Performance Prediction and Evaluation Tools

Author: Sergi Girona Turell
Thesis Advisor: Jesús Labarta Mancho

Departament d'Arquitectura de Computadors
UPC, Universitat Politècnica de Catalunya
Barcelona, Març 2003

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE
Doctor en Informàtica

A la meva mare
i a la Rosa M.

Abstract

Prediction is an interesting research topic. It is not only to predict the future result, but also to predict the past, often called validation. Applying prediction techniques to observed system behavior has always been extremely useful to understand the internals of the elements under analysis.

We have started this work to analyze the influence of several message passing application when running in parallel. The original objective was to find and propose a process scheduling algorithm that maximizes the system throughput, fair, proper system utilization.

In order to evaluate properly the different schedulers, it is necessary to use some tools. Dimemas and Paraver, conform the core of DiP environment. These tools has been designed ten years ago, but still valid and extensible.

Dimemas is a performance prediction tool. Using a single models, it capable to predict execution time for message passing applications, considering few system parameters for the system. It is useful not only to predict the result of an execution, but to understand the influence of the system parameters in the execution time of the application.

Paraver is the analysis tool of DiP environment. It allows the analysis of applications and system from several points of view: analyzing messages, contention in the interconnection network, processor scheduling.

Promenvir/ST-ORM is a stochastic analysis tool. It incorporates facilities to analyze the influence of any parameter in the system, as well as to tune the simulation parameters, so the prediction is close to reality.

The methodology on how to use these tools as a group to analyze the whole environment, and the fact that all those tools are State of the Art, demonstrates the quality of the decisions we made some years ago.

This work includes description of the different tools, from its internal design to some external utilization, the validation of Dimemas, the concept design of Promenvir, the architecture for Promenvir, the presentation of the methodology used with these tools (for simple application analysis to complex system analysis), and some of our first analyses on processor scheduling policies.

Acknowledgments

(In catalan, because most people in this section is used to speak with me in catalan. For those non reading catalan, if your name is in here, this means thanks)

Aquest treball es fruit del treball amb un conjunt de gent, començant per el Jesús Labarta que l'ha dirigit i aportat molts conceptes. Altres persones que han participat, d'una forma o altre en aquest treball son: Judit Gimenez, Cristina Pujol, Toni Cortes, Vincent Pillet (no va voler que Paraver es digués Somemo, ShOw ME MOre), Luis Gregoris, Maite Ortega, Jordi Caubet, Francesc Escalé, Jose Maria Cela, Jacek Marczyk, Xavier Masferrer, Toni Palau, Santi Bello. A tots gracies per el vostre ajut i recolzament.

A totes les persones que durant aquest anys han treballat amb mi al Departament d'Arquitectura de Computadors, especialment a tots aquells que m'han recolzat durant tots aquells anys. Als becaris i als administradors de sistemes del Departament i del CEPBA, per proporcionar-me moltes vegades la ajuda necessària per fer determinats treballs. Gràcies amics: Edu, Torres, Mateo, Marta.

Als meus companys d' EASi Engineering, que m'han recolzat a la part final d'aquest treball.

A la meva família, que m'ha ajudat a arribar on soc, i sense ells mai no hauria estat tant feliç. L'esforç que hi han dedicat per a la meva formació ha assolit un punt molt satisfactori.

I la Rosa M. també. Ella si que sap que es aguantar-me. Ha costat però ja està. Gràcies per l'ajut, l'amor, i per els nostres fills.

Aquest treball a rebut financiament de:

CICCYT TIC 94/0537-E, CICYT TIC 94-429, CICYT TIC 98-511, CICYT TIC 1999-1368-CE
APPARC BRA 6634, ESPRIT HPCN 20189, ESPRIT HPCN 26276

CHAPTER 1

Introduction

This work was started ten years ago. Yes, ten years. Our interest was focused in understanding the influence of sharing processor resources between several message passing applications, and to study possible scheduling policies.

Paros [77] was the resulting microkernel of the European project Supernode II. Paros allowed the execution of multiple message passing applications in a transputer base system. Transputers incorporate two process schedulers: FCFS for supervisor mode and Round Robin for user mode. Our interest was focused on the extension of these schedulers with new ones, to avoid possible contention in the application execution, due to synchronization between the different task of an application.

DiP is the name of the group of tools we developed to analyze the influence of different scheduling policies. DiP was composed by Dimemas and Paraver, a simulation tool and a visualization tool. Once we have this tools, it was clear that they are valid not only for those analyses, but to analyze individual applications to understand the application performance.

There are several reasons to focus our research in tools for performance analysis of message passing applications:

- Developing and tuning message passing programs is expensive both in programming time and hardware resources.
- Message passing parallel programming requires a significant effort before a first parallel version is available.
- Once this step is done, optimization issues have to be considered in order to achieve an efficient version.
- Depending on the complexity of the application, the effort required for this tuning can be significant.
- Issues such as proper data distribution, load balance and minimization of the communication have to be considered.

The availability of tools to ease this job, reducing the development time and increasing programmer productivity are one of the necessities clearly identified by code developers.

Let us explain in detail some of the problems one may have when developing message passing applications. Access restriction to desired target platform is frequently added to parallel programming development difficulties. It is not always easy to use a large platform to study the scalability of a given code, or a varied set of machines to evaluate the portability of a given application. The availability of such desired environment at a local level is clearly limited for a majority of developers by the cost of large parallel machines. Other related problem frequently encountered is the conflict between exclusive access to a system usually desired for application tuning and the high throughput that managers of such installations would like to achieve. It is really difficult to obtain exclusive access to those resources. And the shared access normally produces lower execution performance. Networks of workstations are the most commonly used development platforms, due to their availability and low cost as well as the simplicity offered by public domain versions of message passing libraries. Although very convenient for development, these platforms can only run efficiently a limited set of applications due to the communication performance and overheads typical of such networks. The last two observations constitute the major motivation behind the tools and environment described in this book.

The DiP environment aims at reducing the cost of parallel program development in time and hardware requirements. This is achieved by applying a set of tools that allow the use of sequential machines for development and tuning parallel applications. In the same way those finite element packages minimize the prototype and experiment needs, the DiP objective is not to eliminate tests on real machines but minimize and be able to focus them in a more selective procedure. Before actually testing the application on a parallel machine, the developer should be able off-line to optimize it for the target machine and have enough estimation of what the performance will be.

The DiP environment presented here was originated from the wish to study the problems and possibilities of short and medium term processor sharing policies on DMMPC (Distributed Memory Message Passing Computers) and networks running multiprogrammed loads of parallel programs. DiP also inherits from the initial project the following three level process models: application (Ptask), task and thread. This process model is a superset of the most frequently programming models and can overlapped them.

The functionalities offered by the DiP environment fall in the performance prediction, analysis and visualization areas. As such, it relates to a large set of available tools to help users to develop parallel applications. As prediction tool, DiP estimates the performances that a message passing application would achieve on different types of architectures ranging from workstation clusters to networks of SMPs and MPPs. As analysis tool, the special interest is twofold, first to obtain detailed quantitative statistics of an application run; and second in the effects of different factors (sensitivity) in the performances of the application. Two types of factors are available: target machine architectural parameters for simulator and code blocks or routines modification using a perturbation technique. Finally, visualization is conceived as a support mechanism for the analysis and prediction functionalities, providing flexibility and efficiency.

Although the DiP basic structure is similar to typical postmortem analysis tools based on traces, there are some specific elements and design criteria that differentiate our approach from others. The core of the system lies in a clean and tight integration between three tools: an instrumented library, a simulator and a visualization tool. Among the important issues to achieve a good balance between prediction accuracy and tool efficiency, there is a parameter and feature selection that characterize an application, a system and their interaction. As described in the paper, the type of information included in the traces aims at characterizing the parallel program. This is used instead of one instance of its execution on a target machine. The sufficiently detail fine grain level of this characterization allows to the simulator an accurate efficient estimation of the behavior on a target machine. For the efficiency of the simulator it is also needed the target machine characterization by a set of parameters that models the architecture in a simple way. A special attention was put to carrying out detailed and accurate quantitative values during analysis and visualization. Unlike other many system, DiP is able to provide those values in only few views.

Two objectives in the design were: to emphasize a clear division between parts of the tool set where each module has its own functionalities, and to offer flexible mechanisms to combine those modules leading to construct very powerful analysis and prediction functionalities. From these simple concepts, the tool enables complex and large application analyses.

Dimemas and Paraver are still under development, and have been used for different research topics, including process scheduling and distributed file systems. Although DiP was designed about ten years ago, the concept is still valid, and can be extended to cover current research topics. And this gives a lot of value to it.

Ten years is a long period of time, and apart of the evolution of Dimemas and Paraver, to provide support for current application analysis, we have participate in the design and development of Promenvir, a tool for stochastic analysis. In combination with any simulation tool (Dimemas or Paraver in this work), it provides a very valuable insight of the models, applications, simulators,...

The different chapters of this book are organized as follows

- Chapter 2 describes the most important related work, including performance prediction, performance analysis and robustness management.
- Chapter 3 focuses in Dimemas, a simulation tool for performance prediction of message passing applications. It includes a full description of the model used as well as a full description of all parameters.
- Chapter 4 corresponds to Paraver, the visualization tool for performance analysis, initially focus on message passing applications.
- Chapter 5 describes Promenvir, as robustness management and optimization tool developed in the scope of an European Project. This tool helps in the performance analysis phase, as it clarifies the relativeness of some simulation parameters.
- Chapter 6 contains the validation for Dimemas, using Paraver and Promenvir.
- Examples on how to use and perform analysis with the different tools are presented in Chapter 7.

- Chapter 8 analyze the influence of different low level process scheduling policies.
- And, Chapter 9 concludes with conclusions and possible future work.

CHAPTER 2

Related work

This chapter describes the related work to the different areas covered in this book, including performance prediction, performance analysis, and robustness management. It is important to remark that some of this related work is newer than the tools and methods we have proposed, but the functionalities and potential of out tools and environment are still superior is many respects.

2.1 Performance prediction

In order to analyze the performance of an application it is necessary to take into account the hardware and the software. In the hardware group we include processor, memory, inteconnection network,... and in the software group we consider the application, the communication library and the operating system.

2.1.1 Models for multiprocessors

In order to model multiprocessor systems, is necessary to measure the performance of the processors, the interconnection network, the memory,... as well as the interaction between this different modules or parts. The reader can get a more detailed information on related work in some of the references specially in [1][2].

1. PRAM model

   Parallel Random Access Machine is a set of P processors with a shared global memory [3]. This model consists in a MIMD system, where each processor executes its own instructions and access data in memory. Different PRAM models considers different situations related to memory contention for concurrent reads and writes.

   Although this model is quite different than todays computers, they can provide interesting information regarding parallel programs.As inconvenient, it does not take into account the interconnection network, but the utilization of these resources in very important for the application performance.

There are several variations on this model, including CRCW PRAM (permits concurrent read and write, and have rules to describe how to fix memory access conflicts. The concept for serial access to memory is covered with the EREW (read and write access) and CREW(write only).

The PRAM model has several variations to adapt to actual parallel machines, including LPRAM (Local-memory PRAM), BPRAM[4][5], phase PRAM[6], XPRAM[7], or the Delay Model[8].

2. Models based in interconnection network

Opposite to PRAM models that ignores completely the inteconnection network, there are those models based in interconnection networks that specifies all level of details. There are as many models as different network topologies. An extended description of these models can be found at [9].

Some of those models includes a description of network performance using network latency, and including this latency as a component of an equation in the model[11]. This latency is described as the time to reach destination node for the first byte of the message (data). Every topology has a different latency computation as the number of nodes is directly related to it.

3. Bridging models

An intermediate solution for models, is bridge models. The interconnection network is described but just with few parameters.

