of flow control instructions. Specifying a delay for this group might symbolize a less efficient implementation of the instruction pipeline.

2.2 The Used Model

The machine model chosen, is one consisting of three parts, each with its own abstraction level. It is safe to state that a parallel computer always consists of processor entities, some form of network that interconnects these entities, and that there is some higher level interface available for writing portable code. For PARASOL a few other assumptions are made:

The application is an **SPMD** program. The processors in the parallel computer are all in the same abstract state at one point in time. Such an abstract state can be *calculating* or *communicating*. Due to this, the model does not need to represent delay in calculation speed during communications.

The parallel computer has at most **one instruction cache level**. The processor cache is modelled by one speedup index. This index can be used if a block of code is smaller than a size predetermined for the machine. Effects of data caching are not explicitly present in the model (see section 2.2.1).

The routing of data across the communication channels is a passive method, i.e. the route for a chunk data does not depend on the network load. This is not a problem for the real parallel computers intended to be modelled. The Parsytec GCel has a simple XY routing strategy and the FWI csys SPARCstation cluster has an even more straightforward routing: Ethernet broadcast.

2.2.1 Model Part 1 - The Processor

The model provides for more than one type of processor in the machine. This is realistic. For example the FWI csys SPARCstation cluster does not contain one type of machine but several types of workstations from the SPARCstation range. Those workstations contain different processors from different stages in the SPARC processors evolution. Also, the memory size and cache size vary from workstation to workstation.

Each type of processor in the machine is described separately. A processor description consists of 150 timings, which describe most of the Fortrans intrinsic functions, expression operators and flow control instructions. Where appropriate, a timing is present for every data type of the function/expression. Also in the processor description is a cache specification, that specifies the cache size, speed and speedup factor.

Measured Processor Parameters

Fortran expressions can contain several infix operators, such as addition (+), multiplication ($*$), power ($**$) and several others. Each of these functions can operate on different numeric types and the values they operate on can be in global or local variables. For illustration purposes, all addition (+) parameters are listed below:

addition	location of variables	
	global	local
complex	cga	cla
double	dga	dla
real	rga	rla
integer	iga	ila

Table 2.1: Infix addition parameters

The following infix operators are included in the model (on all four numeric types):

- addition (behaviour is assumed identical to subtraction)
- multiplication
- division
- power with trivial exponent (small integer number)
- power with non-trivial exponent
- assignment
- memory transfer

This set corresponds to instructions used by [Saavedra89]. The distinction between globally and locally defined variables needs to be made as is evident from timing results presented in Chapter 4.1.

Fortran has many intrinsic functions. They are commonly used in applications involving numeric algorithms. These are exactly the type of applications for which the PARASOL project will be most frequently used, so the machine model should be very meticulous in specifying execution times for these instructions. In Fortran these functions use the same amount of time for local as for global variables, due to the way they are commonly implemented. In the PARASOL I machine model one timing is specified for every intrinsic function for every numeric type it operates on. The intrinsic functions are listed below:

Table 2.2: Intrinsic functions in the PARASOL I model

intrinsic	numeric type				
function	complex	double	real	integer	action
abs	●	●	●	●	absolute value
aimag	●				imaginary part of number
aint		●	●		truncation
anint		●	●		nearest whole number
asin		●	●		arcsine
atan		●	●		arctangent
char				●	to character
cmplx	●	●	●	●	to real part of complex
cmplx2		●	●	●	two numbers to complex
conjg	●				conjugate
dble	●	●	●	●	to double
dim		●	●	●	positive difference
dprod		●	●		double product
exp	●	●	●		exponential
int	●	●	●	●	to integer
log	●	●	●		natural logarithm
log10		●	●		common logarithm
min/max		●	●	●	choose extreme value
mod		●	●	●	remainder
nint		●	●		nearest integer
real	●	●	●	●	to real
sign		●	●	●	transfer of sign
sin/cos	●	●	●		sine / cosine
sinh		●	●		hyperbolic sine
sqrt	●	●	●		square root
tan		●	●		tangent
tanh		●	●		hyperbolic tangent

There are several more parameters of the machines processors that are included in the model. This set contains logical operators, loop overhead, procedure call behaviour, other flow control statements, and array indexing. A complete list is presented in appendix B.

The Processors Instruction Cache

The processor cache is modelled using three parameters:

- The *cache size* specifies how many average size Fortran instructions fit in the instruction cache. Some decision needed to be made as to what is an average instruction size at Fortran level. The instruction chosen to be average size is the multiplication of two locally defined double precision numbers.

The assembler code generated to achieve this multiplication is larger than local integer addition and smaller than global complex division.

- The *cache speed* is not intended to be used due to the next cache parameter: small loop speedup. Nevertheless, it specifies the time needed to retrieve one word (the bandwidth) from cache.
- The *small loop speedup* indicates the cache speedup factor. The processor parameters are represented as time needed to execute them, when they are not in instruction cache. In loops smaller than the size of the instruction cache, each of the instructions in the loops will be executed faster due to the decreased fetch time. The time listed for a processor parameter divided by the 'small loop speedup' is the time required for an instruction in that was fetched from instruction cache.

Applications that use the parameters from this model, such as PARASOL II, should use the cache specification from PARASOL I in the analysis of Fortran programs as follows. When the analysis leads to the conclusion that a specific instruction at a specific place in the source has been executed less than 'cache size' instructions before, it is likely to be in the instruction cache. Therefore the fetch will be faster than normal, leading to faster execution of the instruction. To calculate how much time is spent, executing that instruction, the time specified in the model should be divided by the 'small loop speedup'. A small example is given in section 3.1.9. If that speedup is 1.0, the effect of no instruction cache is mimicked. Instructions in small loops are likely to all be in instruction cache, thus the name of the parameter.

Presence of data cache can not be represented in the PARASOL I model. Because the timings in the model incorporate argument fetch times, the instruction and its parameters are combined, it is difficult to make distinctions. Furthermore, in the final executable code, these arguments can be in registers, data cache or memory. Where the arguments, needed for an instruction, are, depends on the source code, the compiler, the operating system (are application programs permitted to operate the cache?) and the size of the data cache. Because compiler actions are not modelled, there is no way to determine when data is in registers.

Combined Processor Parameters

To get a quick impression of the performance of a processor, several ratios are interesting to observe. In this section of the processor parameters a few of these ratios are defined.

The ratio of calculation over communication throughput gives insight into the capacity and computing power. Better arithmetic performance leads to a lower value for the ratio while a better communications performance leads to a higher value for the ratio.

The ratios that have been found useful are:

- time for integer addition over send time per byte (throughput)
- time for integer multiplication over send time per byte
- time for double precision addition over send time per byte
- time for double precision multiplication over send time per byte
- time for double precision division over send time per byte
- time for double precision sine calculation (intrinsic Fortran function) over send time per byte
- time for double precision power (**) calculation over send time per byte

The parameters listed in the machine model clearly need to be combined. This can be done using expressions. They are implemented in PARASOL I and can refer to any element of the specified model, while using every common arithmetic function to express the relations. For illustration purpose, integer addition over send time per byte will be examined in greater detail. There are two timings for integer addition stored in the model: for locally and for globally defined arguments to the addition. Here a design decision is made and global arguments are chosen. This "global integer addition" is specified in the model in the form of the execution time needed for the instruction. The send time per byte (throughput) is specified in the topology section as 'link speed'. Both values are timed is seconds, so they may be used directly. Result:

$$ratio = \frac{time_{integer\ add}}{time_{send\ byte}}$$

An example of how this will look as model parameter is given in section 3.1.3. Presenting the resulting value in a user interface is of help to the user in assessing the capabilities of the processor. Comparing the ratios to those of other processors gives direct perception of the balances between the capabilities of the two.

2.2.2 Model Part 2a - The Topologies

The model provides for two types of topologies: hardware and virtual. The distinction is in the link specification. For the hardware topologies a specification is given of every individual link. The virtual topology however is specified by giving a per-processor mapping of the virtual processor numbers to the hardware processor numbers.

Hardware Topologies

The real topology, as implemented in hardware will always specify the upper bound of the maximum transfer rate and corresponding latency. This topology must be represented in an exact way to be able to reconstruct communication behaviour. The hardware topology description in the PARASOL I model contains the following elements:

- Specification of the processors: each processor in the topology is assigned a processor type. These types are defined in the processor section of the same machine description.
- Specification of the hardware links: for each physical link the following is specified:
  - The latency for message passed over this link. This is the time used between the start of send call (on the lowest common programming layer – not assembly level) and completion of receiving the message. This message will have the minimal size to avoid combining throughput with this parameter.
  - The transfer rate for this link. It is specified in time needed to send one byte.
  - The connectivity of the link. A list is given of the processor numbers connected to this link. Any positive integer number of processors may be connected.
- The routing technique used for this topology. This will be explained in more detail later in this section.
- Amount of time needed to setup the connections for the entire topology. This is a delay typically caused at the startup of an application. The time needed to notify other processing nodes that a new node was added to the computing pool is a common example.

The decision to represent the hardware topology in this way was mainly influenced by the need to specify such fundamentally different topologies. The FWI csys SPARCstation cluster is one extreme (just one link – an Ethernet cable) while the Parsytec GCel is an opposite extreme (976 links with much simpler behaviour).

Virtual Topologies

Virtual topologies would be nothing without implicit routing strategies. In the PARASOL I model these two are split up nevertheless. In the virtual topology definitions in the model, the mapping of the virtual processor numbers on the hardware processor numbers are specified. The routing mechanism for messages is entrusted to an external routing mechanism. Thus the virtual topology description contains the following elements:

- The routing mechanism (explained later in this section).
- The time needed to setup the connections.
- A mapping of the virtual processor numbers to the hardware processor numbers.

Routing Mechanism

Routing mechanisms come in different shapes and sizes. There does not exist a single parameterized model that captures all routing algorithms. They are described with (pseudo) code or using textual explanations. Thus providing a routing parameter for hardware and virtual topologies is a problem. The parameter chosen in PARASOL I for routing is to provide a text string with in it the name of the routing strategy used. This string is matched with a routing mechanism that is implemented in C-code. A library of these routing mechanisms should become part of the PARASOL I code. The requirements that needed to be met for the real machines represented are discussed.

For the FWI csys SPARCstation cluster, the problem of routing mechanisms is trivial. The Ethernet (IEEE 802.3, see also [Boggs88]) that interconnects the processors is a broadcast network. If a packet is successfully sent, all processors that want to receive the packet do so. Contention is not modelled, it is abstracted to a maximum throughput. Virtual topologies implemented on top of this sort of system are cheap.

In the Parsytec GCel, the T805 Transputers are connected by means of a two dimensional hardware grid, see [Parsytec93]. Messages on the hardware grid are routed using XY routing. The message is first sent off in the X direction and when it reaches the matching X coordinate, it proceeds in the Y direction until it reaches its destination. For example, consider a message being sent from node A in Figure 2.1 to node C. The message will travel straight to node B and from there up to node C.

In the PARASOL I model, the routing attribute will have the value **XY**. When routing information is requested by a client, pre-compiled C code will interpret the link descriptions and provide the routing information to the client.

On this level PARASOL I is adjusted to serve the specific needs of PARASOL II. PARASOL II attempts to simulate the state of the network in discrete time-steps. The information exchange between the two parts of PARASOL is done as follows:

I. PARASOL II analyses the provided Fortran code and determines what nodes will be sending, to whom they will be sending, and how much they will be sending.

II. The message descriptions resulting from this analysis are sent – per message – to PARASOL I. Here the routing mechanism is invoked and a route is plotted. The plotted route is stored temporarily and a unique integer number that points to this route is returned to PARASOL II.

III. PARASOL II then starts an event trace to determine which messages are being sent. This is done in one time-step per change in the network load.

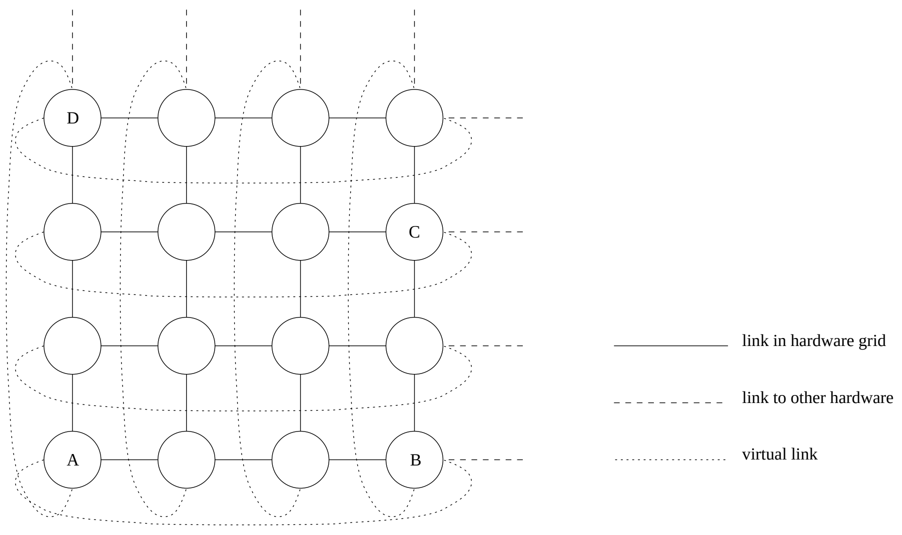


Figure 2.1: Example topology for 4 × 4 grid

IV. The set of messages (route identifiers received earlier) that are being sent at a certain time is sent to PARASOL I, along with a request as to how much time will be needed to complete one of those messages. The set of route identifiers is combined into a network state and the amount of bandwidths available for the requested message is calculated. After all the bandwidths on the route are determined, the transfer time of the message can be determined.

V. The transfer time is returned to PARASOL II, which then calculates how large it can make its next discrete time-step. At this point several design decisions for PARASOL II start to play parts. For further details see [Halderen94]. In normal cases, it will cycle back to step **IV** for every time-step.

This sequence of steps is actually an event trace. Though the main intention of the PARASOL project was to not use simulation, it was judged that this was the most elementary form of network modelling that would achieve any significance. To assess the time needed for the transfer of a message, the number of messages being multiplexed over every link in the route, from sender to recipient, needs to be determined. The client (PARASOL II) is not required to know anything about topology or routing. The many steps listed above are needed to separate the evaluation of the routes and the evaluation of the network states. This achieves a noticeable speedup in comparison to leaving routing and network state evaluation integrated, because the client is satisfied with one route, while it may wish to have many different messages combined into network states.