• Postal Model

Introduced by Bar-Noy and Kipnis[12], the system consists of N nodes, processors with local memory. Each node can simultaneously use a single input port and a single output port, and the communication latency is λ. Namely, each node during each communication step can send a fixed-size packet and at the same time receive a packet of the same size. Furthermore, a packet send at step t will incur a latency of λ and will arrive at the receiving node at time $t+\lambda-1$. This model has been successfully used to analyze the performance of two collective communication patters, the broadcast and the global combine[13].

• LogP Model

The LogP model is a generalization of the Postal model. It reflects the convergence of parallel machines towards system formed by a collection of complete computers, each consisting of an off-the-shelf processor, cache, and a large main memory, connected by a communication network.

Since there appears to be no consensus emerging on the interconnection topology, LogP avoids specifying the structure of the network and instead recognizes three parameters. First, interprocessor communication involves a large delay, as compared to local memory access. Second, networks have limited bandwidth per processor as compared to the local memory or local bandwidth. This bandwidth may be further reduced by contention at destination. Third, there is a cost to the processors involved at both ends of the communication event; this cost is independent of the transmission latency between processors.

Specifically, LogP is a model of a distributed memory multiprocessor in which processors communication through point-to-point messages and whose performance is characterized by the following parameters: *L*, an upper bound on the latency, or delay, incurred in communicating a message containing a small fixed number of words from its source processor/mem-

ory to its target; *o*, overhead defined as the length of the time that processor (during this time the processor can not perform other operations); *g*, gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor (is the inverse of communication bandwidth); and *P*, the number of processors/memory modules. *L*, *o*, and *g* are specified in units of time. The parameters are not equally important in all situations; often it is possible to ignore one or more parameters and work with simple model[14][15]. The model also assumes small message size *w*, although some modifications includes variable message size[16] and analysis of network contention[17], but those modifications includes network topologies and this reduce the generality of LogP.

LogP is an excellent model to optimize latency-bound communication patterns. The latency-bound patterns are those forms of collective communication that can be organized in a sequence of contention-free steps and are thus limited by the injection overhead and the bus network latency. In the absence of contention, the two parameters, L and o, can be properly estimated and the model can lead to important and effective optimizations. Examples of LogP usage are shown in [18], [19], [20], and [21].

• BSP Model

The Bulk-Synchronous Parallel was proposed by Valiant as a bridging model that provides a standard interface between the domains of parallel architectures and algorithms. In the BSP model, a parallel architecture consists of a set of processors, each with its own private memory, that execute some virtual processes, and an interconnection network that ca route packets of some fixed size between processors[22].

2.1.2 Models for applications

Performance prediction is a very important issue in order to obtain good performance in parallel applications. As this performance analysis is very complex, due to the complexity of the parallel application itself, it is necessary to use performance analysis tools in order to develop properly a parallel application.

In order to model applications, there are three different possibilities: analytical, simulation data, and traces[25]. It is also possible to have mixed solutions, where different characteristics of the different model are used.

1. Analytical model

The concept for analytical models is to represent the application as well as the parallel architecture using analytical models. This models permits to analyze applications independently of the architecture. These analytical models are very fast and effective compared to different modelization techniques, because they are based in mathematical equations, but the quality of results is very dependent on the parameter selection, and is very sensible to any assumption and simplification included in the model.

Some of those analytical models are based in complex algebraic expressions, that tries to get the characteristic parameters for the application model[26][27][28].

There are different analytical models based on statistics studies using Markov, queue models[29][30][31] [32]and Petri Nets[33][34].

2. Simulation model

   A simulator is a program that emulates the behavior of all the subsystems of the parallel architecture. It takes into account, not only the application but also important parameters of the architecture: cache failure, flops, network parameters,...This evaluation methodology is well suited for performance evaluation of non accessible platforms, because they permit the evaluation of parallel applications in those systems.

3. Tracing

   The bests method to analyze system performance is using the real application running in the real system. It is necessary to have specific tools as library tracing and analysis tools.

   Normally this methods are used to extract the parameters to use in analytical models, and simulation.

2.2 Tools for analysis, visualization and performance prediction

There are several reviews on tools for performance analysis, including [35] and [36], and this demonstrates the wide variety of tools and all possible and different points of view. We will present here a subset of these tools:

2.2.1 Visualization and Analysis

1. AIMS (Automated Instrumentation and Monitoring System) [37][38]is a set of tools for performance analysis of C and Fortran message passing programs. Supported libraries are NX, PVM and MPI. It includes an instrumentation library (AIMS), a portable communication library (CDS), a trace visualization tool (NTV) and a parallel debugging tool (P2D2).
2. MPP-Apprentice [39] Cray proprietary tool, available to T-series (T3D, T3E). Using a tracefile and the call tree, it shows computation time in sequential model, shared memory and message passing operations. It highlights the load unbalance and the incorrect utilization of memory hierarchy.
3. EMU and ATEMPT [40], monitoring and visualization tool, respectively, of MAD (Monitoring and Debugging Environment) developed at Linz University (Austria). Main goal is to provide debugging facilities.
4. MEDEA [41] Developed in Pavia University for workload characterization on parallel computers. Automatic pattern recognition is provided using statistic methods, for example, it is capable to show the different clusters in communication for a message passing application, grouping by size and message tag. This clustering permits also the detection of outliers, abnormal communication within a pattern.
5. NTV [38] (NAS Trace Visualizer) is the visualization tool for those traces generated with MPL library on IBM SP2 or AIMS.
6. Pablo, SvPablo [42], analysis and performance visualization for message passing application and HPF/Fortran D. It uses its own tracing library, and an module to extract automatically data from the compilers HPF. Trace file format is

SDDF, so it is also capable to read trace files generated on PVM, MPI and PICL.

7. Paradyn [43] for dynamic instrumentation using the library called *dyninst*. This dynamic tracing reduces dramatically the trace file size, as it only contains data relevant to the important phases of the application.
8. PARADE [44] visualization environment on application execution time for application on parallel and distributed system. To remark the tool POLKA for visualization of programs coded in different languages and for different architectures. Is supports simulation of parallel programs running on distributed systems (PVaniM-GTW).
9. Paragraph and Paragraph+ [45] post-morten analysis tool for performance visualization. Uses the trace files generated by PICL message passing library. It has several views and diagrams: Gantt, Kiviat, communications,...
10. PAT (Performance Analysis Tool), Cray proprietary tool. Used in combination with Apprentice for C, C++, and Fortran 90 code instrumentation for T series. It provides information at hardware level, and includes an instrumentation subroutine to include additional information in the trace file.
11. PETSc Viewers [46] is the part of the PETSc toolkit for iterative methods. Include facilities for code instrumentation of the numerical libraries included in PETSc, and to obtain tracefiles jointly with the MPI instrumentation libraries. Can generate traces for upshot/jumpshot[47].
12. PGPVM [48] monitoring took for PVM [49]. It generates trace files readable for ParaGraph. It tries to minimize the influence due to instrumentation, using the internal libraries of PVM as well as some internal buffering methods.
13. PMA [50], component of Annai, an integrated environment for development of parallel applications using HPF, with the possibility to include explicit messages using MPI. It shows HPF data distribution and additional data automatically obtained from the parallelization tools and the compiler integrated in the Annai environment. It shows processor utilization and communication events, with a reference to the source line related to it.
14. Tape/PVM and PROVE [51] a performance visualization tool which is part of the GRADE integrated program development environment for message passing programs. It was developed based on the shared-memory PACvis visualization tool. PROVE can analyze event traces generated by the Tape/PVM monitoring system. The instrumentation of the parallel program is done automatically by the programming environment but it can be customized by the programmer as well.
15. Nupshot/upshot/Jumpshot [47]. Three different versions for visualization of the traces generated with MPE, instrumentation library included in MPI MPICH [52] from Argonne National Laboratory. The latest based is Java based, and includes a better instrumentation for collective operations, and highlighting of anomalous timings, that can arise some bottlenecks in the application.
16. VAMPIR [53], displays post-mortem tracefiles through a number of graphical views. Flexible filter operations reduce the amount of information to be displayed, and rapid zooming and forward/backward motion in time allows quick focus on arbitrary time intervals. Supports MPI, PVM and PARMACS. It was developed by KFA Jülich and is commercially available from Pallas GmbH.
17. VT, used for development of parallel applications using AIX for SP-1 and SP-2 systems. This tool has been replaced with a set of tool, PE Benchmarker suite

[54], which integrates monitoring capabilities for message passing libraries as MPL and MPI, and HPF compilers to generate trace files with a unique file forma (Unified Trace Environment, UTE). These traces are readable by Jumpshot.

18. XMPI [55] a post-mortem tool featuring graphical representation of the program's execution and message traffic. Uses a special trace produced by the LAM-MPI environment.
19. XPVM [56], graphical interface for PVM, providing a graphical version of the PVM monitor and also animated viewing of PVM programs during execution.

2.2.2 Performance Prediction

Several prediction tools have been developed in the recent years. In this section we present a list with a short description for each.

1. ALPSTONE Performance-oriented tools to guide parallel programming. The process starts with an abstract BACS (Basel Algorithm Classification Scheme). This is in the form of a macroscopic abstraction of program properties such as process topology and execution structure, data partitioning and distribution descriptions and interaction specifications. From this description it is possible to generate a time model of the algorithm which allows performance estimation and perditions of the algorithm runtime on a particular system with different data and system sizes. If the prediction promises good performance, implementation and verification can start. This is helped with the skeleton definition language (ALWAN or PEMPI - a programming environment for MPI), which can derive a time model from the BACS abstraction and a portability platform (TIANA) which translates the program to C with code for a virtual machine such as PVM.
2. Carnival [58][59] is a tool designed to automate the process of understanding the performance of parallel programs. It supports performance measurement, modeling, tuning, and visualization. Carnival measurements are based on predicate profiling, which quantifies the time spent in each category of overhead during execution.
   Carnival is a novel attempt to automate the cause-and-effect inference process for performance phenomena. In particular, Carnival currently supports waiting time analysis, an automatic inference process that explains each source of waiting time in terms of the underlying causes, instead of simply identifying where it occurs. A similar technique is being developed to explain the causes of communication.
3. EXCIT and INSPIRE [60] simulation tool for large-scale data-parallel applications. The code is divided into computation and communication sections allowing independent simulation of cpu-dependent and network-dependent effects. The instrumentation tool EXCIT is used to compute execution time on the target hardware and generate message traces. Communication time is generated using the message traces and a network simulator system INSPIRE. Predictions have been tested using the NAS parallel benchmarks.
4. P3T [27] tool for performance prediction using the HPF VCFS compiler (Vienna Fortran Compilation System). Uses analytical model for performance prediction. The parameters used in the model come from source code analysis.
5. PAMELA [61][62]. Methodology for the development of parametrized performance models with low complexity for parallel applications running in shared

memory and distributed memory. The methodology is bases in the PerforMance ModEling LAnguage (PAMELA). This permits to characterized the application and the system, compile it and generate a symbolic performance model.

6. PACE Toolset [63][64] (Performance Analysis and Characterization Environment), aims to extend traditional use of performance prediction to the full software development cycle. Facilities for pre- and post-implementation analysis. Modular performance models reflect parts of the system (software, communications, hardware parameters). Predicted traces can be produced in either PICL or SDDF format.
7. PerFore [65], Performance Forecasters, extracts compile-time knowledge from understanding of run-time performance to enable prediction of execution times. Symbolic analysis is used to relate dataset dimensions and machine parameters to loop ranges. Explicit communication and i/o are located. Summed computation, communication and i/o times are extrapolated as a function of number of processors.
8. PARASIT [40],PARAlle SImulation Tool, include in MAD environment, the main goal is the detection of race conditions. The interesting question is what would have happened if a message had arrived before another one? The detection of race conditions in the MAD environment for debugging parallel programs is based on the assumption of equivalent program execution: Two executions of a process P are considered to be equivalent if the process P receives the same information from the other processes at the same instants. The instant of an event is defined by the interval in which only this event takes place. This means that two executions of a parallel program will be considered to be equivalent if the execution of each of it processes is equivalent.
9. PERFORM y LEBEP[31]. PERFORM is a performance generator for RISC microprocessors. It is a general-purpose package for sequential performance estimation in complex memory hierarchies. An intermediate level of abstraction is used between simple statement counting and full node simulation. LEBEP generates synthetic parallel programs from a simple communication specification. Data movement across different levels of memory hierarchy can be simulated.

2.3 Robustness Management

In this area, the first tool available to cover non specific solver, was Promenvir. Thereafter several tools has grown worldwide covering the same topics, or similar. Here follows a short list and description of those tools:

1. iSIGHT [66]
   captures, automates, integrates, and optimizes existing design environments accelerating the design process. iSIGHT functionalities:
   - ability to quickly couple design/simulation tools to iSIGHT users can rapidly couple virtually any code that uses ASCII input and output files
   - support for CORBA-compliant design and simulation codes
   - Ability to automate processes by constructing and graphically representing hierarchical, multi-task program sequences

- Ability to interactively formulate, execute, monitor, and visualize various design and analysis operations based on the automated process
- "What-if" analysis of design alternatives
- Parametric trade studies
- Design of Experiments (DOE) studies
- Optimization design studies
- Ability to construct and use behavior (approximation) modules, such as response surface and sensitivity-based models for high fidelity simulation tools
- Best-in-class optimization algorithms are integrated into iSIGHT
- Numerous numerical, exploratory and heuristic search algorithms
- Hybrid optimization plans combining search strategies
- Quality Engineering Methods such as Monte Carlo simulations, reliability based design analysis and Taguchi's robust design

Support for custom optimization techniques

- Ability to define rules that capture expert knowledge within the automated process to further accelerate the design/analysis process
- Numerous post-processing and visualization tools provide users with further insight into the design
- Ability to use API's to extend and customize iSIGHT
- Flexible wrapping applications to allow user to eliminate file input and output through use of C API calls
- Ability to perform multidisciplinary optimization with various formal MDO algorithms
- Checkpoint and restart capability
- Database tolerance lookup
- Distributed Job submission
- Able to work with persistent simulation codes
- Services to support parallel processing

2. LMS-Optimus[67]
   LMS OPTIMUS, from LMS International, Belgium, is a Computer Aided Engineering (CAE) product for multi-disciplinary optimization. LMS OPTIMUS manages multiple simulators to achieve an optimal design. The program drives and manages the exchange of data between CAE simulation tools, and modifies the product design based on the simulation outputs until an optimal design is found. The number and type of design variables, simulation software tools, and design objectives is unrestrained, specified by the user. There is no restriction on the design process that LMS OPTIMUS addresses - anything that can be simulated can be optimized. LMS OPTIMUS solves the most complex multi-objective multi-disciplinary optimization problems:
   Automate the design-analyze-change cycle Unique interfacing capabilities to ANY analysis at the user level without programming. Free the user from updating input files, submitting simulation jobs and extracting results from output

files. Use mathematical reasoning to iterate on the design according to the specified Design Objectives

3. HyperOpt [68]
While several commercial codes currently offer optimization capabilities with their linear analyses modules, optimization with non-linear analyses codes is practically nonexistent. Altair® HyperOpt™ fills that void by performing design optimization, parametric studies, and system identification in conjunction with ABAQUS® non-linear finite element analysis code.
HyperOpt can be used to perform sizing and shape optimization in conjunction with any analysis code that has an ASCII Input and Output. In addition, HyperOpt has been used in the optimization for nonlinear problems such as crash worthiness design, multi body systems simulation, metal forming simulation, CFD, and also for multi-disciplinary problems.

4. BOSS Quattro [69]
From Samtech, Belgium, is dedicated to analyzing and optimizing the influence of parameters on responses yielded by external software. On the one hand, the analysis leads to display the response curves with respect to the parameters. On the other hand, BOSS Quattro also performs a sensitivity analysis (to compute the derivatives of responses with respect to parameters), and searches for the optimal parameter values in order to satisfy some criteria on the responses.
BOSS quattro can combine those capabilities with Monte Carlo simulations: BOSS Quattro evaluates the statistical distribution of the responses when the parameters follow a likelihood law. The software controlled by BOSS quattro may differ in the nature: the open architecture of BOSS quattro is able to interact with complex CAD/CAE chains, and is virtually connected to any software through a driver system.
An application manager for parametric studies, multidisciplinary optimization, and Monte Carlo simulations As an application manager, BOSS Quattro controls the execution of external applications and interacts with them. Fully "menu driven", this user friendly tool makes easy the definition of process loops and tasks chaining. The simultaneous use of your models (CAD, finite elements or any other type) is then straightforward. For BOSS quattro, an application is like a "black box" which receives parameters and gives back responses. The aim of BOSS Quattro is to manage and exploit automatically this information through the use of specialized drivers.
The parametric analysis is the basic capability of BOSS quattro. Using the driver system, BOSS quattro reads the available parameters from models and collects the analysis responses ("functions") after the application run. The menus help the user to define the parameter values, and the responses of which he wishes to trace the evolution. After an automatic computation stage (during which BOSS quattro updates the models, runs the analyses, filters the results and stores them in its database), the curves and tables help the user to get more insight on the behavior of his model and guide him through the design process. The "functions" handled by BOSS quattro are built using the model's responses (contained in the result files). BOSS quattro displays the available information in its menus. The user can then define his functions, step by step, by determining their nature, location and reference. A second level (called function "parser") is available to combine these "direct" informations for building new criteria whose value is then computed by BOSS quattro. Using this facility, one is able to mix parameters and functions in the same analytical expression.

2.4 Analsysis

How do we compare to the other tools, methods and methodologies?

Dimemas for an external point of view is similar to LogP. It uses latency and bandwidth to model the interconnection network, and also models contention. But Dimemas uses tracefiles to model the application behavior, and the contention is based in the real communication pattern of the application. This allow a more realistic analysis.

Paraver is, from the external point of view, quite similar to others. The difference come from the internal design and analysis capabilities, including as many levels as necessary to represent nodes, processors, applications, tasks, treads; and different methods to analyze them.

Promenvir is one of the most advanced tools, including an internal queue system to distribute the workload of the necessary simulations over different machines. It also includes simple but effective methods for Stochastic Design Improvement and Robust Design.

CHAPTER 3

Dimemas: a simulator for application analysis and performance prediction

This chapter describes Dimemas, a performance prediction tool. Its name comes from DIstributed MEmory Machine Simulator, although it is important to remember its Spanish origins (Dime Más, equivalent to "tell me more"). Dimemas is a trace driven simulator that rebuilds the behavior of a parallel program from a trace file and some parameters of the target architecture.

First Dimemas version was dated in 1992. Originally, Dimemas was developed to analyze different processor scheduling policies when applied to a multiprogrammed environment for message passing applications. CEPBA was one of the participants in the Esprit Project Supernode II. One of the major goals of the project was to develop a kernel to support multiprogramming for Transputer based machines. Transputers incorporate two different scheduling policies: FCFS (First Come First Serve) for high priority threads and Round Robin for low priority threads. The interest for Dimemas development was to measure the influence of the scheduling policy and the different parameters, but also to analyze and propose new policies.

Once the tool was available with a stable version and connected to the visualization tool (Paraver), the debugging and analysis information for single applications was demonstrated. The possibility to get a tracefile in a shared environment and predict the application behavior with Dimemas allows application tuning and development. Then, it was extended to support different topics such: file system models, shared memory facilities,... in order to analyze all of them jointly and separately.

Dimemas is based on the characterization of the architecture and the characterization of parallel applications, and these are two contributions of our work. This application characterization is based in recording the most important characteristics of the application in a tracefile. Related to the architecture model, we propose to simplify it with a set of parameters that provides accuracy results compared to the complexity of the models. This architectural model not only incorporates the hardware characteristics, but also the characteristics of the middleware. In our studies, this middleware refers to the communication library (MPI, PVM) and its specific implementations.

This chapter is organized as follows. First, Dimemas programming model and architectural model will be described. Second, we will focus in the proposed instrumentation mechanism. Third, the different communication models and the different parameters they support will be analyzed. And last, some extra functionallities for application analysis are described.

3.1 Architectural model

The architectural model employed in Dimemas is presented in Figure 1. It is based in a network of SMP's (Symmetric MultiProcessors).

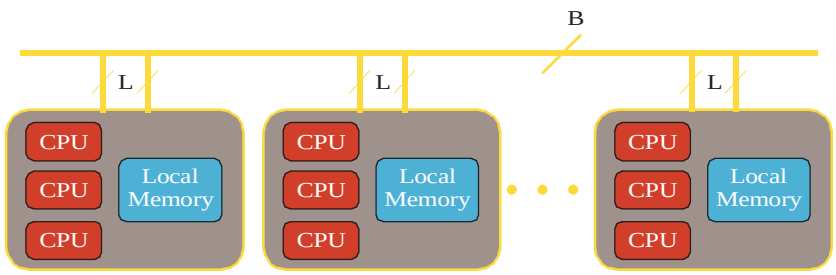
Each node is composed of several identical processors and a local memory to this node (this memory is not accessible from outside the node). The connection to the network is based in links: input links for incoming information and output links for outgoing data.

The network model is accurate because uses the most important parameters that influence the efficiency of the network. At the same time, it is simple because uses few parameters. This simplicity of the model permits a very fast execution time for the simulator, which is in most case a problem for the simulators.

The good decision of using a single representation of the system is confirmed with the good quality of results provided compared to a low simulation cost. The quality of the results and the cost of the simulation are presented in Chapter 6.

Next sections describe the different parts of the model represented in Figure 1.

Figure 1. Architectural model



3.1.1 Interconnection network

The interconnection network model includes the interconnection network itself and the mechanism to connect each node to the network. It has two principal components modeling the network bisection and the injection mechanism.

To model the bisection bandwidth of the system, a maximum number of available buses (defined by the user) are considered by Dimemas. We consider a bus based network, with a fixed number of buses, and represented with letter B in Figure 1. This allows us to model networks of workstations connected with a single bus or a system with full connectivity assuming as many buses as nodes. Parameter B fixes

the total number of communications (messages) that can use simultaneously the network. If there are more communications than resources, Dimemas block the communications exceeding the capacity until free resources become available.

Also, to model the injection mechanism, a number of input and output links between the nodes and the network can be defined. Half-duplex links can also be specified. Each link represents the connection to a network as a network card to a real machine. Links are divided in input and output ones. For a communication involving different nodes, it is necessary to acquire two links (output link at source node and input link at destination node) and a network bus. If some of the previous resources are not available at that time, the communication is delayed until the resources become available.

By default, links are considered full-duplex. If a node has one input and one output link, two communications can be performed simultaneously. Dimemas also considers half-duplex links. In this situation, a node having a single link can only send/receive a message at a time, because the link is necessary for sending/receiving the message.

To model point to point communication, Dimemas uses the following equation

$$T = L + \frac{S}{B}$$

where L is the latency, S the size of the message and B the bandwidth. This equation can be applied once acquired the resources.

3.1.2 Nodes

Each of the nodes of the system is composed of a set of identical processors and a memory shared by all processors. Different nodes may have different processors, as network of workstations or a network of shared memory multiprocessors, with different characteristics. Four different parameters characterize each node: communication bandwidth for intranode communications, communication latency for intranode communications, communication latency for internode communications and relative processor speed. We can also consider processor scheduling policy as a characteristic of a node, although is our studies we have consider always the same scheduling policy to all the SMP nodes. The processor scheduling policies are described in Section 3.6.