Virtual topologies are always implemented on top of a hardware topology. The routing of a message sent from one node to another in a virtual topology is done, in the most efficient way, by leaving the routing to the hardware. The gain in using virtual topologies, is in the mapping of the – new – virtual processor numbers on the hardware processor numbers in the most efficient way.

In Figure 2.2 an example mapping is given for a ring topology. The numbers in the nodes, are the processor numbers in the virtual topology. It is clear that this mapping is an optimal mapping. Every virtual link is mapped on a minimal route in the hardware topology (one link). For the Parsytec GCel, the mapping algorithms were and are still being developed by [Röttger94].

PARASOL I parameterizes virtual topologies by a mapping (earlier in this section) and a routing algorithm

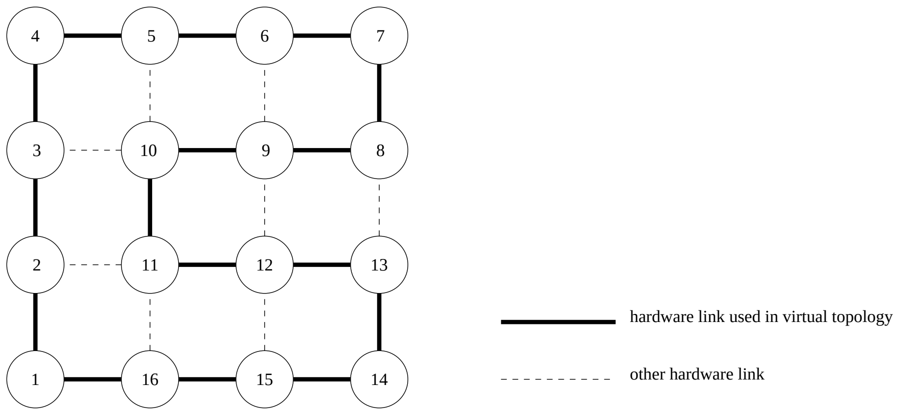


Figure 2.2: Example mapping of a virtual ring topology on 4 × 4 grid

that fits on top of the hardware routing algorithm. This second routing mechanism is used to describe what virtual links exist in the virtual topology. For instance, if a two dimensional torus virtual topology were projected on top of the 4 × 4 hardware grid in Figure 2.1, eight virtual links would need to be added to the hardware topology. They are drawn as dotted lines in the figure. Node A would become virtually connected with nodes B and D for instance. Note that the mapping of the torus might be done differently than the 1 to 1 mapping in the figure.

The virtual topology routing parameter (here **2Dtorus**), refers to a routing mechanism in C-code that implements all the links available. Here, that would be $(4 \cdot 16 =)$ 64 links.

2.2.3 Model Part 2b - The Machine Combinations

Just like the combined processor parameters presented in section 2.2.1, topology parameters too can be combined to form ratios that give quick impressions of the performance. [Markatos92] presents the sum of the maximum throughput for all the processors as a useful index for the communications performance. FLOPS and MIPS are usually specified by manufacturers as an index to the performance. Such indexes are determined using hardware parameters as clock rate and design aspects for the ALU and floating point unit. The PARASOL I model models a machine at Fortran level and attempting to determine the FLOPS or MIPS rate from these parameters would result in a seriously distorted figure that could not be compared to respective rates by manufacturers.

The sum of the maximum throughput for all the processors is the only combined topology parameter that is included in the model. Like the combined processor parameters, this parameter is represented using an expression. For an illustration on how a parameter is represented in the model for the Parsytec GCel refer to section 3.1.4.

2.2.4 Model Part 3 - The SPMD Supporting Run Time Systems

For portability reasons, standard parallel run time systems are being used more and more often. Once a program is written using one of these run time systems, it should be able to run on all parallel computers for which the system is available. This is considered a great benefit. PARASOL implements three of such run time systems: PVM, Express, and MPI.

Not every program written using one of these systems can be handled. In section 2.2 the SPMD restriction is explained. This restriction is mainly for the estimations done by PARASOL II and has little significance for PARASOL I. For PARASOL I the only argument, for requiring SPMD style programming, is that on one processor there is only one process running at any time. This to avoid incorporating multitasking into the timings and having to add parameters that model its behaviour.

For each of the three run time systems, primitives involving communications are parameterized in the machine model. They combine the behaviour of the implementation of the run time system with the communication behaviour of the machine. Therefore, parameters describing any such primitive must contain three basic components:

1. A description of the parameters passed to the run time systems primitive.
2. A description of the activities in the implementation of the primitive in terms of the processor parameters in the model.
3. References to the topology description to describe what delay is caused by the communication done.

These three components can be combined using common arithmetical expressions. PARASOL I implements arithmetic formulas and references to other parameters in the machine description.

2.2.5 Advantages of the Used Model

Any model of a machine will have its advantages and disadvantages. Most of the advantages are direct consequence of the objectives of the PARASOL I project (section 1.2). Advantages of the machine model presented in this paper are discussed in this section.

Fortran Level

The machine is parameterized at Fortran level. By looking at the code at source level, the used benchmarking routines become machine independent. Transferring these benchmarking routines to other run time systems is almost trivial. After completing the benchmark routines for the processor parameters on the FWI csys SPARCstation cluster, only several hours were needed to make them run on the Parsytec GCel. Their sizes could be reduced due to the high precision clock that is available on the Transputer. Compiling and running these ported benchmarks took about six times longer – one day.

The only way compiled applications can be compared with the model, is when they are compiled with optimization switched off. This is not entirely realistic, but essential to the significance of the benchmarks. Every compiler has different optimization strategies and machine dependent tricks to increase performance.

Although not having to represent a machine specific model due to the use of Fortran is an advantage (section 2.2.5) that can not be overlooked, the consequence of missing out on compiler optimizations ensures that only trends in execution time can be obtained.

Intuitive Parameters

The parameters in the model are intuitive. Seeing the cost of a multiplication is much more comfortable than having to realize what fetch/decode/execute cycle is involved. Some old fashion Fortran compilers even implement multiplication with function calls.

Useable Without Compilation

The parameters can be used for performance estimation on code that has not passed through a compiler. Next to being an advantage, this is also a requirement (section 1.2, item 5a). Working with source code instead of compiled code has the advantage of not having to mimic compiler behaviour during the program analysis or having to execute the complete program to gather adequate trace information. Small pieces of code can be analysed without execution traces. This is the main idea behind the PARASOL project.

Attainability

Filling in the parameters is possible by benchmarking real machines. It does not depend on the manufacturers specification of the machine. Whenever designing a machine model, one should consider that all the parameters will eventually need to be assigned a realistic value.

Of course the model can be filled in other ways than with benchmark results, but for validation of the model real machines should be represented. In combination with the PARASOL II application, predicted results can be compared to real execution results – this ultimately validates the models correctness.

2.2.6 Disadvantages of the Used Model

Though no disadvantages of the model are guarantees for failure, a few drawbacks of the model are discussed here.

Abstraction Level

The (Fortran) abstraction level used is too high to produce exact estimations of execution times. It is adequate nonetheless for identifying trends in the execution time caused by fluctuations in input parameters of the application being examined, and the machine model parameters.

Non-Orthogonal Parameters

Many of the parameters depend on each other, while it is not specifically mentioned in the machine description. For instance the time needed to execute a floating point (real) number multiplication will have some relationship to the time needed to execute a double precision (double) multiplication. To solve this, the machine should be modelled at the lowest level. This is highly undesirable (as mention earlier).

Optimizations in the Processor

The arithmetic logic units (ALUs) in modern processors are optimized for handling easy numbers. Multiplying by one, or adding to zero are instructions that will typically execute faster than instructions with more complex arguments.

The power (**) function presents such a fundamental variation in execution time due to differing values of its arguments, that is is the only instruction that is represented in the model using two different parameters. It is represented and benchmarked for low integer values and for random floating point numbers. This is adequate for the machines currently benchmarked, but in the future may be required for more than one instruction. Instructions may even need to be split into more than two complexity levels.

SPMD not Fully Implemented

The PVM, Express, and MPI SPMD supporting run time systems have not been studied in enough detail yet. The model still needs to be extended to correctly represent the behaviour of the functions in these models. Also, not all of the models are present for the Fortran programming language, on which this model is based.

2.2.7 Other Considerations when Using the Model

Some considerations need to be mentioned. They should not be interpreted as disadvantages, but most of these points are listed as topics for further research in chapter 6.

Fortran Specific Model

The model is defined specifically for Fortran programs. Essentially, doing the same for C code would imply redesigning almost the entire model and benchmarks.

As a result of being Fortran specific and at source code level, the processor is parameterized using 150 elements, which all need to be measured separately. For a port to the C programming language, a whole new set of parameters would need to be determined.

Single Cache Layer

The processor cache can normally be split into two types; instruction cache and data cache. The latter is incorporated in the processor timings, which illustrates the problem of non-orthogonal parameters. The instruction cache is represented by a single index. This implies only one level of caching, but also presents the client level (PARASOL II) with a boolean choice of applying the index. This concept might become outdated when new parallel machines need to be represented.

Topology Specification is Elaborate

Specifying the hardware topology per link is quite elaborate. A user may loose all overview when for instance a collection of links representing a hypercube topology is presented. A correct metric for topology specification is not available at this time. For now the problem should be abstracted from by the user interface for the model. Essentially the information should be in a different, but yet unknown, format. This is a topic for further research.

Routing is done External

In addition to the elaborate topology specification, an adequate metric for specifying routing algorithms is absent (see section 2.2.2 for the current method). This too is a topic for further research. Currently it is implemented in C and identified by predetermined string. Adding new routing techniques requires familiarity with the existing code.

Chapter 3

Implementation

3.1 Disk Representation of the Model

The parameters for the machine are collected in one data structure, which can also be saved on disk. In this section the parameters of the machine model are presented and discussed, while traversing the BNF grammar for the file format. The complete BNF grammar for the file format is collected in appendix A.

3.1.1 The Sections

The various sections in the machine are described as follows:

machine	::=	**begin machine** *nl machfield*+ *machsection*+
machsection	::=	*procsection* | *mcombsection* | *topsection* | *virttopsection* | *pvmsection* | *expresssection* | *mpisection*
procsection	::=	**begin processor** *nl procspec*+ **end processor** *nl*
mcombsection	::=	**begin machinecombinations** *nl mcombspec*+ **end machinecombinations** *nl*
topsection	::=	**begin topology** *nl topspec*+ **end topology** *nl*
virttopsection	::=	**begin virtual** *nl virttopspec*+ **end virtual** *nl*
pvmsection	::=	**begin pvm** *nl pvmspec*+ **end pvm** *nl*
expresssection	::=	**begin express** *nl expressspec*+ **end express** *nl*
mpisection	::=	**begin mpi** *nl mpispec*+ **end mpi** *nl*

The start symbol for the machine description is *machine*. From within this main section the processors (*procsection*), the machine combinations (*mcombsection*), the topologies (*topsection* and *virttopsection*), and the SPMD supporting parallel programming environments (*pvmsection*, *expresssection* and *mpisection*) can be specified. Multiple processors and topologies can be defined, the other sections are expected once each. If the sections that are expected once are defined more than once, the last definition will be the one used.

Comments can occur on empty lines or after definitions. They start with a # character and last until the end of the line.

3.1.2 Global Machine Parameters

A few machine parameters do not fit in any of the above mentioned sections, or are considered ‘global’ to the machine:

machfield	::=	*id* | *name* | *proccount* | *topcount*
id	::=	**id =** *integer nl*
name	::=	**name =** *string nl*
proccount	::=	**processors =** *integer nl*
topcount	::=	**topologies =** *integer nl*

The *id*, *name*, *proccount*, and *topcount* fields are each expected to be defined exactly once. If defined more than once, the last definition will be the one used. The definitions for *integer* and *string* can be found in appendix A.

The *id* field defines a reference key for this machine. When applications working with the PARASOL I model refer to a machine, it should be done using this integral number.

The *name* field defines a character string name for the machine. This field is for interactive applications that wish to present the name of the machine.

The *proccount* and *topcount* fields are defined for reference purposes; references are described in section 3.1.10. The *proccount* field defines the number of processor types in the machine, it has no reference to the number of processors in the machine. The *topcount* field defines the number of (possibly virtual) topologies specified for this machine.

3.1.3 Processor Parameters

For each processor in the machine, parameters may be defined. These parameters are split up into four types; timings of instructions (and intrinsic functions), specification of the cache, combined processor parameters, and parameters that do not fit in the former three types. The latter is discussed first:

procspec	::=	*id* | *name* |
		begin timings *nl timings*+ **end timings** *nl* |
		begin cache *nl cache*+ **end cache** *nl* |
		begin combinations *nl combined*+ **end combinations** *nl*
id	::=	**id =** *integer nl*
name	::=	**name =** *string nl*

Within the processor specification, no parameters are expected to be defined more than once; if any of them is defined more than once, the last definition will be the one used.

The *id* and *name* parameters fulfill the same purpose for the processor, as they do for the machine in section 3.1.2.

Instruction Timings

The performance of a machine is greatly influenced by the performance of the processors in the machine. The most frequently executed instructions by Fortran programs are arithmetic instructions, flow control instructions, and intrinsic functions. Instead of presenting the BNF grammar for this section (which can be found in appendix A), the separate parameters are discussed here according to these three categories.

The **arithmetic** instructions commonly used in Fortran programs are addition (and subtraction), multiplication, division, power with integer exponent, and power with real exponent. For ease of use, assignments and memory transfers are also classified as arithmetic instructions. The seven arithmetic instructions listed above are present in the PARASOL I machine model for the following numeric types: integer, float, double precision, and complex.

The **flow control** instructions commonly found in Fortran programs are either simple goto statements, computed goto statements, or procedure call statements. The overhead introduced by loops is also classified as a flow control parameter.

The Fortran **intrinsic functions**, except for those that do character handling, are all specified in the model for each data type they operate on.

All the instructions listed are measured using benchmarks, this is described later in section 4.2.4. The results of these measurements are timings; the number of seconds required to execute the instruction. These include instruction fetch and parameter fetch. For representation of measured values see section 3.1.8.

Cache Parameters

The cache parameters are purely provided to specify a more complete model of the machine.

cache ::= **size** = *integer nl* |
speed = *timing nl* |
smallloopspeedup = *timing nl*

The cache speed is already incorporated in the instruction timings and provides little extra information for performance estimation. It does however provide for a better insight for humans as to how the machine is set up. The cache speed is the time needed to retrieve one word (the bandwidth) from cache.

The cache size is useful in combination with the small loop speedup factor. When estimating the performance of a program, the estimator might decide that the loop in the program is so small, it fits into the cache. It can then use the small loop speedup factor to see what happens to the performance. This was explained in greater detail in section 2.2.1.