For each node, Dimemas uses an internal communication bandwidth in case there is a communication involving different processors but in the same node. This situation is applicable for implementations of the communication library based in shared buffers.

Latency is the time spent in the preparation of a communication, corresponding for example to the time necessary to setup the internal buffers. This parameter depends on the specific implementation of the communication library, but it is also different for internode communications and intranode communications. Dimemas allows the definition of both parameters separately.

The last parameter is the relative processor speed. This parameter measures the different relative speed of two processors. This is the ratio of the time spent by the same program in different processors. This is by sure, a simplification of a comparison of two processors. We know this is not the best way to compare two processors, because there are many differences that are not considered in our approximation. Furthermore, this is not true for all applications (the same value is not valid when using different applications) and for different memory size (the amount of memory requested by applications). But, we claim for using simplified models for processor comparison instead of very complex ones, because prediction is good enough for the objectives we propose for the tools. Results presented in Chapter 5 will demonstrate the correctness of this hypothesis. Given this parameter, Dimemas predicts the execution time of an application of a new processor when the tracefile has been obtained in a different one.

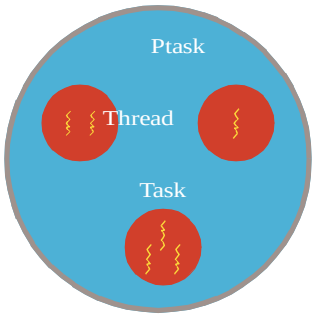
For all the previous parameters, the mechanism to obtain the proper value is benchmarking the target machine. In some situations, this benchmarking is not possible because the target machine is not available. For those, the vendor values must be used to predict the application behavior. The different values obtained in benchmarking those parameters as well as effects of using vendors parameters are presented in Chapter 6.

3.2 Process model

The process model used is the one defined in Paros, having three different levels: Ptask, task and thread. This is represented in Figure 2. This model allows the specification of both programming models, message passing and shared memory.

The Ptask concept encapsulates a single parallel program in execution. It is the entity to which resources are allocated. A Ptask consists of a set of tasks. A task is a logical address space in which several flows of control can execute. A task is mapped on one node within a Ptask, thereby all its threads will run on any processor in the node. A thread is a sequential flow of control within a task. All the threads in task share the same address space.

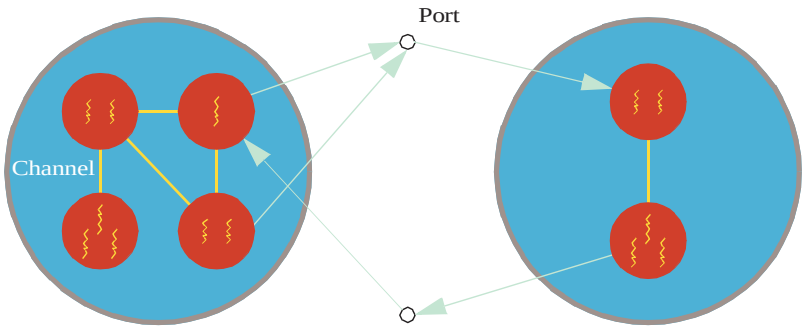
Figure 2. Process model



3.3 Communication model

Dimemas models two different message passing interfaces: port and channels. Ports allow the communication involving threads of different applications or Ptasks. This is the mechanism employed to communicate with servers. Channels allow the communication involving threads of the same Ptask, but from different tasks. Both models are presented in Figure 3. Threads from the same task use shared memory to communicate and this is out of the scope of Dimemas, except for synchronization primitives.

Figure 3. Communication model



Additional communication mechanisms have been also included in Dimemas: remote memory access and shared memory synchronization. MPI2 standard includes one-sided communications, referring to operations to get memory content from remote nodes. Dimemas can support two different implementations of these operations: using servers and port implementation or simply using remote memory access. Although synchronization using shared memory is not a communication itself, it has been included in this section because it allows some kind of communication between threads of the same task.

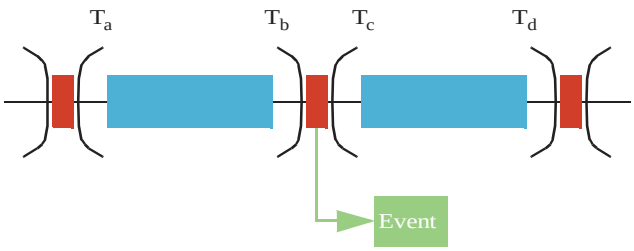
3.4 Application characterization: Instrumentation libraries

One of the contributions of this work is the instrumentation mechanism, in other words, which information is necessary to store in a tracefile with all the necessary information of the application.

The mail goal of the instrumentation libraries is to store in a tracefile all status records and all the events that characterizes the application. As we will use a tracefile to predict the behavior of the application on a different machine than the instrumented, it is necessary to eliminate all effects of the machine used for instrumentation.

Application characterization is based in two different records: states and events. This is presented in Figure 4. State events refers to resource request and duration (in this case processor time). It is shown in blue. The information recorded in this case is: (T_b-T_a) and (T_d-T_a). Processor time is stored as CPU time consumption, not including idle time because the process has been scheduled out. Event records represent instantaneous events in between state events. There are two classes of event records: user defined and system defined.

Figure 4. Tracefile record



User defined event record is the mechanism to generate extra information of the application, for example call stack, values for variables, location of source code,... In some cases, a precompiler can generate automatically the necessary library calls to generate this records, but usually the user is requested to modify the application to include those calls. In Figure 4 these records have red color.

System defined events are those related to communication and MPI I/O. Presented in red color with a green label in Figure 4. These records refer to communications involving different tasks, and the instrumentation library automatically generates them whenever a communication point is encountered. Although these records contain information about tasks involved in communication and file system access, we do not include any absolute time, because absolute time depends on an specific application execution. Dimemas uses the information stored in these records to rebuild the communication or file system access (using the models), and then reconstruct the whole application.

For example, if the tracefile is obtained in a single workstation (with a unique processor) and the wall time is annotated, the tracefile will contain the serial execution of the different tasks. This serial execution is based in the process scheduler and in the message dependencies.

Dimemas annotates CPU bursts (CPU time consumption) between two consecutive events, and the semantics of the events (communication of a given size involving two different tasks). CPU burst annotation eliminates process scheduling serialization, although there is some variation from reality due to cache influence. Whenever an event record is encountered in the tracefile, Dimemas schedules it properly, according the available resources.

The tracing instrumentation is built into the message passing library avoiding the need of recompilation and making possible to use it with applications where only object files are available.

To control the level of tracing detail, some functions have additionally been defined to support on the fly adjustments to each trace event mask of the tasks (switching it on and off, anytime, anywhere). Other functions offer the possibility to introduce user defined event records in the trace which can be used to tag specific "events" in the code. These user events are used to mark the entry and exit of routines or code blocks.

Other significant feature of the tracefile is the support of multithreaded applications. The different threads within a task share the address space and can only synchronize themselves through semaphores. This model is reflected in the trace as follows: each application has a single tracefile where each record specifies the task and thread whose behavior it describes. Additional records are included to identify the semaphore wait and signal operations that threads can use for local synchronization within a task.

This model is more general than those ones supported by most communication libraries today, and it is on the way of the evolution for different programming models. For example, in PVM each task is single threaded and several features of the interface greatly increase the difficulty of the multithreaded execution. In MPI more attention has been paid to these interface definition issues but available implementations are normally single threaded.

3.4.1 Shared memory prediction

A nice feature of our instrumented libraries is that we offer a functionality to mark blocks of a sequential code which could be executed as different threads, sharing the processor if just one is available or in parallel if the task is run on SMP mode.

This feature requires the user to do the dependence analysis and properly insert the instrumentation call but this cost is less than that of actually implementing the multithreaded version. This approach offers the possibility to evaluate how much would the effort of moving to a two level parallelism cost be, in terms of performance benefits. This type of evaluation would also be very useful if the target platform (network of SMP's) is not available.

The mechanism description follows: whenever the execution arrives a parallel loop, instead of generating a single trace record for all the loop, the user can instruct the instrumentation library to create a record for each thread, and, for example, dividing the total cost of the loop into all the threads records. And the end of the block is required to include also a synchronization record in the trace.

3.4.2 Tracing accuracy

An important issue in the implementation of the tracing facility is the precision and accuracy in the measurement of the duration of the CPU bursts. A microsecond precision is desirable for the appropriate characterization of many message passing programs, because it is usual to have minimal computation between bursts of consecutive communications.

Proper accuracy of the time measurements is required in order to achieve a good reconstruction of the parallel program behavior. Furthermore, this measurement has

to be available on a per process/thread basis. Operating system functions or direct access to high resolution clock is performed, depending on the machine. For machines with less accurate clocks, the usage of this tool may be restricted to applications which have a sufficiently coarse granularity.

3.4.3 Tracing overhead. Probe effect.

A central problem to all measuring instruments is one of observance: it is difficult to determine the precise behavior of a set of concurrently executing tasks. Heisenberg tells us that we simply cannot examine the behavior of such a system without perturbing it. This is know as the probe effect. Therefore, the goal of effective parallel program analysis is to provide information that is as accurate as possible without significantly intruding on the parallel program. Our approach only needs to measure the duration of CPU bursts which results in the elimination of the typical problems encountered because of recording the absolute time of the events. In Dimemas we do buffer the records in memory before writing them to disk, but this is only done for I/O performance reasons and not to avoid interference with the running application. The size of the buffers can thus be relatively small. The overhead of the timing routines is compensated by measuring their duration when the tracing is turned on, and subtracting it from the measures obtained.

Once the I/O overhead, which is the most important one in other approaches is overcome, cache related overheads remain. This includes pollution of code caches as well as data caches. The first one is caused by the instrumentation routines, the second by the trace buffer. In our approach this can be in some cases reduced by resizing the trace buffer.

In some systems, even if the processors are exclusively allocated to a user, some daemon processes interfere from time to time perturbing the measured times. Our approach only considers the execution time of the application processes, but the cache pollution caused by daemons or other processes can be considered as part of the probe effect. This is the most significant prove effect that remains in our approach, especially if, for convenience, the traces are obtained on a single workstation. Nevertheless, the results presented in Chapter 6 show that the quality of the predictions is still very good.

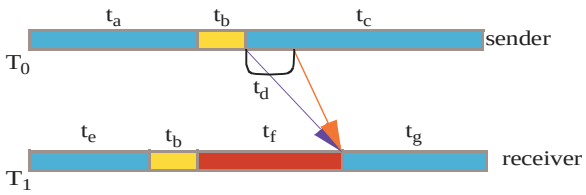
3.5 Dimemas models

In this section, we present how Dimemas models communication and resource management, including communication and processor resources.

3.5.1 Point to point communication

Figure 5 shows how Dimemas rebuilds point to point communication, and the different phases of the communication.

Figure 5. Point to point communication



The blue arrow in Figure 5 symbolizes what we call logical communication, its origin is the point where the sender thread submits the message and its destination is located in the point where the receiver thread collects the message. The red arrow symbolizes the physical communication, its origin and destination point enclose the interval where the resources are used and the message is travelling through the network.

TABLE 1. Timing in point to point communication

Time	Description
T_0	Initial time for sender thread
T_1	Initial time for receiver thread
t_a	Time consumption of the sender thread before arriving to the communication point This information correspond to the first record of the sender thread in the tracefile.
t_b	Communication latency. This time requires processor time, and it is used before any communication resource (link or bus) is acquired.
t_c	Time consumption of the sender after the communication point. This information correspond to the 3rd record for the sender thread.
t_d	Contention due to not having the necessary communication resources, it is necessary to delay the communication as long as this time. This value is provided by Dimemas depending on the availability of resources. When modeling asynchronous communications, this time overlaps t_c.
t_e	Time consumption of the receiver thread before arriving to the communication point This information correspond to the first record for the receiver thread in the tracefile.
t_f	Blocking time of the receiver thread due to the fact that the requested message is not local to the thread's node.
t_g	Time consumption of the receiver after the communication point. This information correspond to the 3rd record for the receiver thread.

3.5.2 Collective operation model

For collective operations, we are not going to map those in point to point communication, because this solution will be very dependent on a specific communication library implementation. We plan to user also a model, a simplification that provides good accuracy related to the cost, applicable to any implementation of collective operations.

The following timing model for collective operations was initially proposed as a first version.

- Synchronization: all collective operations are assumed to synchronize their execution before actually starting communication.
- Resources: Collective operations grab all the communication resources of the system while the communication takes place. The model does not consider that the time of the operations depends on the current number of resources in the system.
- Communication time: once synchronized, the time taken by a collective operation depends linearly on the message size and on the number of processors. Regarding the size, the same latency and bandwidth of the point to point communications are used. The message size used is the largest size involved in the collective operation. Regarding the number of processors, three alternatives were offered: linear, logarithmic or constant.

Once all resources are allocated, each thread consumes communication time based on the three available models:

- Constant

$$T = latency + \frac{size}{bandwidth}$$

- Lineal

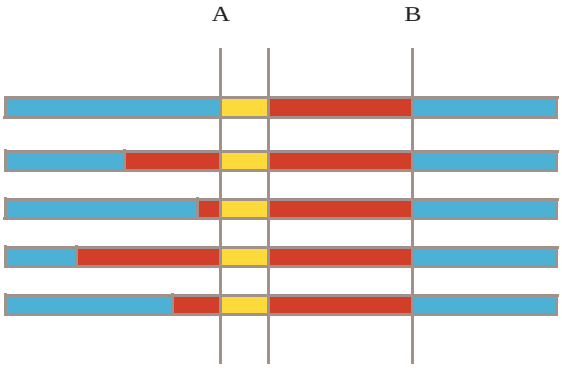
$$T = \left(latency + \frac{size}{bandwidth} \right) \times nprocs$$

- Logarithmic

$$T = \left(latency + \frac{size}{bandwidth} \right) \times \log(nprocs)$$

Figure 6 shows the timing model. At point A processes get synchronized and the communication takes place till point B. Yellow bar stands for time spent for latency. Red bar prior to point A stands for blocking until synchronization.

Figure 6. Collective operation model



The proposal is clearly a very simplified model of reality, a decision made on purpose to match the key philosophical point behind the whole design of Dimemas: simplicity. The reasons for that are:

- We claim that the very simple sets of parameters chosen are sufficient to produce results accurate enough to gain enormous understanding of the application behavior.
- We consider that users tend to be overwhelmed when offered too many tunable parameters. We considered that Dimemas will not be used if it goes to just some tenth of parameters.

3.5.3 Extended collective communication model

Many collective operations have two phases: a first one, where some information is collected (fan in) and a second one, where the result is distributed (fan out). Thus, for each collective operation, communication time can be evaluated as:

$$T = FANin + FANout$$

FAN_IN time is calculated as follows:

$$FANin = \left(latency + \frac{sizein}{bandwidth}\right) \times MODELinFACTOR$$

Depending on the scalability model of the fan in phase, the parameter MODEL_IN_FACTOR can take the following values:

TABLE 2. MODEL_IN_FACTOR possible values

MODEL_IN	MODEL_IN_FACTOR	
0	0	Non existent phase
CTE	1	Constant time phase
LIN	P	Linear time phase, P = number of processors
LOG	*Nsteps*	Logarithmic time phase

In case of a logarithmic model, MODEL_IN_FACTOR is evaluated as the *Nsteps* parameter. Nsteps is evaluated as follows: initially, to model a logarithmic behavior, we will have $\lceil \log_2 P \rceil$ phases. Also, the model wants to take into account network contention. In a tree-structured communication, several communications are per-formed in parallel in each phase. If there are more parallel communications than available buses, several steps will be required in the phase. For example, if in one phase 8 communications are going to take place and only 5 buses are available, we will need $\lceil 8/5 \rceil$ steps. In general we will need $\lceil C/B \rceil$ steps for each phase, being C the number of simultaneous communications in the phase and B the number of available buses. Thus, if $steps_i$ is the number of steps needed in phase *i*, *Nsteps* can be evaluated as follows:

$$Nsteps = \sum_{i=1}^{\lceil \log_2 P \rceil} steps_i$$

For FAN_OUT phases, the same formulas are applied, changing SIZE_IN by SIZE_OUT. SIZE_IN and SIZE_OUT can be:

TABLE 3. Options for SIZE_IN and SIZE_OUT

MAX	Maximum of the message sizes sent/received by root
MIN	Minimum of the message sizes sent/received by root
MEAN	Average of the message sizes sent and received by root
2*MAX	Twice the maximum of the message sizes sent/received by root
S+R	Sum of the size sent and received root

3.6 Processor scheduling

Dimemas was originally designed and developed for analyzing different scheduling policies for distributed memory machines. The idea was to provide the system with a good scheduler, in order not to avoid the delay in the execution of the whole application when sharing the processor time among several applications.

The main problem for processor scheduling in parallel applications is contention in communications due to running tasks waiting for messages to be delivered by non running tasks. In shared environments this problem has several solutions from the point of view of scheduling, and it has been analyzed extensively in the past. One of the best solutions for shared memory machines is Gang scheduling, where all tasks of a given application are scheduled at a time, thus avoiding any blocking due impossibility of freeing resources.

But in distributed memory, those solutions are not applicable in most cases, because communication delays penalize centralized or global scheduling systems. Different scheduling algorithms have been included in Dimemas for evaluation, see Table 4.

For all the scheduling policies in Table 4, the scheduler modifies the priorities (when available), trying to find an automatic mechanism to increase the performance of the system. There is no mechanism that allows the application to modify its priority, although it is quite easy to modify the tracing of the application to include such information. One of the objectives of the work is to evaluate and provide automatic scheduling policies applicable to multiprocessors systems with distributed memory, but having exclusively the information of application behavior included in the tracefile: processor time consumption and communication pattern.

TABLE 4. Scheduling policies

Name	Description
FCFS (First Come, First Serve)	This is the default scheduling policy. A processor is assigned to a thread until the thread frees it due to blocking for a message non local to the node, or blocking for a resource different than the processor.
Round Robin	Processor sharing is based in a fixed time slice. A thread obtains a time slice for execution, but if it blocks for any reason, a new time slice is assigned to a new thread.
FCFS with priorities	Identical to FCFS, but fixed priorities are assigned. Once the scheduler selects a new thread for running, the one with higher priority is selected. The preemption is also supported as an option in the definition. If the preemption is active, it automatically stop the execution of the running thread if there is one thread with higher priority ready to run.
Round Robin with priorities	Identical to Round Robin, but using fixed priorities. Preemption is also supported.
Unix like (SVR4)	This is the time sharing scheduling policy implemented in most Unix system. The thread assigned to the processor is the one with higher priority, and the scheduler assigns a time slice depending on its priority. Priorities are modified dynamically, depending on the utilization of the different resources.
BOOST, FCFS and priorities	Identical to FCFS with priorities, but instead of having fixed priorities, the system modifies the priorities according to the behavior of the application related to message reception.

TABLE 4. Scheduling policies

Name	Description
BOOST, Round Robin and priorities	Identical to Round Robin with priorities, but instead of having fixed priorities, the system modifies the priorities according to the behavior of the application related to message reception.
Critical Path	This is based in FCFS with priorities, where each application tracefile incorporates in the tracefile some events with the indication of the priority of given blocks of the application. This information is based in a static analysis of the critical path of the application. The highest priority is requested when the application is located in its critical path.

For example, BOOST policy tries to automatically detect the application critical path. If the system is able to locate this critical path, it can assign higher priorities to threads involved in critical path, to avoid delays in the messages in the critical path. Lower priorities are assigned to other threads of the application.

There is a set of characteristics applicable to most of the different scheduling policies. Here follows a list:

- Context switch cost

  Time due to context switching. Time consumed when a new thread is scheduled to a new processor. From the point of view of the model, it represents cache invalidation.

- Busy wait time

  In order to improve the performance of parallel applications in shared memory systems, and to avoid some delays in synchronization, a thread performs a busy wait for data prior blocking. For well balanced applications, this mechanism reduces, in many cases, the total number of context switches (and its associated cost).

  Dimemas incorporates the same mechanism but applied to messages. When a thread blocks for message reception, it performs a busy wait for this message before really blocking.

- Time slice

  Time assigned to a thread for executing exclusively in a given processor.

- Minimum time before preemption

  This value is included to prevent context switch before a minimum time is consumed. It is only applicable for preemptive scheduling policies. The objective is to spent some time within the same thread and reusing the cache, before a context switch is allowed.

- System V Release 4, priority and time slice table

  Each priority has an associated entry in a table. The columns in this table fix the different time slices and new priorities of the threads. The philosophy of these different values is to provide larger time slices for those threads doing no communication (I/O) and shorter ones but with higher priority to those doing I/O.

TABLE 5. System V Release 4, priority and time slice table

Entry	Description
ts_globpri	global priority
ts_quantum	time slice given to threads at this level
ts_tqexp	new priority assigned when thread at this level exceeds its time slice. Worst priority than current one
ts_slpret	new priority assigned when thread at this level returns to user mode after sleeping (blocking). Best priority than current one

3.7 Additional Dimemas functionalities

There are some additional functionallities included in Dimemas to provide more analysis capabilities for parallel applications. In this group, we include critical path computation, sensitivity analysis and per module analysis.

3.7.1 Critical Path

Given a tracefile, containing the information of a given execution for an application, it is important to detect the longest communication path. With this information, the user can try to modify the application behavior, trying to reduce the amount of time spent in the critical path.

Figure 7. Critical path description

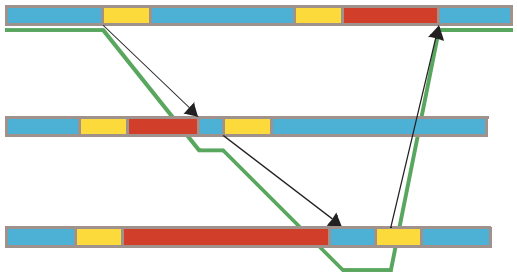


Figure 7 contains an example of critical path computation, and it is shown in green under each thread. It is computed from end to beginning, once the complete simulation has finished. Dimemas takes the thread that finish later, in this cases the first one. Then goes backward until it finds a block for a message, and then the critical path is translated to the sender thread. This is up to beginning of tracefile. By definition, the critical path never contains a blocking due to point to point communication. Different colors in Figure 7 have the same definition as in Figure 5.

3.7.2 Sensitivity Analysis

For some specific applications it is important to know how sensible is the application execution based in different parameters, including subroutine duration and architectural parameters. Dimemas incorporates a module for performing 2-factorial design, analyzing the influence of variation of the selected parameters within a factor of 10%.

This method is based in the execution of several simulations with different values for the parameters, and then fitting a curve for the predicted application time. For the selected parameters we present the percentage of influence and the coefficient for the computed curve or surface.

With this option, the user can analyze the influence of some not really well known values of the architectural parameters. For example, in those case that for the communication bandwidth is necessary to use the vendor information. It is a good idea to perform sensitivity analysis on bandwidth, as the information provided by vendors differs from the real one (in some cases completely).

3.7.3 Per module analysis

Application developer may have interest on how application will behave if a given subroutine or a block in the code is executed twice faster or slower. This interest is based in subroutine mapping to fastest processors and effort dedicated to code optimization.