The **smallloopspeedup** parameter defines the average speedup of instructions if they are present in the instruction cache. How this parameter is to be used is discussed in section 2.2.1. The BNF reference *timing* is discussed in section 3.1.8; it is a measured value.

Combined Processor Parameters

In this section parameters concerning the processor speed are compared to communication parameters. It is common practice to specify such ratios.

combined ::= *combparams* = *expr nl*
combparams ::= **ia_spd** | **im_spd** | **da_spd** | **dm_spd** | **dd_spd** | **ds_spd** | **dp_spd**

These parameters stand for integer addition, integer multiplication, double precision addition, double precision multiplication, double precision division, double precision sine, and double precision power respectively divided by the send time in seconds for one byte by the same processor.

The BNF reference *expr* is explained in section 3.1.9. The expressions used in these definitions usually consist of a reference to a processor timing (this section) divided by a reference to the topology (section 3.1.5). Defining these parameters in the processor section might not seem logical, since they combine parameters from this section with those from the topology section, but for ease of reasoning one might say: the maximum output speed is also processor dependent.

Knowing this, the example presented in section 2.2.1 is extended with an entry from the machine model for the Parsytec GCel. In the combinations section of the processor definition for the T805 Transputer, the following definition can be found for the integer addition over byte send ratio:

```
ia_spd = pt(1).iga / tl(1)(1).speed
```

Here the global integer addition timing (**iga**) from processor type 1 (the T805) is divided by the send time for one byte from hardware topology 1, link 1. This expression (by looking ahead at chapter 4) actually evaluates to `0.81`.

3.1.4 Machine Combinations

Very few parameters give a global impression of the machine. [Markatos92] presents the sum of the maximum transfer rates of all the processors as a useful index for the communications performance. This parameter is therefore specified in de model, though it provides little extra information for performance estimation. This parameter has the following BNF grammar:

mcombspec	::=	**maxtotalspd** = *expr nl*

In the machine combinations section of the definition for the Parsytec GCel, the following definition can be found for the sum of the maximum throughput for all the processors:

```
maxtotalspd = (tl(1)(1).connect(0) * to(1).links) / tl(1)(1).speed
```

Here the number of processors connected to link 1 (`tl(1)(1).connect(0)`) is multiplied with the number of links in the topology (`to(1).links`). The resulting value is the number of link connections in the machine. This value is multiplied by the throughput (`1/tl(1)(1).speed`) of a link. This expression (by looking ahead at chapter 4) actually evaluates to `2.145e+9`.

3.1.5 Topology Parameters

Multiple topologies may be specified in one machine. One of these should be the hardware topology, the others virtual topologies. For each topology there is a reference to a routing system. A collection of routing systems is predefined. This is documented in section 3.1.6.

topspec	::=	*id* | *name* | *setuptime* | *routing* | *proccount* | *linkcount* |
		begin procid *nl procident*+ **end procid** *nl* |
		begin linklist *nl link*+ **end linklist** *nl*
id	::=	**id** = *integer nl*
name	::=	**name** = *string nl*
setuptime	::=	**setup** = *timing nl*
routing	::=	**routing** = *string nl*
proccount	::=	**processors** = *integer nl*
linkcount	::=	**links** = *integer nl*
virttopspec	::=	*id* | *name* | *setuptime* | *routing* | *proccount* |
		begin mapping *nl mapping*+ **end mapping** *nl*

The above parameters are all expected to be defined once per topology. If any of them is defined more than once, the last definition will be used.

The *id* and *name* parameters fulfill the same purpose for the topology, as they do for the machine in section 3.1.2.

The *setuptime* parameter defines the amount of time needed to setup this topology. If a program of which the execution time is to be estimated uses this topology, the creation time must be added. The BNF reference *timing* is discussed in section 3.1.8; it is a measured value.

The *routing* indicates the routing technique used for this topology. Its value is one of a predefined set that refers to a externally implemented routing strategy. These will typically be implemented in the C programming language (see section 3.1.6). Abstract descriptions of routing techniques are either to elaborate to be useful or cannot be parameterized (with parameters available in this model). Valid routing strategy names are **XY** (grid) and permutations, **XYZ** and permutations (for 3D topologies), **1Dtorus** (ring), **2Dtorus** (doughnut), **3Dtorus**, and **broadcast** (clique) routing.

The *proccount* field defines the number of processors used in this topology. This is opposed to the *proccount* field in the global machine parameters (section 3.1.2) where it indicated the number of types of processors in the machine. For each topology a processor type must be defined for every processor, this is discussed later in this section.

The *linkcount* field defines the number of links in this topology. Each link must be defined separately. This is discussed later in this section.

Processor Types in the Topology

A processor type description has the following BNF grammar:

procident ::= **p[***integer***] =** *integer nl*

For each processor – as defined above by *proccount* – a processor type must be defined. The processor number (the first *integer*) must be between one and the value for *proccount* (inclusive). The processor type (the second *integer*) must be defined in the same machine definition, though the processor type may be defined after the topology is defined.

Individual Link Descriptions

Each link in the topology is defined separately. They are listed in one block as defined in section 3.1.5. A link description has the following BNF grammar:

link ::= **begin link** *nl linkelt*+ **end link** *nl*
linkelt ::= **id =** *integer nl* | **latency =** *timing nl* |
speed = *timing nl* | **connect =** *integer*+ *nl*

For each link an **id** is specified. This is for reference purposes only and must be a number between one and the value of *linkcount* (inclusive) as defined in section 3.1.5. The links need not necessarily be defined in *id*-order.

The **latency** is a measured time (see section 3.1.8) and represents the time that passes between the moment of the send call and the moment that the entire message has arrived (and can be used) at its destination.

The **speed** field defines the time required to send one byte over this link and is expressed in seconds.

The **connect** field defines connectivity of this link and is specified as follows: the first integer following the assignment defines the number of processors connected to this link. This should be a number greater than one. The integral numbers following the first specify the processor numbers that are connected. These numbers must be between one and the value for *proccount* (previously defined in this section).

Virtual Topology Descriptions

The virtual topology is defined by defining a processor mapping and a new routing algorithm. This routing algorithm always depends on the hardware routing; defining zero delay routing for a virtual topology implies no overhead on the hardware routing. The processor mapping has the following BNF grammar:

mapping ::= **map[***integer***] =** *integer nl*

The first integral number in the above **map** entry is the processor number in the virtual topology. The second number is the real number of the processor in the hardware topology.

3.1.6 Routing Techniques

Routing techniques are implemented in C code and are referred to in the machine model by a name that refers to a pre compiled routing subroutine. This routine will be called with the specified topology in the argument list.

Valid routing technique names are:

- **XY** and **YX**. For two dimensional grids. This implies that packets sent across the network are sent along one axis until the coordinate matches and are subsequently sent along the other axis until the destination is reached. The Parsytec GCel uses XY routing.
- **XYZ** and all possible permutations. This is similar to XY type routing, but is used for 3D grid (cube) topologies.
- **1Dtorus** (ring). Every processor has two neighbours, one with a processor number one lower and one with a processor number one higher than its own (folded over the number of processors in the topology).
- **2Dtorus** (doughnut). This is similar to a two dimensional grid, but all processors are given four neighbours by adding extra links from the processors on a border of the grid, to the processors on the opposite border of the grid.
- **3Dtorus** (multi-layer doughnut with half rings around it?). Analogous to the relationship between two dimensional grids and two dimensional tori, a three dimensional torus is a cube-like topology, where each processor on one a side of the cube is connected with a processor on the opposite side.
- **broadcast** (clique). Effectively, this implies that no routing technique is necessary. This is common under bus topology machines. The FWI csys SPARCstation cluster uses **broadcast** routing.

The routing is specified in the file as a *string* so the above mentioned names will need to be contained by double quotes (as specified).

3.1.7 SPMD Supporting Run Time Systems

To make a parallel program usable on many different types of computers, it is common practice to write such programs in a standard parallel programming environments. Currently, the PVM, Express, and MPI run time systems are commonly accepted as standards. The PARASOL I model defines the behaviour of the functions in these run time systems that are needed for SPMD application programs. This usually involves functions for send and receive, packing and unpacking the data before and after these calls.

The SPMD run time system functions that will be present in the final model represented by PARASOL I will not be exactly what is specified here. The feasibility of representing all three models still needs further research. What is presented here is implemented in the model, but the fields are not yet filled in for any of the SPMD models. Until further research is done, the BNF grammar is as follows:

pvmsection	::=	**begin pvm** *nl pvmspec*+ **end pvm** *nl*
pvmspec	::=	*pvmfunc* **=** *expr nl*
pvmfunc	::=	**sync** | **broadcast** | **multicast** | **send** | **receive** |
		packbyte | **packdouble** | **packfloat** | **packint** |
		packlong | **packstring** | **unpackbyte** | **unpackdouble** |
		unpackfloat | **unpackint** | **unpacklong** | **unpackstring**
expresssection	::=	**begin express** *nl expressspec*+ **end express** *nl*
expressspec	::=	*expressfunc* **=** *expr nl*
expressfunc	::=	**sync** | **broadcast** | **multicast** | **send**
		receive | **exchange** | **combine** | **concat**
mpisection	::=	**begin mpi** *nl mpispec*+ **end mpi** *nl*
mpispec	::=	*mpifunc* **=** *expr nl*
mpifunc	::=	**send** | **receive**

All the SPMD run time system parameters are specified with expressions (see section 3.1.9). These expressions will contain many references to other machine parameters since the SPMD functions are implemented using lower level instructions (represented in other sections of the PARASOL I model).

3.1.8 Measured Values

A timing in the PARASOL I model is specified by a real number specifying the time in seconds (except when explicitly defined differently). The measurement of the execution time of one instruction or other measurable action can vary due to clock grain size, system dependent optimization, cache hits and missed, and so forth. Therefore multiple samples of the instructions execution time are taken and averaged. The standard deviation of these measurements is also calculated.

So a timing in the PARASOL I model consists of an average time and to provide an impression of the accuracy of the measurement, the standard deviation may also be specified.

timing	::=	*float float*?

The first value of the timing is the average value, the second value is the standard deviation of the measured samples.

3.1.9 Expressions with References

Expressions as machine parameters are used to define how the execution time of the function being described is constructed from other machine parameters. An expression in the PARASOL I model has the following BNF grammar:

expr	::=	*expr binaryop expr* | *unaryop expr* | **(** *expr* **)** | *function* **(** *expr* **)** | *reference* | *float* | *integer*
binaryop	::=	**-** | **+** | **∗** | **/** | **%** | **∗∗** | **==** | **>=** | **>** | **<=** | **<** | **!=** | **>>** | **<<** | **&** | **|** | **ˆ** | **&&** | **||**
unaryop	::=	**-** | **!** | **˜**
function	::=	**abs** | **acos** | **asin** | **atan** | **cos** | **cosh** | **double** | **exp** | **floor** | **int** | **log** | **log10** | **round** | **sgn** | **sin** | **sinh** | **sqrt** | **tan** | **tanh**
float	::=	**-**?*digit*+**.***digit*∗(**e**(**-**|**+**)?*digit*+)?
integer	::=	*digit*+ | **0x***hexdigit*+
digit	::=	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
hexdigit	::=	*digit* | **A** | **B** | **C** | **D** | **E** | **F**

References (*reference*) are explained in section 3.1.10. These expressions can express a wide range of complexities. The precedence of the operators is the same as is defined in the C programming language.

An example expression with references could look like:
(4∗pt(1).ila + 2∗pt(1).ils + pt(1).goto) / pc(1).smallloopspeedup

This example could represent the execution time used for one pass through a non-terminating loop, with the code

```
10    a = b + c + d + e
      b = a + a
      goto 10
```

Thus: four integer additions and two assignments, both with local parameters. Added to this is one goto instruction and this loop code is considered to be in cache so the resulting time is divided by the small loop speedup factor.

3.1.10 References to Machine Parameters

References are basically pointers to other fields in the machine description.

reference	::=	**ma.**(**id**|**processors**|**topologies**) | **pr(***integer***).id** | **pt(***integer***).***timeparams* | **pc(***integer***).**(**size**|**speed**|**smallloopspeedup**) | **po(***integer***).***combparams* | **mc.maxtotalspd** | **to(***integer***).**(**id**|**setup**|**processors**|**links**) | **tp(***integer***)(***integer***)** | **tl(***integer***)(***integer***).**(**id**|**latency**|**speed**|**connect(***integer***)**) | **vt(***integer***).**(**id**|**setup**|**processors**) | **vm(***integer***)(***integer***)** | **pv.***pvmfunc* | **ex.***expressfunc* | **mp.***mpifunc*

A few BNF terms are not mentioned here, they can be found in appendix A. The **ma** prefix stands for a global machine parameter (section 3.1.2). The fields from that section can be referred to using this prefix. The **pr**, **pt**, **pc**, and **po** prefixes are references to a processor definition; global, timing, cache and combined

(3.1.3) respectively. To refer to the correct processor, these four prefixes must be followed by a processor id number.

The **mc** prefix refers to the combined machine parameters. The only field that can be defined in the *mcombsection* section is the **maxtotalspd** field.

The **to**, **tp**, and **tl** prefixes are references to the topology definition; global, processor definitions, and link descriptions (3.1.5) respectively. The integral number following the prefix defines the topology id number referred to. The **tp** prefix is followed by a second integral number; this is the number of the processor in the topology. It refers to the processors type. For the **tl** prefix the second integral number refers to the link id number being referenced.

Referring to the connectivity of a link needs some extra attention. A connectivity reference with parameter 0 refers to the number of processors connected to the link. Other references to the connectivity, with a number between one and **tl(**. . . **)(**. . . **).connect(0)**, return what processors in the topology are connected to the link.

The **vt** and **vm** prefixes are references to the virtual topology definitions; global and processor mapping. The integral number following the prefix defines the virtual topology id number referred to. The **tp** prefix is followed by a second integral number; this is the number of the processor in the virtual topology. The value that is returned is the number of the processor in the hardware (real) topology (see section 3.1.5).

The **pv**, **ex**, and **mp** prefixes refer to the PVM, Express, and MPI programming models respectively; of which only the SPMD supporting functions are modelled. References to expressions return the value represented by the expression.

3.1.11 Memory Representation of the Model

As mentioned in the introduction to section 3.1, there is an analogous data structure in which the same data can be stored in memory. Precise explanation of this data structure would look exactly like the above section, except it would be annotated with C typedefs and defines. It is left out for the readers convenience.

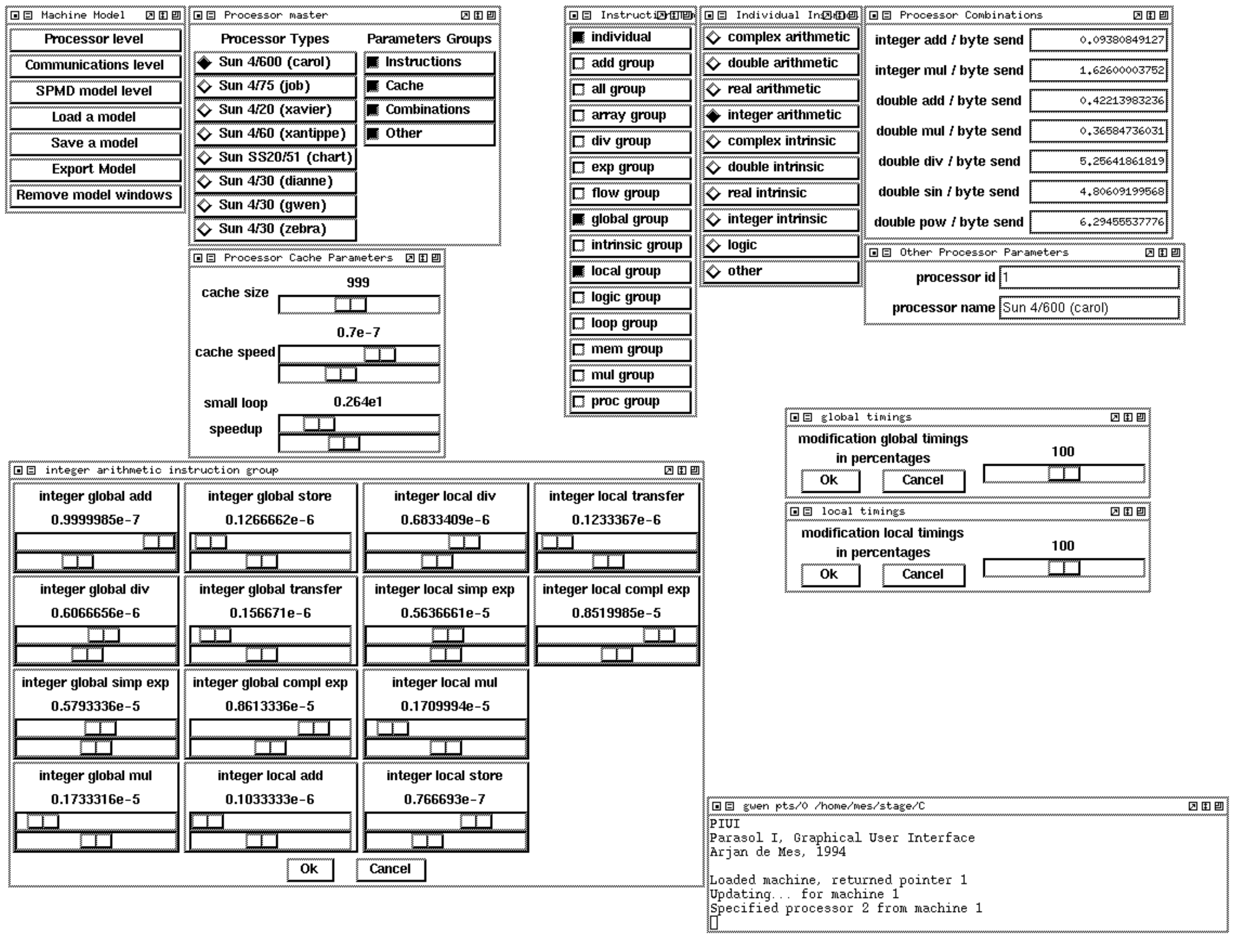
3.2 Graphical User Interface

To present users of the PARASOL system with a comprehensive overview of the machine and an intuitive interface for adjusting the parameters, PIUI (PARASOL I USER INTERFACE) was designed. In the optimal case, all aspects of the machine model should be presented in an X-windows environment. In practice some parameters are to complex to visualize. Expressions for instance are too difficult to represent in any other than a text string. PIUI does not enable editing of expressions, but instead evaluates them and presents the resulting value.

To avoid having to program the X-windows system on a low level, the TkTcl environment by [Ousterhout93] was chosen to provide the graphical front-end functions. The interpreted TkTcl environment is of course much slower than X environments that use compiled code but has many advantages. Not having to recompile and link with huge libraries, greater ease of coding and Motif look and feel are several of the advantages of this tool. The interpreter can also be interrupted to pose debug queries or simply insert commands and windows. TkTcl is becoming a de facto standard for this sort of functionality and is in the public domain. All together, a wise choice for a X-windows front-end that needs to communicate with C code.

The C code for PARASOL I is called by PIUI to perform functions such as loading, saving, and parsing of specified machine models. Also many functions the impose too much delay to the interpreter are implemented directly in C. The PARASOL II layer (effectively the SAD front end) running on a different machine can also communicate with PIUI and retrieve the latest modifications to the model. This is implemented using RPCs (Remote Procedure Calls [Birell84]).

Figure 3.1 shows a typical machine modification session using PIUI. The interface will be discussed in detail in the next sections.



integer add / byte send	0.09380849127
integer mul / byte send	1.62600003752
double add / byte send	0.42213983236
double mul / byte send	0.36584736031
double div / byte send	5.25641861819
double sin / byte send	4.80609199568
double pow / byte send	6.29455537776

processor id	1
processor name	Sun 4/600 (carol)

cache size	999
cache speed	0.7e-7
small loop speedup	0.264e1

integer global add 0.9999985e-7	integer global store 0.1266662e-6	integer local div 0.6833409e-6	integer local transfer 0.1233367e-6
integer global div 0.6066656e-6	integer global transfer 0.156671e-6	integer local simp exp 0.5636661e-5	integer local compl exp 0.8519985e-5
integer global simp exp 0.5793336e-5	integer global compl exp 0.8613336e-5	integer local mul 0.1709994e-5	
integer global mul 0.1733316e-5	integer local add 0.1033333e-6	integer local store 0.766693e-7	

modification global timings in percentages 100

modification local timings in percentages 100

```
gwen pts/0 /home/mes/stage/C
PIUI
Parasol I, Graphical User Interface
Arjan de Mes, 1994

Loaded machine, returned pointer 1
Updating... for machine 1
Specified processor 2 from machine 1
```

Figure 3.1: A snapshot of the PIUI windows while modifying the FWI csys SPARCstation cluster model

3.2.1 The Window Layout

In the upper left hand side of Figure 3.1 is a window called “Machine Model”; this is the main window from which all user operations are controlled. Directly after startup, the user can choose the “Load a model” option by clicking on the appropriate button, or wait for the PARASOL II layer to load a machine. When the load is complete, the user may choose to view or edit any of the three layers of the model. In the example given, the processor level was chosen by clicking on the appropriate button. This caused creation of the window named “Processor master”.

The Processor Windows

The “Processor master” window initially presents a list of all the processors present in the machine model. After selecting one of these processors, the window is extended to offer a choice of the processors sections. In the example, all processor sections are opened (which can be recognized by the black squares to left of the words ‘Instructions’, ‘Cache’, ‘Combinations’, and ‘Other’.

The “Instructions” window is designed to change the behaviour of groups of instructions. For instance, the time needed for all the addition instructions (addition of double precision numbers in global variables, addition of integer number in local variables, etc.) can be raised or lowered by scrolling one scale. All the processor timing parameters are in more than one group. In the example, the ‘global group’ and the ‘local group’ are opened (their windows were placed to the right of the center of the figure). All the arithmetic operator parameters are measured using global and local variables; their behaviour can be adjusted using these group modifiers. In most cases, this level of abstraction will be the one desired by users of the application.

If however a lower level is to be modified, this can be done by selecting the ‘individual’ button. At this level, all 150 processor parameters can be modified. To avoid having a window with all those parameters on the screen, they are – again – subdivided into categories. In the example, the ‘individual’ button was pressed, and ‘integer arithmetic’ was chosen. At this point the large window at the bottom left of the figure was opened. It presents all the integer expression operator parameters, for both globally defined and locally defined parameters. There are similar windows for all the buttons in the “Individual Instructions” window. Here the exact timing for each parameter can be adjusted by scrolling the two scales; the top scale scales the mantissa and the bottom scale scales the exponent.

The processor cache parameters can be modified in the “Processor Cache Parameters” window. The ‘cache size’ is a scale with logarithmic response. Just like the double scales for real numbers (as mentioned above) these are not natural to the TkTcl environment. The logarithmic scale allows a wide numeric range, while being specific for low numbers and being less specific for high numbers. The ‘cache size’ is the number of Fortran instructions that fit in the cache. This is described in section 2.2.1.

The ‘cache speed’ can be modified by adjusting the two scales representing the real number. As explained earlier, it is more effective to modify the ‘small loop speedup’. The latter is also real number. Changing in to values lower than 1.0 implies a cache that is slower than normal memory. This is unlikely, but may be experimented with.

The “Processor Combinations” window contains several ratios that give an index to the performance of the processor. They can not be changed because they are represented by expressions in the PARASOL I model. This is also the reason why they are only updated when the ‘Export Model’ or ‘Save a model’ options are selected. Their value is calculated by the PARASOL I expression evaluator.

The numbers can be compared to those of other processors. They have no other function. Comparing the different ratios on one processor, quickly indicates the differences in processing power between the normal ALU and that of a possible floating point unit. For instance, observing in one glance that integer multiplication is often much slower than double precision multiplication. When comparing the ratios to those of other processors, they indicate where the imbalances are between processing power and communicating power.

The Topology Windows

This section has not been implemented yet. Designing and implementing graphical user interfaces is a very time consuming practice. However, what should be presented in this part of the user interface will be described in detail:

The topology windows can be classified in four sections:

- Options to modify the link parameters in the hardware topology. Note that the hardware topology is specified as link parameters. Note also that it is useless to try and modify more than four links separately. If there are many links in the topology, options to modify each of them individually should not be presented. The notion of 'a group of links' should be adopted. One such group would be 'all links'.
- Options to modify the hardware topology. Standard topology types could be propagated over the existing link specification, thereby adding new, deleting, replicating or modifying existing links. For example, a grid could be changed to a ring topology by deleting several links. The other way around several links would need to be created. Here the user should be prompted for link parameters.
- Options to modify mappings of virtual topologies on the hardware topology. Mapping schemes should be offered in list boxes and if attainable, the available processors could be presented in a grid, where the user could change the mapping per processor.
- For both hardware and virtual topology multiple routing strategies should be offered. Though they are represented as strings in the PARASOL I model, valid values will need to be offered in list boxes.

Possibly a fifth section (not actually a section, but just a window) should be added to display the current evaluated value of the 'maximum total transfer speed' expression. It is an index to the total amount of data the machine can transfer per second.

The SPMD Programming Models Windows

This section has not been implemented either. Examining the required SPMD models in a way that they could be accurately represented did not fit in the time slot given for this assignment. How it should be represented in the model is discussed in section 2.2.

At this point there are few usable suggestions, on how to make these SPMD model parameters visible in a user interface. They will be represented using expressions that combine other machine parameters with arguments to the models function call. Visual representation could possibly exist of the evaluated value of the expression for several types of parameters to the SPMD function. Adjusting parameters here would then imply adding a multiplication factor in some part of said expression. This is not an optimal solution.

What Remains

The 'Export Model' button evokes the collection of all the modified values. All the opened windows report the modified parameters back to the PARASOL I layer, which collects the values and then notifies the PARASOL II application that the model has changed. A similar function is called before a file is written to disk. Then too, the most recent changes should be collected from the open windows. Some of the windows offer an 'Ok' / 'Cancel' function. When collecting the new values for exports or saving operations 'Ok' is assumed. After that point, 'Cancel' is meaningless. The 'Cancel' function is provided for directly undoing something that happened by accident.

"Remove model windows" terminates the PIUI application.

Last but not least, is the 'Terminal' window (bottom right); errors and log messages are printed in this window. This is also where the interactive debugger interacts with the user.

3.3 Interaction With Other Applications

The machine model is setup in such a way, that the information in the model can be exchanged with other applications in several ways. Information can be exchanged at the programming language level and at the file level.

3.3.1 Information Exchange using existing Functions Calls

The PARASOL I model has functions to read its own type of machine description files. The collection of those files is informally referred to as the *machine database*. The files are scanned with a scanner generated by `bison` (a parser generator frequently used as replacement for `yacc`). It accepts the syntax described in section 3.1.

The scanned model is placed in a dynamic data structure, which is validated directly after the scan. Possible errors can be listed to a return string or to a file.

The lexical analyser, scanner, validator, and data structure management routines are linked into a single library. This library provides all the functions needed for other C programs to work with the model. Examination of the data structure indicates that it is easily written to a different file type, which might be required when PARASOL I is combined with work by others.

3.3.2 Information Exchange at Runtime

Information exchange at runtime, is typically required when using an interactive interface with the model. Two types of exchange are needed. One with the user interface - be it graphical or otherwise, and one with the client that needs to use the information (PARASOL II for example).

The information exchange with the existing user interface PIUI is done by making a few functions that conform to the format required by TkTcl. These functions do nothing else than map TkTcl functions to PARASOL I functions. When the TkTcl interpreter is loaded, these functions are introduced as new command to the interpreting environment. Effectively, the interpreter can access all the PARASOL I functions that it needs.

Amongst other functions, PIUI maintains a state of the machine model, as it should be offered to PARASOL II. This leads to the other type of runtime information exchange. At this point in time, there is but one client for the PARASOL I external interface: PARASOL II. The information exchange is implemented using RPCs. Just like the additions for the user interface, there are functions that map pointers and parameter requests through RPCs to abstract pointers and answer the requests.

Chapter 4

Measurement of Real Machines

4.1 Assumptions and Limitations

One of the major objectives of the PARASOL project is to be able represent many types of parallel computers. Virtual 'what-if' machines as well as real existing parallel computers. In other words, the parameters in the model need to be able to be filled in with phony as well as real values.

The phony values can help the user of the PARASOL application to assess the possibilities of a parallel computer that has still to be made, or to speculate on what the future may bring. Real values are especially useful for estimating the computing time needed on an existing machine. This will help assess the feasibility of a port to a different parallel computer, or simply to gain insight as to how much computer time should be purchased to complete the actual computation.

In this chapter, the issue will be on how to fill in the model with correct values for existing computers. Specifically the FWI csys SPARCstation cluster and the Parsytec GCel will be measured.

4.2 Processor Level

In measuring real processors, one immediately runs into several problems. As mentioned in section 2.2.1, the time parameters for the instructions in the model should be such that the instruction was not in the cache, i.e. it needed to be fetched from memory.