This facility is implemented simply modifying the cpu time request for the application when located inside specific blocks. Although this is not exactly how it will behave, results provides a good quantitative analysis to take decisions.

Another valid analysis for this functionality is the study of application execution in vector machines, where some subroutines or modules perform really better because they take advantage of the vector unit.

3.7.4 Quasi random parameters

One of the most important problems for simulation is the difficulty for validation. Although the model is simple and uses few parameters, it is important to remark that reality is quite complex, and in most case a single parameter can not model all the behavior of real world.

Dimemas allows the definition of statistical functions for each of the different user tunable parameters. In every situation Dimemas needs, for example, the bandwidth value, it will be computed as a density function (uniform, gaussian,...). This functionality permit us to perform simulations without considering an ideal system, also taking into account some applications not being analyzed by Dimemas but using different resources (network, processor,...). These applications modify dynamically the behavior of the system, but Dimemas can use this statistical function to model the influence of those applications.

CHAPTER 4

Paraver

Paraver is a flexible parallel program visualization and analysis tool based on an easy-to-use Motif GUI. Paraver was developed responding to the basic need of having a qualitative global perception of the application behavior by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Paraver provides a large amount of information on the behavior of an application. This information directly improves the decisions on whether and where to invert the programming effort to optimize an application. The result is a reduction of the development time as well as the minimization of the hardware resources required for it.

Some Paraver features are the support for:

- Detailed quantitative analysis of program performance
- Concurrent comparative analysis of multiple traces
- Fast analysis of very large traces
- Mixed support for message passing and shared memory (networks of SMPs)
- Easy personalization of the semantics of the visualized information

One of the main features of Paraver is the flexibility to represent traces coming from different environments. Traces are composed of state transitions, events and communications with an associated timestamp. These three elements can be used to build traces that capture the behavior along time of very different kinds of systems.

This chapter includes a detailed description of the Paraver programming model, trace format and internal structure. The possibilities offered by the visualization, semantic and quantitative analyzer modules are powerful enough allowing users to analyze and understand the behavior of the traced system. Paraver also allows customizing of some of its parts as well as the plugging of new functionalities.

So expressive power, flexibility and the capability of efficiently handling large traces are key features addressed in the design of Paraver. The clear and modular

structure of Paraver plays a significant role towards achieving these targets. Let us briefly describe the key design philosophy behind these points.

Views

Paraver offers a minimal set of views on a trace. The philosophy behind the design was that different types of views should be supported if they provide qualitatively different analysis types of information. Frequently, visualization tools tend to offer many different views of the parallel program behavior. Nevertheless, it is often the case that only a few of them are actually used by users. The other views are too complex, too specific or not adapted to the user needs.

Paraver differentiates three types of views:

- Graphic view: to represent the behavior of the application along time in a way that easily conveys to the user a general understanding of the application behavior. It should also support detailed analysis by the user such as pattern identification, causality relationships,...
- Textual view: to provide the utmost detail of the information displayed.
- Analysis view: to provide quantitative data.

The Graphic View is flexible enough to visually represent a large amount of information and to be the reference for the quantitative analysis. The Paraver view consists of a time diagram with one line for each represented object. The types of objects displayed by Paraver are closely related to the parallel programming model concepts and to the execution resources (node and processors). In the first group, the objects considered are: application (Ptask in Paraver terminology), task and thread. Although Paraver is normally used to visualize a single application, it can display the concurrent execution of several applications, each of them consisting of several tasks with multiple threads.

The information in the graphics view consists of three elements: a time dependent value for each represented object, flags that correspond to punctual events within a represented object, and communication lines that relate the displayed objects. The visualization control module determines how each of these elements is displayed. The essential choices are:

- Time dependent value: displayed as a function of time or encoded in color. Furthermore, time and magnitude scale can be changed to zoom the visualization.
- Flags: displayed or not and color.
- Communication: displayed or not.

The visualization module blindly represents the values and events passed to it, without assigning to them any pre-conceived semantics. This plays a key role in the flexibility of the tool. The semantics of the displayed information (activity of a thread, cache misses, sequence of functions called,...) lies in the mind of the user. Paraver specifies a trace format but no actual semantics for the encoded values. What it offers is a set of building blocks (semantic module) to process the trace before the visualization process. Depending on how you generate the trace and combine the building blocks, you can get a huge number of different semantic magnitudes.

Expressive power

The separation between the visualization module which controls how to display data and the semantic module which determines the value visualized offers a flexibility and expressive above than frequently encountered in other tools.

Paraver semantic module is structured as a hierarchy of functions which are composed to compute the value passed to the visualization module. Each level of function corresponds to the hierarchical structure of the process model on which Paraver relies. For example: when displaying threads, a thread function computes from the records that describe the thread activity, the value to be passed for visualization; when displaying tasks, the thread function is applied to all the threads of the task and a task function is used to reduce those values to the one which represents the task; when displaying processors, a processor function is applied to all the threads that correspond to tasks allocated to that processor.

Many visualization tools include a filtering module to reduce the amount of displayed information. In Paraver, the filtering module is in front of the semantic one. The result is added flexibility in the generation of the value returned by the semantic module.

Combining very simple semantic functions (sum, sign, trace value as is,....), at each level, a tremendous expressive power results. Besides the typical processor time diagram, it is for example possible to display:

- The global parallelism profile of the application.
- The total per CPU consumption when several tasks share a node.
- Average ready queue length of ready tasks when several tasks share a node.
- The instantaneous communication load geometrically averaged over a given time.
- The evolution of the value of a selected variable,...

The default filtering and semantic function setting of the tool results in the same type of functionality of many other visualization tools. A much higher semantic potential can be obtained with limited training.

Quantitative analysis

Global qualitative display of application behavior is not sufficient to draw conclusions on where the problems are or how to improve the code. Detailed numbers are needed to sustain what otherwise are subjective impressions.

The quantitative analysis module of Paraver offers the possibility to obtain information on the user selected section of the visualized application and includes issues such as being able to measure times, count events, compute the average value of the displayed magnitude,...

The quantitative analysis functions are also applied after the semantic module in the same way as the visualization module. Again here, very simple functions (average, count,....) at this level combined with the power of the semantic module result in a large variety of supported measures.

Some examples are:

- Precise time between two events (even if they are very distant)
- Number of messages sent between time X and Y
- Number of messages of tag T exchanged between processor P1 and P2 between time X and Y
- Average CPU utilization between time X and Y
- Number of events of type V on processor W between time X and Y
- Average CPU time between two communications

Multiple traces

In order to support comparative analyses, simultaneous visualization of several traces is needed. Paraver can concurrently display multiple traces, making possible to use the same scales and synchronized animation in several windows.

This multi-trace feature supports detailed comparisons between:

- Two versions of a code
- Behavior on two machines
- Difference between two runs
- Influence of problem size

Comparisons which otherwise are very subjective or cumbersome.

Customizing

Developing a tool which fulfills the needs of every user is rather impossible. Initially Paraver aimed at supporting the projects carried out at CEPBA as part of our research and service activities.

One objective of the design was to provide some support for expert users having new needs and willing to extend the functionalities of Paraver. For this purpose, Paraver is distributed with the possibility of personalizing the two modules that provide the expressive and analysis power.

A procedural interface is provided in such a way that a user can, with a limited effort, link into its Paraver version the actual functionality needed. Taking into account the built in semantic functions and analysis functions and their relation to the hierarchical process model and the possibility of combining them in a totally orthogonal way at each level, a user can obtain a large number of new semantics by developing very simple modules.

Another aspect where the users may have personal preferences is in setting color tables. More important is the possibility to specify the textual values associated with the values encoded in the trace. All this is achieved through a simple but powerful configuration file.

Configuration files are simple, but very useful files which lets to the user customize his/her environment, save/restore a session,... Also, we can change some default options and redefine the environment.

Large traces

A requirement for Paraver was that the whole operation of the tool has to be very fast in order to make it usable and capable to maintain the developer interest. Handling traces in the range of tenths to hundreds of MB is an important objective of Paraver.

Easy window dimensioning, forward and backward animation, zooming are supported. Several windows with different scales can be displayed simultaneously supporting global qualitative references and detailed analysis. Even on very large traces, the quantitative analysis can be carried out with great precision because the starting and end points of the analysis can be selected on different windows.

This chapter is organized as follows. First, the trace file format will be described in Section 4.1. In Section 4.2, we will focus in the internal structure of Paraver, describing the different modules (filter, semantic and representation) and their interaction. Section 4.3 describes some characteristics of the semantic module. The representation module is explained in detail in Section 4.4.

4.1 Paraver trace file format

The first step in the visualization procedure is to get a visualization trace file. The visualization trace file contains records which describe absolute times at/during which events/activities take place on a run of the parallel code. Each record represents an event or activity associated to one thread in the system. Three basic types of records are defined in Paraver:

5. State records

   represent intervals of actual thread status or resource consumption.

6. Event records

   represent punctual user defined "events" in the code. These user events are often used to mark the entry and exit of routines or code blocks. They can also be used to flag changes in variable values. Event records include a type and a value.

7. Communication records

   represent the logical and physical communication between the sender and the receiver in a single point to point communication. Logical communications correspond to the send/receive primitive invocations by the user program. Physical communication corresponds to the actual message transfer on the communication network.

An important issue in the tracing facility implementation must be precision and accuracy in the measurement. The tracing libraries works with microsecond precision, so all the absolute times must be in a microsecond precision.

Paraver accepts two trace format types: ascii and binary. Only the ASCII trace format is described in this section. The binary format is an internal Paraver trace file format supports the same functionalities and we will not include the description as it will not introduce new topics.

The ASCII trace basically contains a header and a list of records. The header describes the system and application characteristics in terms of number of nodes, number of processors, number of tasks and threads and their mapping to the nodes and processors. The trace records represent state of a thread, events on a thread or communication between a couple of threads. Since a single trace file may contain several applications, a Ptask field is needed on each record besides task and thread fields to state which object is applied by the record.

An ASCII trace is composed by a header and a body. The header contains information about the object structure and the body contains all the traces (or records) which the trace file is composed.

4.1.1 Paraver trace header

The trace file must have a header with a specific format, where there is information about the trace file. This header defines completely the different objects in the trace body.

The header must have the following format:

```
#Paraver (dd/mm/yy at hh:mm):final time:number CPU:number PTASK:ptask
list[:ptask list]
```

- final time

  Total application time in microseconds

- number CPU

  Number of processors in the trace file

- number PTASK

  Number of applications in the trace file

- ptask list

  Number TASK(mapped task [,mapped task])

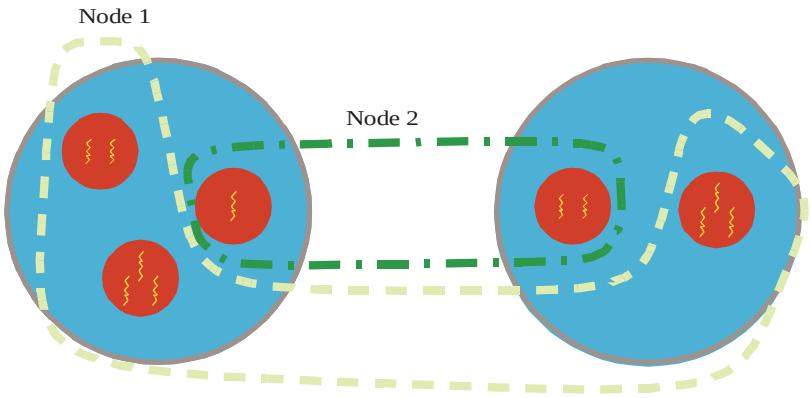
Each application (Ptask) has its own "Ptask list".

- mapped task: number THREADs:allocated node

  Each task has its own threads; It must be indicated the number of threads of each one and the node where is mapped. Its position in the list will indicate what task is.