4.2.1 Determining Cache Size and Speedup

Before running any other benchmarks on a processor, information needs to be gathered about the processor cache. The size of all the other processor benchmarks depends upon the outcome of this one.

To determine the cache size and speedup factor, a benchmarking program was written to measure the time needed to calculate the double precision multiplication time for locally defined variables (this choice was presented in section 2.2.1).

The double precision multiplication instruction is repeated in a loop (say x times). The loop repeats those x instructions y times. The time needed to execute these $x \cdot y$ multiplications is measured and the time needed to calculate one instruction is determined. In the pseudo code example in section 4.2.2 the lines marked with $\star$ are the lines that are repeated.

Method used to determine the cache size is as follows: the number of instructions in the loop (x) determines the size of the source program and also the final executable. If the number x is small (say 10), the instructions are likely to fit in cache and if x is large, the instructions are likely to overflow the cache. What needs be determined is the maximum value of x that does not overflow the cache. The benchmark program for this problem contains a main section that calls eight different Fortran subroutines. Each subroutine does the same calculation, but every time, with a greater value of x. The measurements are done with the following sizes of x: 10, 50, 100, 500, 1000, 5000, 10000, and 50000 (thus the lines marked with $\star$ are duplicated 10, 50, etc. times).

The measured times for the double precision multiplication of locally defined variables are plotted for an increasing number of x in Figure 4.1. This measurement works best on a computer when the processor load is low. The results presented are for a SPARCstation LX from the FWI csys SPARCstation cluster named *dianne*. Not all the plots for all the machines in the FWI csys SPARCstation cluster were this clear. For the servers, one needs to wait until the load is low (typically at night) to get a similar looking plot.

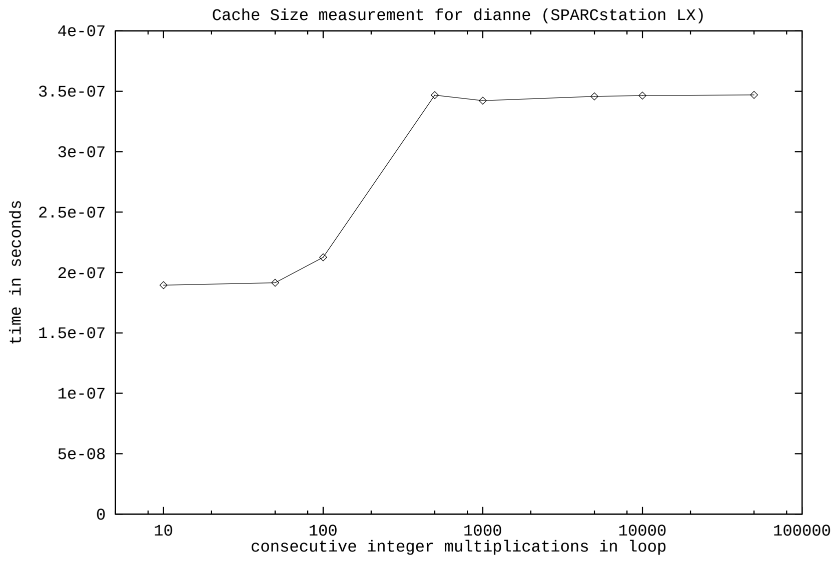


Figure 4.1: One outcome of the cache size benchmarking routine

Along the x-axis of the plot, the value for x is increased. The time needed to execute one multiplication is plotted against the y-axis. Beyond a certain value of x the time needed to execute one multiplication suddenly increases and stays at that height (approximately) for all higher values. of x. This point is exactly what could be expected for single level cache machines.

The cache size (in Figure 4.1) is the somewhere between the last plotted value for x where the timed valued is still 'low' and the first plotted value where x is 'high'. The processor is not given the benefit of the doubt, so the last x where the measured value is low is entered in the machine model as the cache size. For *dianne* this value is 100.

'Doubt' was mentioned in the last paragraph. This doubt could be eliminated by increasing the value of x is smaller steps. When the determined cache size would always be in the range between 100 and 500, this would be feasible, but for other machines in the FWI csys SPARCstation cluster cache sizes reached the 1000. The big problem here is the compilation and execution time for these benchmarks. The benchmark mentioned, that went up to 50000 repetitions took almost 11 hours to compile and 1 hour to run. The executable files generated exceeded 4.5 megabytes for the FWI csys SPARCstation cluster. This activity was too time consuming to explore to its full extent and the model is too far from being exact for these small differences to make a difference in the precision of the outcome.

From the two different values measured for the double precision multiplication, the speedup can be extracted. The time for this instruction when it is in cache is 2.13e-7, while the time needed when it is not in cache is 3.46e-7. De speedup is the ratio between these values. In this case, it is 1.62.

The assumption that this speedup factor can be used for all instructions is an assumption made in designing the model. It was validated for one other instruction (addition of locally defined integers) on one specific machine in the FWI csys SPARCstation cluster.

For the sake of curiosity, this cache size benchmark was also run on the fastest SPARC available at the FWI, a SPARCstation 10-51 named *mail*. The resulting plot is presented in Figure 4.2. This machine has second level cache, which can not be represented by the model. Despite that, understanding or explaining the resulting plot is still difficult. Is the second level cache faster than the first? How large are the caches? If the plot should have continued for more consecutive instructions in the loop, what would happen? This couldn't be tested due to the incredible compile time.

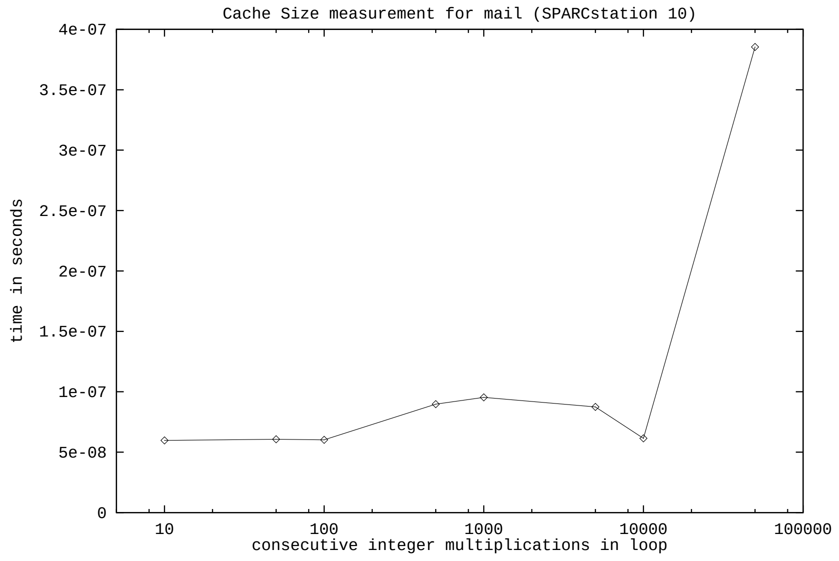


Figure 4.2: Cache size benchmarking routine attempting second level cache

It is interesting to note that for the Parsytec GCel the speedup factor was determined to be 1.0. The plot consisted of one straight line. For all values of x the same execution time was measured. This indicates that the Fortran compiler for this machine does not use the 4096 bytes of fast on-chip memory (present on the T805 Transputer) for user data or programs. This is documented in the manual, but should be mentioned nonetheless.

The value of the cache size should not be set to 0 in the model. The logarithmic scales used in PIUI can not handle the number zero and will convert it to 1 when found. This should not be a problem; as long as the speedup is 1.0, the cache size is irrelevant.

4.2.2 Basic Setup of a Processor Benchmark

To explain how the benchmarks for processor parameters were set up, an example is presented in pseudo code:

```
// first run

start = gettime()
loop index = 1 to samplesize
     value_A = data[index][1]
     value_B = data[index][2]
     value_C = data[index][3]
     result = value_A + value_B                              ⋆
endloop
end = gettime()
firsttime = end - start

// second run
// determine increment due to extra addition

start = gettime()
loop index = 1 to samplesize
     value_A = data[index][1]
     value_B = data[index][2]
     value_C = data[index][3]
     result = value_A + value_B + value_C                    ⋆
endloop
end = gettime()
secondtime = end - start

// addtime is average time needed for an add instruction
// where one argument is retrieved from memory

addtime = (secondtime - firsttime) / samplesize
```

An example of benchmarking code (in pseudo code)

Several problems in measuring the execution time for one instruction can be distinguished:

Time Occupied by ‘Other’ Instructions

To measure the time needed for one specific instruction, a lot of overhead is introduced. Code such as the time measurement code, loop overhead, and retrieving values, all become part of the measurement. In the example, this problem is solved by presenting two loops. The first loop does everything the second loop does, except execute the second addition. Compare

`result = value_A + value_B` with

```
result = value_A + value_B + value_C
```

The time needed by the extra addition instruction can be derived from comparison between the `firsttime` and the `secondtime`.

The Clock Mechanism

A problem common to most computers is the granularity of the ticks. On the machines in the FWI csys SPARCstation cluster the timer resolution is 60 ticks per second. This is a major disadvantage, when the instruction being measured completes in 100 nanoseconds.

The experiment size used to determine the time occupied, must be increased to ensure that at least 100 clock ticks pass between calls to the timer function. In the example, this can be done by increasing the value for the constant `samplesize` or repeating the lines marked with $\star$. The latter needed to be done anyway, as explained in section 4.2.1 but has the disadvantage of dramatically increasing the compile time for the application. It is repeated often enough to ‘just’ overflow instruction cache. Only after having determined how often to repeat the $\star$ lines, the constant determining the number of loop repetitions is increased to surpass the 100 clock ticks.

Once these numbers are determined for the fastest instructions in the machine, they are also used for the benchmarks for the other instructions. That way standard deviations of the measured values can be compared. It also saves an enormous amount of work for the programmer making the benchmarks.

The T805 Transputers used in the Parsytec GCel, have a much higher clock resolution. The high resolution timer produces 1 tick per microsecond. The amount of instructions that need to be executed to cover the 100 μs gap is of course much lower than the instructions needed to fill the $1\frac{2}{3}$ second gap on the SPARCs. As a result, the benchmarks for the Parsytec GCel run faster. Due to obscure compiler behaviour however, the compilation takes as much time as for the huge benchmarks on the FWI csys SPARCstation cluster.

Different Argument Values

Instructions may take more or less time, depending on the value of their arguments (section 2.2.6). With an instruction such as multiplication it is apparent that the evaluation could be done faster when one of the arguments is 1 (one) or 0 (zero). Based on the lack of context attained in the Parasol II project, the assumption is made that it is not necessary to consider this type of exception when measuring the execution time for these instructions. Therefore, when benchmarking the instructions, only more complex arguments are passed to the instructions. A stable and usable average instruction time is measured using a wide range of values to the instruction. The average should then be taken over the collective time used by these instructions. This average is not guaranteed to be the same as the average of any application program, but it will be more realistic than using the same value over and over.

The conclusion that random arguments need to be passed to the instructions, is not in itself a solution. The `random()` function occupies a non-constant amount of time. Therefore the function to generate random numbers can not be called from within the timed loop. Instead, a large array is filled with random numbers, before the timed loop starts. Accessing the array with a constant index can be done in a constant time. In the example, all three values are retrieved in both loops, while in the first, it isn’t actually used.

Distortion by Parallel Activity

Parallel activity on SPARC processors passes unnoticed by the other applications. The wall clock time incorporates the swapping in, halting, and swapping back in of an application. The desired timings of instructions should not contain these effects. The Fortran compiler on the UNIX programming environment that is available for the machines in the FWI csys SPARCstation cluster provides a notion of user time. This is the elapsed runtime (`etime()`) for the application (excluding system calls). Except for the low resolution, this is perfect for the measurement timings.

On the Parsytec GCel such a function is not available. The operating system environment called PARIX does not implement the `etime()` system call. This is not really a problem, because the user task that does the benchmarking, can be the only process that is not in a suspended state. The requirement is that the processor is not used as relay (hop) for messages between other processors. The wall clock time can be used. The high resolution timer available on the Transputer can be read via PARIX; its 1 μs accurate wall clock behaviour is the solution.

Another View on how to Benchmark Fortran Instructions

Benchmarks were made for similar purposes by [Dunlop93]. The instruction set that was modelled in that project was quite different from those in the PARASOL I model. The benchmarking subroutines used by [Dunlop93] have parts in common with those used in PARASOL I, but also differ on several points. The differences will be discussed in more detail.

- The most important and significant difference is the modelling of the instruction cache. The benchmarking routines presented by [Dunlop93] do not attempt to model speed or size of the instruction cache. This has the advantage of not having to generate the benchmarks (small Fortran programs will do), but it might be a loss when attempting to model real behaviour of an application.

- The values that are passed in the arguments to the operations by [Dunlop93] are constants. In a benchmark, the time to calculate the same value over and over again is measured. Optimizations in the ALU or floating point unit are not recognized due to this. The PARASOL I benchmarks use different (though not the trivial) values as arguments to the operations. For [Dunlop93], this has the advantage of not having to juggle with arrays of random values.

- The loops in the PARASOL I benchmarks are executed multiple times. For multi-user machines this is needed to determine the amount of distortion in the measurements, caused by task switching. The measured user time in UNIX programming environments is not as reliable as is needed for such benchmarks. PARASOL I also calculates a standard deviation of its measurements and includes that in the model. [Dunlop93] does measure multiple instructions, but measures one time. Without multiple timings, [Dunlop93] obtains no information on the accuracy of the measurement. A reason for not wanting that information, could be that there is nothing to do about it.

Though there are important differences, there are also similarities between the benchmarks for PARASOL I and those by [Dunlop93].

- Both benchmarks measure the time of Fortran instructions. They do so, by measuring the time needed to execute multiple instructions, and dividing the measured time by the sample size.

- Neither of the benchmarks attempt to model data cache, or measure its hit and miss rates.

In [Dunlop93], discouraging results are presented about the their model. Estimations done with the results of the benchmarks are mostly to low. They recognize three causes for the inaccuracy in their benchmark prototype:

- Varying execution time for an instruction.
- The benchmarks are not suited for statements that generate little assembly code. [Dunlop93] fails to explain why.
- Problems in isolating the one instruction being measured.

4.2.3 Arithmetic Expression Operators

As specified in section 2.2.1, there are 56 entries in the category 'arithmetic operators'. There are four numeric types, two possible locations of the variables used, and seven distinct instruction types. For each of these entries, a benchmark needed to be created. These benchmarks all look alike, except for the following modifications:

- The array named `data[][]` in the example needs to have the appropriate numeric type, as do the variables to which the values are assigned.
- When the variables need to be global, code is added to place `value_A`, `value_B`, and `value_C`, in a Fortran `COMMON` block.
- The arithmetic operator in the line marked with ⋆ in the example is changed to the correct operator.
- The random values placed in the `data[][]` array are adjusted to match the domain of the binary arithmetic operator.

Considering these adjustments, all the benchmark routines for arithmetic expression operators can be generated from one main skeleton. A generator was written in C-code to generate all 56 Fortran benchmarking subroutines.

The generator is a very straight forward piece of code. There is one problem however. The expression operators listed in section 2.2.1 contain two instructions involving memory transfer. The listed entry 'assignment' specifies the cost for one assignment (`a = ...`), while the entry 'memory transfer' specifies the cost for a copy of a block of data in memory to another location in memory (`a = b`). This distinction is also made in [Saavedra89]. While the assignment assumes the result of an expression to already be in some register or location that is easily accessed by the processor, the memory transfer implies the loading of the value and then storing its result to the new location in memory. It should be noted that the loading of values from memory is incorporated in timings for the arithmetic expression operators listed in the model.

While the cost of a memory transfer can easily be measured with the presented benchmarking routine, the cost of an assignment is considerably more difficult. The solution chosen to measure the assignment, is to modify the assembly code generated by the Fortran compiler for the memory transfer. It involves changing the assembly code from the first experiment from something that looks like this:

```
    ld [offset1],ra
    st ra,[offset2]
```

to something that looks like this:

```
    ld [offset1],ra
```

In effect, the time needed for an assignment is measured by comparing the time needed for a load instruction in assembly level to the time needed for a load and a store instruction in assembly level.

Then is it safe to modify assembly code, when claiming to be modelling Fortran code? Yes, because the assembly code being modified, is code generated by the Fortran compiler (F77). The only modification applied is deleting several lines that form half of the memory transfer action.

4.2.4 Intrinsic Functions

The intrinsic functions that are modelled in the machine model are listed in section 2.2.1. The following aspects are considered when measuring the time needed to execute an intrinsic function:

- A broad and realistic range of values over the input domain of each intrinsic function should be provided. The execution time for any intrinsic function is bound to vary depending on the difficulty of the input value. An average execution time over a randomly distributed valid input domain is used.
- Many intrinsic functions can have multiple numeric types as input. Their output types depend on their input types. Each intrinsic function that is represented in the model, has an entry for every valid numeric type and each of them should be benchmarked separately.
- Every modern Fortran compiler implements all its intrinsic functions using library calls. Due to this, there is no significant difference in execution time, whether the value passed to the function is in a locally defined variable, a globally defined variable, or is the result of an expression. For the PARASOL I benchmarks, locally defined variables are used.
- Because the basic layout of the benchmark presented in section 4.2.2 must be maintained, the execution time for the intrinsic function must be compared to the most trivial function available: the identity function. So in the pseudo code example, the first line marked with $\star$ is:
  ```
  result = id(value_A)
  ```
  while the second line marked with $\star$ is:
  ```
  result = function(value_A)
  ```
  where `function` is the intrinsic function being measured.

Again, all the benchmarks for intrinsic functions are similar. Therefore one generator, written in C-code, is used to create all the Fortran benchmarking subroutines.

4.2.5 Remaining Processor Timings

The other processor parameters that need to be measured are the timings for:

- unconditional and computed ‘go to’s
- array indexing, from 1 through 7 suffixes
- boolean operators
- operators for comparing numbers

Branching Statements

The time occupied by the `GOTO` branching statement is modelled using two parameters, as it has two different syntaxes in Fortran. The normal ‘go to’ statement – unconditional branch – is a ‘go to’ instruction followed by a label determining where to branch, while the other – computed branch – ‘go to’ is followed by a list of possible labels and an expression to determine which of those labels is target for the branch.

To exceed the instruction cache for the unconditional branch, the number of ‘go to’ instructions the loop needs to be large enough, and because the instructions involved are branching instructions, they should branch equally throughout the ‘go to’s available in the loop. The C program that generates the loop, fills in the branch labels in a manner that achieves maximum jumping distance while staying within the loop.

The computed ‘go to’ benchmark is generated in a slightly different manner. The computed ‘go to’ needs to compute something, to determine where to branch to. For that purpose, a locally defined array is filled with random numbers. These random numbers are low: between 1 and the number of labels following the ‘go to’ statement. In the measurement loop, enough computed ‘go to’ instructions are placed to overflow the instruction cache. Each of these instructions is followed by a series of labels and is ended by an array reference to a random number. In this way, the jumping distance is also random.

Array Indexing

In Fortran, arrays may be indexed with at most 7 suffixes. Each suffix represents a dimension in the arrays data space. From measurements on the SPARC processors, it was determined that the time needed to index an extra dimension, was not constant. In other words, every extra index took a varying amount of extra time.

As a result, the benchmarks measure each of the allowed number of indexes separately. To avoid distortion in the time needed to determine the offset caused by an index, the C program generating the benchmark fills in random constants, in the range that is valid for the dimension.

As an example, the measurement of array indexing with four suffixes is discussed in more detail. Starting with the basic setup of a benchmark (section 4.2.2), the first line marked with a $\star$ becomes:

```
result = variable
```

while the second line marked with $\star$ becomes:

```
result = array[a][b][c][d]
```

These lines are repeated as often as necessary to overflow instruction cache. The variables a, b, c, and d are filled in by the benchmark generator. The difference in time needed to execute both lines is the time needed for accessing an array with four suffixes. The numeric types of `variable` and `array` appeared to be of no influence to the measurement. This is what could have been expected.

Logic Operators

Two different types of instructions work with boolean types. Comparing two variables results in a boolean to represent the result of the test, and boolean values can be used further with operators specific to the numeric type (such as *and* and *or*).

The following assumptions were made and validated:

- The comparison operators $>$, $\geq$, $<$, $\leq$, $=$, and $\neq$ all use the same amount of time.
- The amount of time for a comparison operator does depend on the numeric type and whether the variables they are comparing are defined locally or globally.
- A logical *and* executes in about the same amount of time as the logical *or*.
- The timings for *and* and *or* do depend on whether the variables are defined locally or globally.

The benchmarks for the aspects of logical operations that needed to be measured were made analogous to the benchmarks for the ‘arithmetic expression operators’ (section 4.2.3).

4.2.6 Other Processor Parameters

The remaining processor parameters are not of the type that need to be measured. Parameters such as a processor name string can just be typed in (using PIUI). The combined processor parameters are expressions. These expressions look the same for all processor types, it is just the evaluated value that varies. They should can typed in using any text editor.

4.3 Topology Level

Measurements of the topology level are a part of PARASOL I, but the lack of time to research the diverse aspects of measuring link speeds and routing techniques and the availability of results of the same nature by other researchers, led to the adoption of values found by others.

4.3.1 Ethernet Systems

Ethernet is a bus system without a bus-master. In this type of system contention and packets collision occurs frequently. While Ethernet (IEEE 802.3) implements CSMA/CD with random exponential back off, PARASOL I models it very naively using a maximum throughput and a latency parameter.

The document by [Boggs88] provides a study of Ethernet based on realistic measurements. They provide measurements with different packet sizes, measurements to determine overhead due to the random back off, and also describe the effect of the amount of machines attached to one Ethernet.

The Ethernet that interconnects the individual machines in the FWI csys SPARCstation cluster hooks up more than 40 machines. Through reading and extrapolating the graphs [Boggs88] presents, it can be concluded that for the FWI csys SPARCstation cluster the maximum throughput is 9.3 megabits per second (at packet size of about 1024 bytes, which is an average normal amount). The time to send one byte, which is the parameter that needs to be filled in, is calculated as follows:

$$link\ speed = \frac{8\ bits/byte}{9.3 \cdot 10^6\ bits/sec} \approx 8.6 \cdot 10^{-7}\ sec/byte$$

The latency (transmission delay) for a message can be read from another graph in the same document. For the FWI csys SPARCstation cluster the extrapolated value is 30 milliseconds. [Boggs88] explicitly contradict the myth that transmission delays increase dramatically when the load on the network exceeds 37%. Instead their measurements show linear behaviour against an increasing load. This means that the PARASOL I model can represent Ethernet better than expected, though still far from perfect.

4.3.2 Links in a Grid

For the link parameters in the Parsytec GCel, the document by [Linden94] was used. In this document, the parameters measured are abstracted from the fact that there are 16 T805 Transputers on one board, 4 boards in one cube and 8 cubes in the Parsytec GCel (thus containing 512 processing nodes). In the real machine, extra delays are introduced when a message is sent off of a board and it is delayed even more when it is sent off of a cube. The parameters for the links were measured in [Linden94] analogous to how links are represented in the PARASOL I model, so the useability and validity of these results are high.

The maximum throughput for one link including the PARIX software layer, was found to be 1.1 megabytes per second. The resulting send time per byte in the PARASOL I model for the Parsytec GCel is 9.1e-7 seconds per byte. The corresponding latency of a link was found to be 28.5e-6 seconds for every message sent. A large part of this latency will, again, be due to the software layer that is called to send the message.

4.3.3 Routing Analysis

Routing methods are parameterized using pre-compiled C-code functions. Essentially, the routing technique used by the hardware it represents must be mimicked. In addition, the routing function call, as referred to in PARASOL I, implements the virtual links. For the Parsytec GCel, the information on what virtual links exist can be derived from [Röttger94]. For the FWI csys SPARCstation cluster, no virtual links need to be added, as the Ethernet hardware already interconnects all processing nodes (clique).

4.4 SPMD Supporting Run Time Systems Level

As mentioned in section 3.1.7, the SPMD supporting run time systems can not yet be fully represented. This is offered in chapter 6 as research that still needs to be done.

The implementation of a single SPMD supporting environment function on a specific machine, may be analysed using PARASOL II. The resulting SAD expression ([Halderen94]) may be converted to a arithmetical expression usable by PARASOL I and thus save a lot of time in correctly specifying the functions implementation costs. Using PARASOL II, significant information as to how the parameters to the function influence its execution time should also be derived.

Combining the resulting SAD formula with the available processor and topology parameters should provide a fast and portable method for modelling the functions in the SPMD supporting run time systems (PVM, Express, and MPI).

4.5 Validation

One point is still open: does it work? In other words, does the model that is presented, represent the machine in such a way that it can be used for determining trends in execution time, as is the objective (section 1.2, item 2)?

Two examples are presented to validate the correctness of the parameters in the model, and for values measured on real machines. Virtual ('what-if') machines are of no relevance until it is determined that the model works for real machines.

4.5.1 Experiment with a Small Linear Program

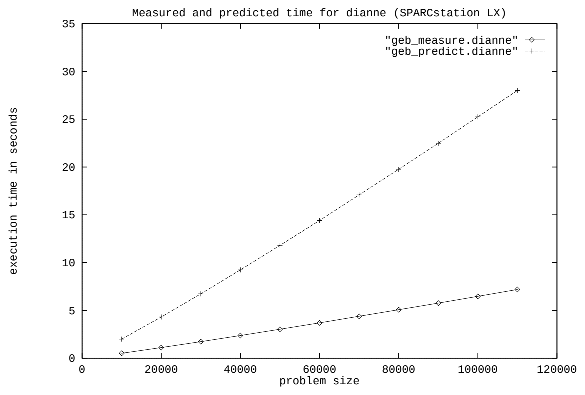
A small program, with linear behaviour in time (though it is hard to prove for unmeasured values) was taken from [Hofstadter79]. In this section the problem will be referred to as the GEB problem (from Gödel, Escher, Bach).

The problem is: will the following algorithm terminate for any integer number greater than one?

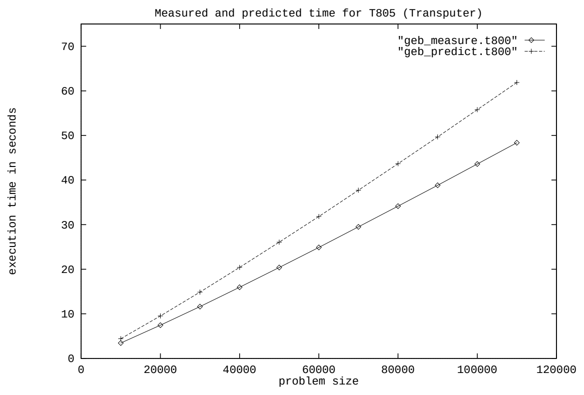
```
1  $x$ = the given input number
2  repeat
3      if $x$ is even
4          $x$ = $x$ divided by two
5      else
6          $x$ = $x$ times three plus one
7  until $x$ = one
```

The GEB problem was translated to Fortran in a program that tests this algorithm for a range of numbers; from two to a value y. The variable y thus represents the problem size. No optimizations were made in the algorithm. It simply runs the above test for all values in the range and measures the required execution time. The experiment was done for values for y from 10000 through 110000 in steps of 10000.

To predict execution time, the number of times each instruction was executed needed to be determined. This was done using the line profiling option in the Fortran compiler. The instructions that were represented in the PARASOL I model were multiplied by the number of times they were executed, and the sum all of these products formed the estimated execution time (just like the small example given in section 3.1.9). Because the amount of code is so small, it fits into the instruction cache, so the resulting sum needs to be divided by the 'small loop speedup'.



4.3 (a): SPARC Processor



4.3 (b): T805 Transputer

Figure 4.3: Measured and predicted timings for the GEB problem

In Figure 4.3, the measured execution time on the real machine and the predicted execution time are plotted. From graphs for both SPARC processor and Transputer, it is apparent that trends in execution time are correct. The diverging lines are not a problem. In the objectives presented in section 1.2, the goal is to predict trends.

Because the graphs for the SPARC processor diverge faster than those for the Transputer, the results for SPARC processor will be discussed in more detail. Discussion of the results for the Transputer would be analogous.

The predicted execution times for the SPARC processor are higher than the measured times for all problem sizes. The differences between the values are listed in the following table:

problem size	predicted time		measured time		ratio
10000	1.996	/	0.512	=	3.898
20000	4.309	/	1.105	=	3.900
30000	6.728	/	1.725	=	3.900
40000	9.236	/	2.370	=	3.897
50000	11.802	/	3.026	=	3.900
60000	14.408	/	3.694	=	3.900
70000	17.079	/	4.381	=	3.898
80000	19.772	/	5.067	=	3.902
90000	22.487	/	5.765	=	3.901
100000	25.252	/	6.472	=	3.902
110000	28.011	/	7.185	=	3.899

Table 4.2: Comparison of measured and predicted values

The ratio ≈ 3.9 for all the problem sizes, thus the machine parameters that are modelled in PARASOL I, are clearly those that influence the performance. The possibility of errors in the instances of these parameters (i.e. the benchmark results) are not out of the question, though.