Here follows an example of a trace header. We will use the object structure shown in Figure 8.

Figure 8. Example of Paraver header



We will use an object structure where there are two Ptask which are composed by some tasks. Also, we have two nodes where all the objects will be mapped. On the NODE 1 we would map two tasks of PTASK 1 (we would map TASK 1 and TASK 2) and the TASK 1 of PTASK 2. The other objects will be mapped onto the NODE 2.

The header that contains the mapped structure is:

```
#Paraver (dd/mm/yy at hh:mm):final_time:2:2:3(4:1,1:1,3:2):2(3:1,2:2)
```

where ...:final time:2:2:... says that our structure has 2 nodes and 2 Ptasks; where each Ptask is mapped like:

- Ptask 1 has 3(4:1,1:1,3:2)

  Indicates that it has 3 tasks where the first task has 4 threads mapped on the node 1, the second task has 1 thread mapped on the node 1 and the third task has three threads mapped on the node 2.
- Ptask 2 has 2(3:1,2:2)

  Indicates that it has 2 tasks where the first task has 3 threads mapped on the processor 1 and the second task has 2 threads mapped on the node 2.

4.1.2 Paraver trace body

Paraver has three types of records: states, user events and communications.

State record

```
type:object:begin_time:end_time:state
```

- type

  The constant 1. Each type record is identified with this field.
- object

Describes the identifiers of the object to apply this record. It includes the processor identifier (a unique sequential number generated all over the different nodes), the Ptask identifier, the task identifier and the thread identifier.

- begin time

  Starting absolute time of the state burst

- end time

  finishing absolute time of the state burst

- state

  integer greater than zero. The default configuration of Paraver is:

TABLE 6. Default configuration values for Paraver record

Value	Description
0	Idle, the object is not working
1	Running, the object is computing
2	Stopped, the execution is finished. There are no more records of this object.
3	Waiting for message.
4	Waiting for link (network resource)
5	Waiting for CPU. The corresponding object is ready for execution, but there are no free processors in the current node to start the execution.
6	Waiting for semaphore. The object has performed a wait operation on a semaphore, and the corresponding signal is pending.
7	Latency. The object is consuming the overhead time for sending a message.
8	Probe, the object is checking if there is a message ready for reception. (If the trace file comes from Dimemas, this record indicates processor context switch cost)
9	Send Overhead, the object is working i.e. packing message (Using the Paraver instrumentation library. This record is not available for Dimemas generated files)
10	Send Overhead, the object is working i.e. packing message (Using the Paraver instrumentation library. This record is not available for Dimemas generated files)
11	Disk I/O, the object is making and I/O operation (For Dimemas generated files, this status corresponds to the physical Disk I/O)
12	Busy Wait, the object is in a busy wait
13	Collective OP Synch., the object is making a group synchronization
14	Collective OP Comm., the object is making a group communication
15	Disk I/O block, the object is block due I/O contention (Only available for Dimemas generated files)

Obviously, the user can interpret these ones like he/she wanted.

Event record

```
type:object:current_time:event_type:event_value
```

- type

  The constant 2. Each type record is identified with this field.

- object

  Describes the identifiers of the object to apply this record. It includes the processor identifier (a unique sequential number generated all over the different nodes), the Ptask identifier, the task identifier and the thread identifier.

- current time

  current absolute time of the event occurrence.

- event type

  identifier of the event. The user interpret this value like he wanted, although there are some values reserved, corresponding to the following table.

- event value

  value of the event, to distinguish events of the same type.

TABLE 7. Paraver event types

Event type		Event value	
Type	**Description**	**Value**	**Description**
40	Block / subroutine	>0	Enter block number
		=0	Exit current block
99	Semaphore signal	>0	Semaphore number
98	Semaphore wait	>0	Semaphore number
90	Critical Path	0	Exit
		1	Enter

Communication record

```
type:object_send:time_send:object_rec:time_rec:size:tag
```

- type

  The constant 3. Each type record is identified with this field.

- object send

  Describes the identifiers of the object sending the message. It includes the processor identifier (a unique sequential number generated all over the different nodes), the Ptask identifier, the task identifier and the thread identifier.

- time send
  - logical send

absolute time indicates when the application wants to send a message.

- physical send

  absolute time indicates when the message is really sent.

- object receive

  Describes the identifiers of the object receiving the message. It includes the processor identifier (a unique sequential number generated all over the different nodes), the Ptask identifier, the task identifier and the thread identifier.

- time receive
  - logical receive

    absolute time indicates when the application wants to receive a message.

  - physical receive

    absolute time indicates when the message is really received.

- size

  integer greater than zero. It represents the size in bytes of the message.

- tag

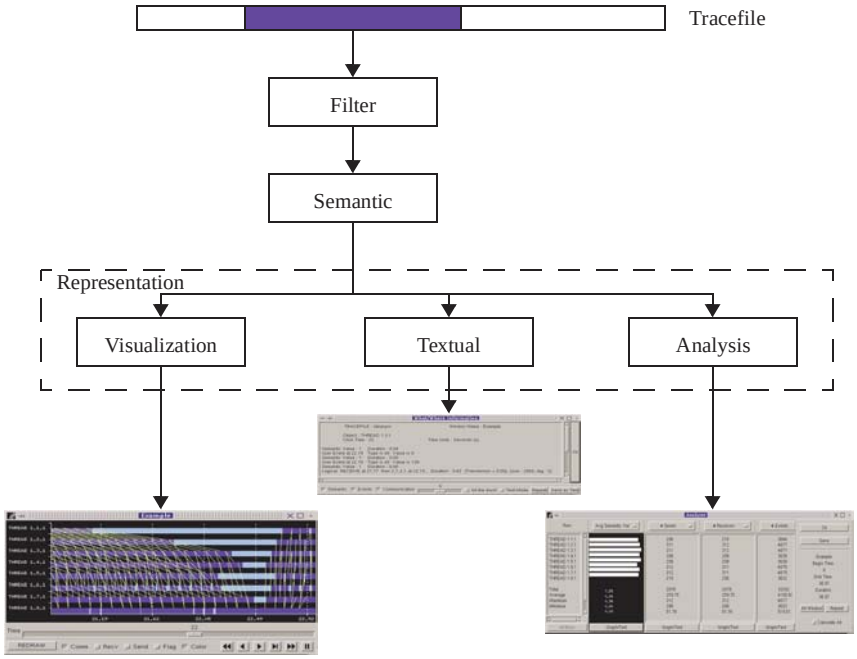
  integer greater than zero. It is the label of the message.

4.2 Paraver internal structure

Paraver structure consists of three levels of modules represented in Figure 9. First, working onto the trace file there is the Filter Module. This module gives to the next level a partial view of the trace file. Second, there is the Semantic Module which receives the trace file filtered by the previous module and interprets it. The Semantic Module transforms the record traces to time dependent values which will be passed to the Representation Module. The Semantic Module is the most important level because it extracts and gives sense to the record values in the trace file. The trace file has a lot of information that could be extracted and this module selects what will be extracted, this information is called, the semantics of the trace file.

Finally, there is the Representation Module. It receives the time dependent values computed by the semantic module and displays it in different ways. The Representation Module drives thus the whole process and offers a graphical display of the trace file.

Figure 9. Paraver internal structure



4.2.1 Representation Module

The representation module is composed of three modules, the Visualization, the Textual and the Analyzer modules. These modules are the actual driving engine of the system. The mode of operation can be seen as demand driven. When a new value has to be displayed (for example, going to a time stepping through the animation or computing an statistic) the sequence of values or events needed is requested from the next module.

- Visualization module

  The Visualization module is responsible of values and events displayed on the screen and of the pointing device handling. It is aware of concepts such as window sizes, display scales (time and magnitude), type of display (color or level), type of object (processors, application, tasks of a given application, threads of a given task, user events), number of rows to display, measurement of time (even between different windows for precise measurements of long intervals) and so on. This module concerns with the creation of new windows either by a direct user command or through the zooming and cloning facilities. It takes care of the trace animation which can be displayed both forwards and backwards. It also offers the possibility to go to a position into the trace file to be displayed by specifying: an absolute or relative time, an user event type or value, a message tag or size.

- Analyzer module

The Static Analysis module computes statistics on overall or even just a user selected part of the trace file. There are some predefined statistics such as the average semantic value, number of events, number of communications (sends, receives),...; a frequent problem encountered by tool users is that an specific analysis type is not directly included in the tool. To overcome this problem, a procedural interface is provided to develop and link user specific functions to Paraver. The output of the analysis functions is presented in a Paraver window either as table or as histogram.

4.2.2 Semantic Module

The Semantic module computes the values to be transferred to the representation module. The computation of the values is based on one or several records as returned by the filter module described below. The representation module is the only one that looks for the actual coding of the trace state records and for the process model semantics. As mentioned before, each window represents objects of a given type: processor, application, task or thread. The semantic module uses, to generate values that will be given to the visualization or analysis module, the composition of one function for each object level submitted to the one selected for the window. Examples of functions at: thread level can return:

- active, inactive (0,1) depending on the actual trace state (useful computation, overhead, waiting, transferring data,...)
- state code
- last user event type, appeared in the trace
- last user event value, appeared in the trace

task level can apply the sum or the boolean function to the values returned by the thread level function for all threads within each task. **application** level can apply the sum or the boolean function to the values returned by its task members.

Each of these elemental functions are really simple but an important part of the power of Paraver lies in this module and how the user combines the different functions to generate the type of value that is relevant for his analysis. In case the system provided insufficient functions, the user can link to Paraver its own function.

4.2.3 Filter Module

The Filter module offers to the previous ones a partial view of the trace file. The user can specify that only certain events are passed to the Semantic module, like only a specific type of user event, messages of a given tag or destination, and so on. This is specified by a Filter module control window that offers a nice interface for a very flexible set of filtering options. State records are always passed to the Semantic module without filtering.

Filtering any communication or any event means that those filtered traces won't appear in the next modules and also, they won't appear in our visualization or in our analysis. These modules are very useful when we are working on the next levels with the communication and event traces because it lets to focus our study onto a specific group.

For communication is it possible to filter message from a given source, to some destination, with a given size and given tag. All these possible selections allows semantic module and representation modules, to handle and display only the necessary records in the tracefile. For events, the selection includes event type and event value. This permits the analysis of a subset of events, for example, a subset of subroutines, or a subset of processor counters.

4.3 Semantic module

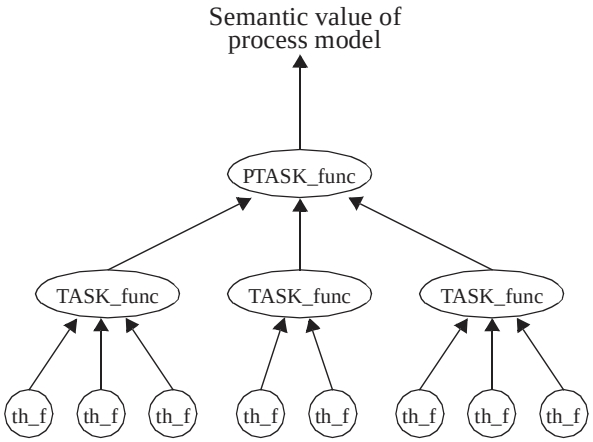
The Semantic Module computes the values that will be transferred to the representation levels; these values are based on one or several records returned by the Filter Module and are known as semantic values. The representation levels take those values and works in different ways. For example, those values could be seen on a displaying window like a code colors where each value has associated a color, or can be analyzed in the Analyzer Module.

Note, that Semantic Module could extract the information from the trace file in several ways, and we will have to select it. Also, when we are working in the semantic module, we have to use the Paraver Process Model, because the semantic module will work over a selected level.