The PARASOL I model combined with profiler information or an execution time estimator such as PARASOL II, can predict performance trends. Together with a measured execution time (for this program one measured value is enough – the required number of measurements might be related to the number of parameters that the problem size consists of), a ratio can be determined, after which all other execution times can be calculated with considerable precision.

For instance, assume the program was measured for problem size 10000. The measured time would be 0.512, while the predicted size would be 1.996 (see Table 4.2). The ratio 3.9 could be derived, and the execution time for all other program sizes would be predictable.

To determine where all the extra time being predicted is coming from, a close examination of the source code and the generated assembly code for the program is needed. Examination of the predicted time for a problem size of 100000 revealed that 12.2 seconds seconds were predicted for determining if a number was even (`mod(`x`,2) == 0`) and 4.5 seconds were predicted for dividing by two. Examination of the assembly code showed compiler optimizations. Even though the program was compiled with optimization turned off, the Fortran compiler implemented these operations using bit shifting instructions. Bit shifting operations are much faster than the original instructions. To see how large the effect of these optimizations are on the predicted timings, the time for a 'logical and' was substituted as time for division and mod operations. As a result, the predicted time dropped from 25.3 seconds to 11.3. The latter is only 75% off, from the predicted value.

This example shows that for many instructions, optimizations are so trivial that a compiler implements them even when 'no optimization' is selected. What could be considered as an optimization by one, could be considered an efficient implementation by another. In retrospect, the model might need to be adjusted for differing complexity levels of all instructions, thus preventing more such errors. Both compiler and processor should be studied to model all effects. Defining the complexity level of arguments would be a difficult problem. Effectively using all those parameters for performance estimation, would certainly be another.

The causes for the higher accuracy in the estimation of execution time on the Transputer are (see also section 4.2.2):

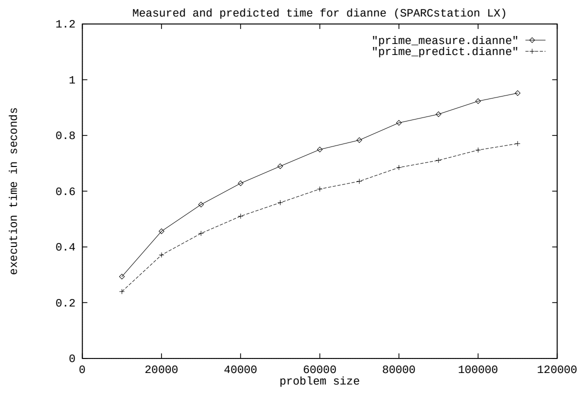
- the higher resolution of the clock provides more accurate measurements for both benchmarks and the application targeted for prediction,
- the compiler does less optimization than for the SPARC workstations; the requirement was to have no optimization, so accuracy in the prediction is gained,
- there is no interference by other processes; the measured time is not influenced by operating system or peer processes, and
- the cache model can provide no distortion, as there is no cache.

All in all, the Transputer is a much simpler and more straight forward processor.

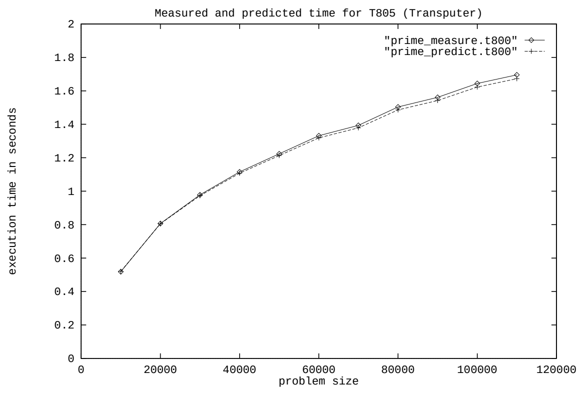
4.5.2 Experiment with a Small Polynomial Program

Another example is presented to establish that estimated times need not have a linear relation to the problem size. An algorithm is used that searches a number range for prime numbers. When it determines that a number is or is not prime, it proceeds to the next number in the range. In this example, the range is always 10000 numbers wide, and the problem size is the upper bound of the range, starting at 10000, e.g. for problem size 40000, the range of numbers from 30001 to 40000 are check for being prime numbers.

Again, the results are plotted (Figure 4.4). The predicted values are now below the measured, real values. The small ditches in measured execution times around problem sizes 70000 and 90000 are also present in



4.4 (a): SPARC Processor



4.4 (b): T805 Transputer

Figure 4.4: Measured and predicted timings for the prime number problem

the estimated execution times. As is clearly visible in the graphs, the ratios between the measured and the estimated execution times can be determined for small problem sizes, and used to estimate execution time for larger problem sizes. The ratios are ≈ 0.81 and ≈ 0.99 for Figure 4.4 (a) and Figure 4.4 (b) respectively.

4.5.3 Experiments with the Performance Estimator

For estimations of more complex programs, and ultimately of the PAM-CRASH code, analysis of the executed instructions cannot be done by hand. PARASOL II is being developed to do it automatically. At this point in its development, the memory usage involved and the Fortran instructions used in its core, prevent the PAM-CRASH code from being analysed adequately.

Other (less) large finite element codes are available however. To validate the machine model, MD1 will be subject to automatic performance prediction, using with both PARASOL tools. MD1 is presented as a realistic benchmark in [Addison93]. It solves a molecular dynamics problem which in turn involves the solving of the equations of motion for a system of a large number of interacting particles, i.e. solving second order differential equations. The problem is solved numerically by calculating approximate solutions at a large number of time steps. A time step consists of calculating the forces on the particles, calculating their new positions, and calculating other properties of interest. Its parallel version is a typical SPMD program (section 2.2).

For validation of the PARASOL I model, the sequential version of MD1 was analysed by PARASOL II and the machine parameters were filled in. For discussion on how this was done, see [Halderen94].

In Table 4.3, the results of the predictions are compared with the corresponding measured values. The average ratio ≈ 0.76 over the compared range. However the trend in the ever decreasing ratio indicates that the predicted value diverges further and further from the measured value. This is not the behaviour that would be expected. There are however several things to keep in mind concerning the current state of the PARASOL II project: the overhead for a procedure call is not incorporated in the predictions and the small loop speedup is always used (i.e. all the instructions are always in instruction cache).

A trend in execution time for the MD1 program can be recognized nonetheless. Tuning machine parameters will still give relevant information about what happens to the performance of the application. Varying execution times due to the tuning of an application (e.g. optimizing code fragments) will also give adequate information about what happens to the performance. Unfortunately, the deviation from the measured value is not constant, thus accuracy is lost.

4.5.4 Experiments with Topology Parameters

To validate that *parallel* computers are represented correctly, experiments need to be done with applications that use the communications capabilities of the machine. The PARASOL model for is not suited for evaluating these complex types of experiments by hand. Assisting in the analysis of communication behaviour of applications, is also a task that needs to be done by the PARASOL II tool. At this point in time, PARASOL II is not capable of modelling communication behaviour, like sending of messages and synchronizing of processes, as its development is only partially complete. The routing aspects that will be modelled as part of PARASOL I are not implemented yet either.

Thus validation of the topology parameters cannot be done. Possible inaccuracies in representing the PARIX layer on the Parsytec GCel might be caused by the delay in the hops, and the number of crossbar switches that need to be passed (see section 4.3.2 – abstraction from boards and cubes).

Possible inaccuracies in the modelling the behaviour of Ethernet might be caused by not being able to represent the effects of random exponential back off, when a collision occurs. When an Ethernet link becomes saturated, the throughput can drop dramatically ([Boggs88]); this is not incorporated in the PARASOL I model.

problem size	predicted time		measured time		ratio
1	2.55	/	3.04	=	0.84
2	4.68	/	5.97	=	0.78
3	14.23	/	17.62	=	0.81
4	23.74	/	30.94	=	0.77
5	45.01	/	58.11	=	0.77
6	68.31	/	87.87	=	0.78
7	108.86	/	141.01	=	0.77
8	149.22	/	204.74	=	0.73
9	198.64	/	265.05	=	0.75
10	277.29	/	363.04	=	0.76
11	350.50	/	465.32	=	0.75
12	463.37	/	617.03	=	0.75
13	565.03	/	762.07	=	0.74
14	718.28	/	952.90	=	0.75
15	853.07	/	1149.61	=	0.74
16	1052.88	/	1495.07	=	0.70
17	1225.46	/	1675.49	=	0.73
18	1416.26	/	1921.28	=	0.74
19	1693.02	/	2333.41	=	0.73
20	1928.34	/	2659.20	=	0.73

Table 4.3: Comparison of measured and predicted values for MD1

Chapter 5

Conclusions

5.1 A Model is Presented

An abstract model for representing massive parallel computers is presented. It can represent processors at Fortran level and topologies at a level that contains throughput, latency, and routing aspects. The model also provides a framework for specifying execution behaviour of SPMD supporting run time systems. The processor parameters in the machine can be visualized and modified in a graphical user interface. From within this interactive environment, the machine parameters can be exchanged with client applications that use the values. In effect, the model combined with the user interface provide an useful interactive angle on the machine. The combination with a performance estimation application such as PARASOL II, makes it an appropriate tool for designers of Fortran programs and massive parallel machines.

5.2 The Different Caches in Modern Computers

In the presented model, single level instruction cache is modelled. All other forms of caching are ignored. When attempting to determine the instruction cache size of a processor, problems arise. The processor cache was not meant to be measured in this way. Instead, the cache attempts to delay the effect sought after to the maximum possible extent. A reasonable conclusion would be to assume every instruction is in the instruction cache. Modelling the cache with an ‘enormous loop slowdown factor’ would be closer to reality. Attempting to model more than one level of cache will probably bring forward even more problems along with the conclusion that it is nearly impossible (and not realistic) that a second or third level overflows.

It would be very useful however if a way were devised to represent the data cache. It overflows frequently in most applications and will influence the performance in a much more radical way.

5.3 Usability and Results

Two methods were used for validation of the presented processor model. Validation by hand, using two small examples, and validation by means of automatic program analysis on larger, more relevant applications. The parallel aspects of the model are not validated. This will need to be done, using the PARASOL II tools, but they are not yet developed far enough for this purpose.

From the validation using small sequential examples, it is concluded that the model represents the correct parameters. These parameters *are* the parameters that influence performance. The model can be used to correctly predict trends in execution time. When the execution time of an application can be measured for small input sets, the model can be used to predict the real the time needed for the same application with larger input sets with considerable accuracy.

To get more accuracy in a first estimation however, the model should be adjusted to represent the differing levels of complexity of values passed to all the instructions.

When using automatic program analysis to assist in the estimation of the execution time needed, the deviation from the measured execution time is not a constant percentage. The results are good enough however for detecting trends in execution time due to adjustments of machine parameters or adjustments in the code.

Chapter 6

Future Work

6.1 Completion of the Parasol I Project

The SPMD supporting run time systems that are to be modelled in PARASOL I should be subject of further research. It should be determined which functions in the respective environments should be in the PARASOL I model, how they are implemented on the existing machines, and how to incorporate parameters to the functions in the expressions currently available in the machine model.

PIUI is still incomplete due to the enormous amount of work required in the making of any user interface. The topology and SPMD supporting system parameters can not be visualized at this point. An aspect that may prove very interesting, is a method for visualizing the expressions. These expressions are very important as they define the relationships between parameters in the different levels of the model. Currently, expressions are visualized by their respective evaluated values.

The various routing techniques introduced are not yet implemented. For the Parsytec GCel, the techniques change for each update in PARIX. [Röttger94] are still developing more optimal mappings (‘embeddings’) for processors numbers in virtual topologies on processors in the hardware topology.

The way routing techniques are defined in the PARASOL I model, is by referring to a text label attached to a compiled subroutine, implemented in C. For the time being, the parser accepts the values marked as available (in section 3.1.6), but cannot handle calls concerning topologies from clients.

6.2 Imperfections in the Parameter Values

The communication parameters filled in, are directly taken from references [Boggs88, Linden94, Röttger94]. It should be considered to attempt measuring these values as part of the benchmark routines for PARASOL I.

Appendix A

Complete BNF Grammar for Model Files

This is the complete BNF grammar for the model file format as described in section 3.1.

Table A.1: Complete BNF grammar for model files

machine	::=	**begin machine** *nl machfield*+ *machsection*+ **end machine** *nl*
machfield	::=	*id* | *name* | *proccount* | *topcount*
id	::=	**id =** *integer nl*
name	::=	**name =** *string nl*
proccount	::=	**processors =** *integer nl*
topcount	::=	**topologies =** *integer nl*
machsection	::=	*procsection* | *mcombsection* | *topsection* | *virttopsection* | *pvmsection* | *expresssection* | *mpisection*
procsection	::=	**begin processor** *nl procspec*+ **end processor** *nl*
procspec	::=	*id* | *name* | **begin timings** *nl timings*+ **end timings** *nl* | **begin cache** *nl cache*+ **end cache** *nl* | **begin combinations** *nl combined*+ **end combinations** *nl*
timings	::=	*timeparams* = *timing nl*
timeparams	::=	**absc** | **absd** | **absi** | **absr** | **aimagc** | **aintd** | **aintr** | **anintd** | **anintr** | **argld** | **arr1** | **arr2** | **arr3** | **arr4** | **arr5** | **arr6** | **arr7** | **asind** | **asinr** | **atand** | **atanr** | **cga** | **cgd** | **cge** | **cgm** | **cgs** | **cgt** | **cgx** | **chari** | **cla** | **cld** | **cle** | **clm** | **cls** | **clt** | **clx** | **cmplx2d** | **cmplx2i** | **cmplx2r** | **cmplxc** | **cmplxd** | **cmplxi** | **cmplxr** | **conjgc** | **dblec** | **dbled** | **dblei** | **dbler** | **dga** | **dgd** | **dge** | **dgm** | **dgs** | **dgt** | **dgx** | **dimd** | **dimi** | **dimr** | **dla** | **dld** | **dle** | **dlm** | **dls** | **dlt** | **dlx** | **dprodr** | **expc** | **expd** | **expr** | **gcom** | **goto** | **iga** | **igd** | **ige** | **igm** | **igs** | **igt** | **igx** | **ila** | **ild** | **ile** | **ilm** | **ils** | **ilt** | **ilx** | **intc** | **intd** | **inti** | **intr** | **lga** | **lgcc** | **lgdc** | **lgic** | **lgrc** | **lla** | **llcc** | **lldc** | **llic** | **llrc** | **log10d** | **log10r** | **logc** | **logd** | **logr** | **loopi1** | **loopin** | **loopoh1** | **loopohn** | **maxd** | **maxi** | **maxr** | **modd** | **modi** | **modr** | **nintd** | **nintr** | **pcall** | **realc** | **reald** | **reali** | **realr** | **rga** | **rgd** | **rge** | **rgm** | **rgs** | **rgt** | **rgx** | **rla** | **rld** | **rle** | **rlm** | **rls** | **rlt** | **rlx** | **signd** | **signi** | **signr** | **sinc** | **sind** | **sinhd** | **sinhr** | **sinr** | **sqrtc** | **sqrtd** | **sqrtr** | **tand** | **tanhd** | **tanhr** | **tanr**

continued...

Table A.1: (continued)

cache	::=	**size =** *integer nl* |
		speed = *timing nl* |
		smallloopspeedup = *timing nl*
combined	::=	*combparams* **=** *expr nl*
combparams	::=	**ia_spd** | **im_spd** |
		da_spd | **dm_spd** | **dd_spd** | **ds_spd** | **dp_spd**
mcombsection	::=	**begin machinecombinations** *nl mcombspec+*
		end machinecombinations *nl*
mcombspec	::=	**maxtotalspd =** *expr nl*
topsection	::=	**begin topology** *nl topspec+* **end topology** *nl*
topspec	::=	*id* | *name* | *setuptime* | *routing* | *proccount* | *linkcount* |
		begin procid *nl procident+* **end procid** *nl* |
		begin linklist *nl link+* **end linklist** *nl*
setuptime	::=	**setup =** *timing nl*
routing	::=	**routing =** *string nl*
linkcount	::=	**links =** *integer nl*
procident	::=	**p[***integer***] =** *integer nl*
link	::=	**begin link** *nl linkelt+* **end link** *nl*
linkelt	::=	*id* |
		latency = *timing nl* |
		speed = *timing nl* |
		connect = *integer+ nl*
virttopsection	::=	**begin virtual** *nl virttopspec+* **end virtual** *nl*
virttopspec	::=	*id* | *name* | *setuptime* | *routing* | *proccount* |
		begin mapping *nl mapping+* **end mapping** *nl*
mapping	::=	**map[***integer***] =** *integer nl*
pvmsection	::=	**begin pvm** *nl pvmspec+* **end pvm** *nl*
pvmspec	::=	*pvmfunc* **=** *expr nl*
pvmfunc	::=	**sync** | **broadcast** | **multicast** | **send** | **receive** |
		packbyte | **packdouble** | **packfloat** | **packint** |
		packlong | **packstring** | **unpackbyte** | **unpackdouble** |
		unpackfloat | **unpackint** | **unpacklong** | **unpackstring**
expresssection	::=	**begin express** *nl expressspec+* **end express** *nl*
expressspec	::=	*expressfunc* **=** *expr nl*
expressfunc	::=	**sync** | **broadcast** | **multicast** | **send**
		receive | **exchange** | **combine** | **concat**
mpisection	::=	**begin mpi** *nl mpispec+* **end mpi** *nl*
mpispec	::=	*mpifunc* **=** *expr nl*
mpifunc	::=	**send** | **receive**
expr	::=	*expr binaryop expr* | *unaryop expr* | **(** *expr* **)** |
		function **(** *expr* **)** | *reference* | *float* | *integer*
binaryop	::=	**-** | **+** | **∗** | **/** | **%** | **∗∗** | **==** | **>=** | **>** | **<=** | **<** | **!=** |
		>> | **<<** | **&** | **|** | **ˆ** | **&&** | **||**
unaryop	::=	**-** | **!** | **˜**

continued...

Table A.1: (continued)

function	::=	**abs** | **acos** | **asin** | **atan** | **cos** | **cosh** | **double** |
		exp | **floor** | **int** | **log** | **log10** | **round** | **sgn** |
		sin | **sinh** | **sqrt** | **tan** | **tanh**
timing	::=	*float float*?
reference	::=	**ma.**(**id**|**processors**|**topologies**) |
		pr(*integer***).id** |
		pt(*integer***).***timeparams* |
		pc(*integer***).**(**size**|**speed**|**smallloopspeedup**) |
		po(*integer***).***combparams* |
		mc.maxtotalspd |
		to(*integer***).**(**id**|**setup**|**processors**|**links**) |
		tp(*integer***)(***integer***)** |
		tl(*integer***)(***integer***).**(**id**|**latency**|**speed**|**connect(***integer***)**) |
		vt(*integer***).**(**id**|**setup**|**processors**) |
		vm(*integer***)(***integer***)** |
		pv.*pvmfunc* | **ex.***expressfunc* | **mp.***mpifunc*
float	::=	**-**?*digit*+**.***digit*∗(**e**(**-**|**+**)?*digit*+)?
integer	::=	*digit*+ | **0x***hexdigit*+
digit	::=	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
hexdigit	::=	*digit* | **A** | **B** | **C** | **D** | **E** | **F**
string	::=	**"***character*+**"**
character	::=	any character other than newline
nl	::=	newline | vertical tab | **;**

Appendix B

Description of all Processor Timing Parameters

This is the complete list of descriptions for the timed processor parameters in the PARASOL I machine model. Many of these entries were mentioned throughout this document and each of them is present in the BNF grammar in appendix A. Here the mnemonics are explained.

Table B.1: Description of all Processor Timing Parameters

absc	complex abs	**cld**	complex local division
absd	double abs	**cle**	complex local simple power
absi	integer abs	**clm**	complex local multiplication
absr	real abs	**cls**	complex local store
aimagc	imaginary part of complex number	**clt**	complex local memory transfer
aintd	double truncate	**clx**	complex local complex power
aintr	real truncate	**cmplx2d**	two doubles to one complex
anintd	double nearest whole number	**cmplx2i**	two integers to one complex
anintr	real nearest whole number	**cmplx2r**	two reals to one complex
argld	procedure call argument load (per byte)	**cmplxc**	complex to complex
arr1	one dim array index	**cmplxd**	double to real part of complex
arr2	two dim array index	**cmplxi**	integer to real part of complex
arr3	three dim array index	**cmplxr**	real to real part of complex
arr4	four dim array index	**conjgc**	complex conjugate
arr5	five dim array index	**dblec**	real part of complex to double
arr6	six dim array index	**dbled**	double to double
arr7	seven dim array index	**dblei**	integer to double
asind	double arcsine	**dbler**	real to double
asinr	real arcsine	**dga**	double global addition
atand	double arctangent	**dgd**	double global division
atanr	real arctangent	**dge**	double global simple power
cga	complex global addition	**dgm**	double global multiplication
cgd	complex global division	**dgs**	double global store
cge	complex global simple power	**dgt**	double global memory transfer
cgm	complex global multiplication	**dgx**	double global complex power
cgs	complex global store	**dimd**	double positive difference
cgt	complex global memory transfer	**dimi**	integer positive difference
cgx	complex global complex power	**dimr**	real positive difference
chari	integer to character	**dla**	double local addition
cla	complex local addition	**dld**	double local division

dle	double local simple power	**loopin**	n times loop increment
dlm	double local multiplication	**loopoh1**	one time loop overhead
dls	double local store	**loopohn**	n times loop overhead
dlt	double local memory transfer	**maxd**	double max/min
dlx	double local complex power	**maxi**	integer max/min
dprodr	double multiplication of two reals	**maxr**	real max/min
expc	complex exponent	**modd**	double mod
expd	double exponent	**modi**	integer mod
expr	real exponent	**modr**	real mod
gcom	computed go to	**nintd**	double to nearest integer
goto	unconditional go to	**nintr**	real to nearest integer
iga	integer global addition	**pcall**	proc call overhead
igd	integer global division	**realc**	real part of complex to real
ige	integer global simple power	**reald**	double to real
igm	integer global multiplication	**reali**	integer to real
igs	integer global store	**realr**	real to real
igt	integer global memory transfer	**rga**	real global addition
igx	integer global complex power	**rgd**	real global division
ila	integer local addition	**rge**	real global simple power
ild	integer local division	**rgm**	real global multiplication
ile	integer local simple power	**rgs**	real global store
ilm	integer local multiplication	**rgt**	real global memory transfer
ils	integer local store	**rgx**	real global complex power
ilt	integer local memory transfer	**rla**	real local addition
ilx	integer local complex power	**rld**	real local division
intc	real part of complex to integer	**rle**	real local simple power
intd	double to integer	**rlm**	real local multiplication
inti	integer to integer	**rls**	real local store
intr	real to integer	**rlt**	real local memory transfer
lga	global logic and/or	**rlx**	real local complex power
lgcc	global complex compare	**signd**	double sign
lgdc	global double compare	**signi**	integer sign
lgic	global integer compare	**signr**	real sign
lgrc	global real compare	**sinc**	complex sine
lla	local logic and/or	**sind**	double sine
llcc	local complex compare	**sinhd**	double hyperbolic sine
lldc	local double compare	**sinhr**	real hyperbolic sine
llic	local integer compare	**sinr**	real sine
llrc	local real compare	**sqrtc**	complex sqrt
log10d	double log10	**sqrtd**	double sqrt
log10r	real log10	**sqrtr**	real sqrt
logc	complex log	**tand**	double tangent
logd	double log	**tanhd**	double hyperbolic tangent
logr	real log	**tanhr**	real hyperbolic tangent
loopi1	one time loop initialization	**tanr**	real tangent

Appendix C

Glossary

ALU	Arithmetic Logic Unit
BNF	Backus Naur Form
CAMAS	Computer Aided Migration of Applications System
CSMA/CD	Carrier-Sense Multiple Access with Collision Detection
ESPRIT	European Strategic Programme for Research and Development in Information Technology
F77	Fortran, 1977 version
Fortran	Formula Translation, programming language
FWI	Faculteit der Wiskunde en Informatica, in English: Faculty of Mathematics and Computer Science
GCel	Giga Cube entry level
GEB	Gödel, Escher, Bach, see [Hofstadter79]
HPC	High Performance Computing
IEEE	Institute for Electrical and Electronics Engineers
IPS	Implicit Parallel Solver
MAP	Mapping tool
MD1	Molecular Dynamics benchmark one
MPI	Message Passing Interface
PAM-CRASH	Programmable Applied Mechanics for crash simulation An explicit finite element and finite difference code
Parasol	Parallel Solver
Parasol I	Parasol part 1, machine description
Parasol II	Parasol part 2, program analysis involving SAD
PARIX	Parallel extensions to UNIX, see [Parsytec93]
PET	Performance Estimation Tool
PIUI	Parasol I User Interface
PVM	Parallel Virtual Machine
SAD	Symbolic Application Description
SAD I	SAD, basic blocks
SAD II	SAD, flow control
SAD III	SAD, communication behaviour
SPE	Static Performance Estimator
SPMD	Single Program Multiple Data
Tcl	Tool Command Language, pronounced “tickle”

Tk	Tool Kit for widget programming in Tcl
TkTcl	A combination of two packages called Tcl and Tk, see [Ousterhout93]
UvA	Universiteit van Amsterdam in English: University of Amsterdam

Bibliography

[ACE93]	ACE, University of Amsterdam, ESI SA, ESI GmbH, FEGS, Parsytec, and University of Southampton. *CAMAS, Technical Annex Part II*, ESPRIT III Project no. 6756, Commission of the European Communities, November 1993.
[Addison93]	C. A. Addison, V. S. Getov, A. J. G. Hey, R. W. Hockney, and I. C. Wolton. *The GENESIS Distributed-Memory Benchmarks*, Department of Electronics and Computer Science, University of Southampton, 1993. *MD1 - Molecular Dynamics for a Lennard-Jones fluid benchmark*, by M. Pinches, May 1993.
[Andrews91]	J. B. Andrews and C. D. Polychronopoulos. *An Analytical Approach to Performance / Cost Modeling of Parallel Computers*, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Journal of Parallel and Distributed Computing, 1991.
[Boggs88]	D. R. Boggs, J. C. Mogul, and C. A. Kent. *Measured Capacity of an Ethernet: Myths and Reality*, DEC Western Research Laboratory, Palo Alto, September 1988.
[Birell84]	A. D. Birell and B. J. Nelson. *Implementing Remote Procedure Calls*, ACM Transactions on Computer Systems, February 1984.
[Dimpsey91]	R. T. Dimpsey and R. K. Iyer. *Performance Prediction and Tuning on a Multiprocessor*, Center for reliable and high-performance computing, University of Illinois at Urbana-Champaign, ACM, 1991.
[Dunlop93]	A. N. Dunlop and A. J. G. Hey. *PPPE Output 5.3a : Specification of Code Patterns*, Department of Electronics and Computer Science, University of Southampton, September 1993.
[Halderen94]	A. W. van Halderen. *Application and Machine Simulation*, Parallel Scientific Computing and Simulation Group, Faculty of Mathematics and Computer Science, University of Amsterdam, December 1994.
[Hofstadter79]	D. R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*, Basic Books, Inc., New York, April 1979.
[Linden94]	F. van der Linden. *The implementation of portable programming layers: a case study*, Parallel Scientific Computing and Simulation Group, Faculty of Mathematics and Computer Science, University of Amsterdam, June 1994.
[Markatos92]	E. P. Markatos and T. J. LeBlanc. *Shared-Multiprocessor Trends and the Implications for Parallel Program Performance*, University of Rochester, Computer Science Department, Rochester, New York, May 1992.

[Muller93] H. Muller. *Simulating Computer Architectures*, Faculty of Mathematics and Computer Science, University of Amsterdam, Amsterdam, 1993.

[Ousterhout93] J. K. Ousterhout. *Tcl and the Tk Toolkit*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1993.

[Parsytec93] Parsytec GmbH. *PARIX 1.2 manuals*, 1993.

[Rosen76] S. Rosen. *Lectures on the Measurement and Evaluation of the Performance of Computing Systems*, Purdue University, Philadelphia, Pennsylvania, SIAM, 1976.

[Röttger94] M. Röttger, U. Schroeder, and J. Simon. *Virtual Topology Library for PARIX*, Department of Mathematics and Computer Science, University of Paderborn and Paderborn Center for Parallel Computing (PC^2), June 94.

[Saavedra89] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya. *Machine Characterization Based on a Abstract High-Level Language Machine*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, IEEE Transactions On Computers, December 1989.

[Sarkar89] V. Sarkar. *Determining Average Program Execution Times and Their Variance*, IBM Research, Watson Research Center, Yorktown Heights, New York, ACM, 1989.

[Sunderam89] V. S. Sunderam. *PVM: A Framework for Parallel Distributed Computing*, Department of Math and Computer Science, Emory University, Atlanta, Concurrency: Practice and Experience, 1989.

[Tanenbaum88] A. S. Tanenbaum. *Computer Networks*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Index