4.3.1 How does the semantic module work?

Paraver offers some default ways to collect the values from the previous levels. The Paraver Process Model has four levels where we can work: the application level (PTASK), task level, thread level and processor level (CPU). In each level Paraver applies some functions to collect the values from the previous levels; and finally, applies the two levels called compose levels. These two levels have been added to modify the semantic value at the top level.

Figure 10. Semantic value computed at Ptask level



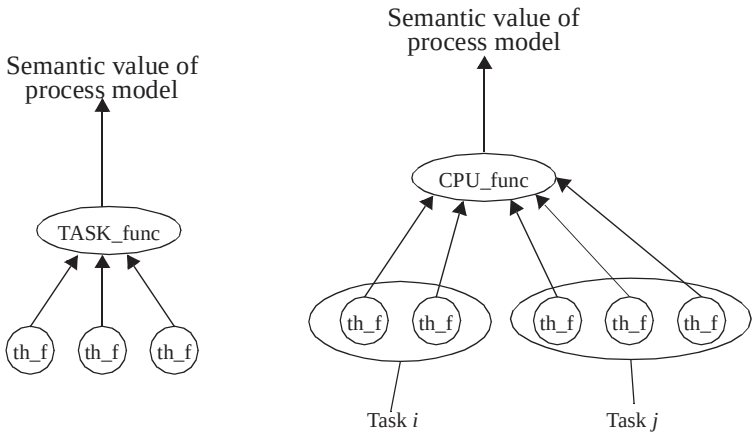
If we are working at PTASK level, the semantic value returned through the process model is a combined value of all the values computed for all the objects within the

ptask object where: each thread within a task computes a value, the tasks computes a combined value with all the values returned by its threads, and finally, each task returns the combined value to its ptask which compute a combined value with all the tasks values (Figure 10).

When we are working with levels TASK or CPU levels the semantic value is computed as:

- when working at TASK level, it returns a combined value of all the values returned by its threads (Figure 11 left.).
- when working at CPU level, it returns a combined value of all the threads which are executing on that CPU (Figure 11 right.). Remember, that a cpu has mapped all the threads of a task, and can have mapped more than one task of the same or different Ptasks.

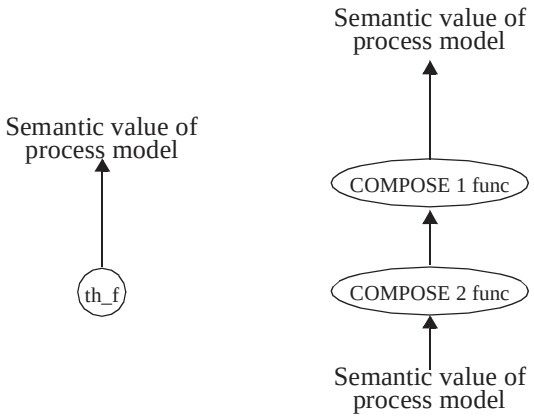
Figure 11. Semantic value computed at Task (left) and CPU levels (right)



Working at THREAD level is the simplest way, because each thread returns the result of apply the selected function (figure 9.3.a.).

Finally, when working with any of the four mentioned levels, the two compose levels are always applied. These two composed levels allow applying a function to the semantic value returned by the Process Model computation (figure 9.3.b.).

Figure 12. Semantic value computed at thread (left) and compose levels (right)



Note, that the thread level always is the lowest level and always appears when we are working with an object level. This level works onto the trace file (which has been filtered by the Filter Module) and select in what we are going to work: states values, events or communications. The functions applied in the other levels only transform the value(s) passed from the previous level.

4.3.2 Default semantic functions

By default, Paraver has some semantic functions in each level that could be used in different ways to extract the information that will be analyzed. All these functions return a value that will be passed to the next level, sometimes this value is a combined value of a group of inputs values. The level where we are working tell us which function will be used, and selecting the appropriate functions we could obtain our goals. The user can write his/her own functions to extract his/her specific information (the interface is described in the User Manual).

4.3.2.1 Thread functions

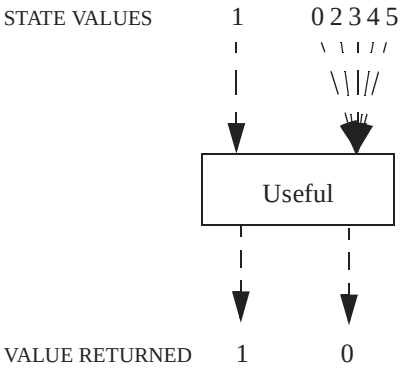
This is the lowest, but the most important level. This level decide in what we are going to work such as states, events or communications and what we are going to do with them.

Thread functions work over the trace file (it has been filtered by the Filter Module). This level is always enabled. These functions are grouped, the first group (State group) has functions that work with state traces; usually, these functions only work with the state value of the state trace. The second one is composed by functions that work with event traces; some functions work with event types and some with event values. Finally, the third group is composed by functions that work with communication traces. The next points will explain how works each function and what value will return.

1. Functions that work with state traces

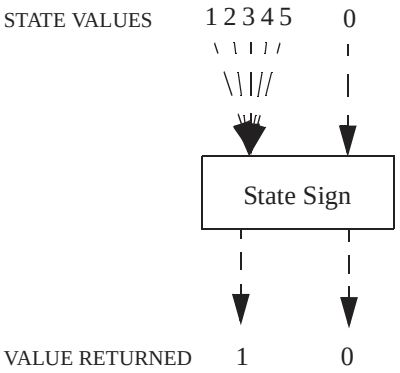
- **Useful**: It takes the state trace and returns if the state value is Running (state value 1) or not (if the state trace isn't Running it returns an idle state or value 0). Only state values 1 are returned, the rest of state values are returned as an idle state (state value 0). See Figure 13.

Figure 13. Useful function



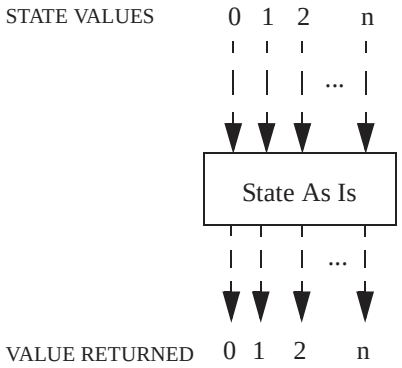
- **State Sign**: It takes the state trace and returns a state value 1 (running state) when state value isn't an idle state (state value 0) and if state value is 0 (idle state) a 0 is returned.

Figure 14. State Sign function



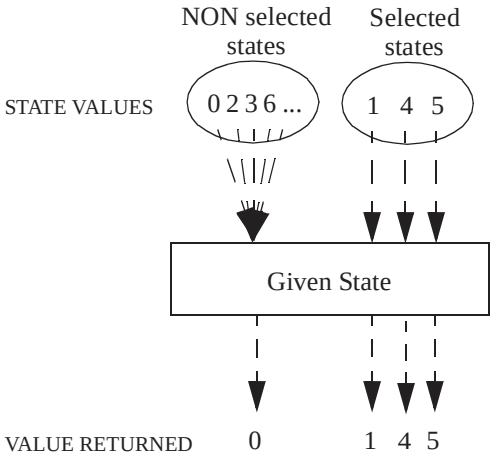
- **State As Is**: It takes the state trace and returns its state value. This function does not modify the state value, it only returns the state value.

Figure 15. State As Is function



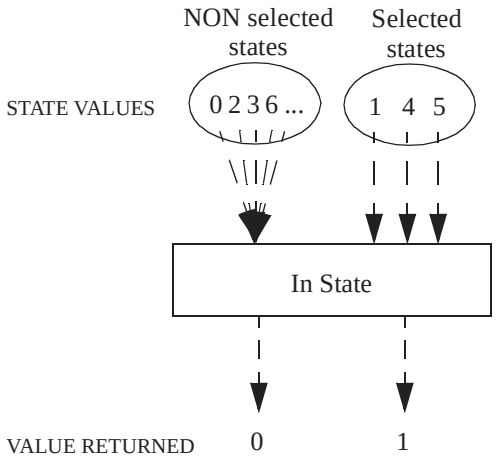
- **Given State**: It takes the state trace and returns its state value if it is selected or idle if not. Is like the State As Is function but only works with selected states, these states are returned, but the states which hasn't been selected will be converted to an idle state (state value 0). The user can select more than one state.

Figure 16. Given State function



- **In State**: It takes the state trace and returns if the state trace is in one of the states selected or not. Is like the State Sign function but only returns as a running states the selected state values, those selected values are converted to the a running state, the rest are returned as an idle state. The user can select more than one state.

Figure 17. In State function



- **Not In State**: It takes the state trace and returns if the state trace isn't in one of the sates selected or not. The user can select more than one state.

2. Functions that work with state traces
   - **Last Event Value**: This function takes the event trace and returns its event value. So the value from this event trace to the next event trace will be the current event value.
   - **Last Event Type**: It takes the event trace and returns its event type. Is like the Last Evt Val function but it returns the event type instead of event value.
   - **Next Event Value**: When an event trace is encountered this function searches for the next event and returns its value. So from the current event to the next event, the value returned will be the value of the next event.

     This function will be very useful when for example, the user events are counting something that had happened within the interval between the current time and the next event time, because it lets to paint the interval with the counting value.
   - **Next Event Type**: When an event trace is encountered this function searches for the next event and returns its type. Is like the Next Evt Val function but it returns the event type instead of event value.
   - **Average Next Event Value**: This function works like the Next Evt Val (working with the value of the next event) but returning a value in function of the interval duration

$$\frac{value_{i+1} \cdot factor}{t_{i+1} - t_i}$$

where:

- $value_{i+1}$ is the value of the next event
- t_{i+1} is the time of the next event (in microseconds)

- *ti* is the time of the current event

The value of the next event (value i+1) is multiplied by a factor (by default, value 1000). This factor could be used to change the units of the returned value, for example, can be used to obtain the value per seconds instead of value per milliseconds. By default, we obtain a value per milliseconds because factor is 1000 and we are working with microsecond precision.

- **Average Last Event Value**: This function works like the previous one but works with the current event value instead of the next.
- **Given Event Value**: It takes the event trace and returns its event value if it has been selected or an idle state if not. To select the event values that will be returned, when this function is selected Paraver raises a window to select these event values. As the state parameters windows, more than one value could be selected.
- **In Event Value**: It takes the event trace and returns a running state if it has been selected or an idle state if not. More than one value could be selected.

  The difference between this function and Given Evt Val is the same that the difference between Given State and In State; the previous one returns the event value, and this one returns it as a running state (value 1).
- **Int. Between Evt**: This function returns as a value the time between the current event trace and the next event trace. When an event trace is encountered, search for the next event and substracts the two times.

3. Functions that work with communication traces
   - **Last Size**: If the communication trace is a physical receive returns its message size.
   - **Last Tag**: If the communication trace is a physical receive returns its message tag.

4.3.2.2 CPU functions

When CPU functions are enabled, we are working at the CPU level where CPU and THREAD levels are enabled. The CPU functions receive the values returned from the previous level, THREAD level, and compute a combined value with these values which will be passed to next level, COMPOSE level.

The functions implemented by paraver at this level are:

- **Adding**: It adds all the input values and returns the sum.
- **Adding Sign**: It adds all the input values and returns a running state (value 1) if the sum is greater than zero. Otherwise, return the idle state (value 0).
- **Average**: It returns the average of the input values.
- **Maximum**: It returns the maximum value of all the input values.
- **Minimum**: It returns the minimum value of all the input values.

4.3.2.3 TASK functions

When the TASK functions are enabled it means that we are working at TASK or PTASK levels. The TASK level works over the THREAD level. In this level, each task receives the values from its mapped threads (remember that in CPU

level each cpu receives the values from the threads of all its mapped tasks), takes all these values and computes a combined value which will be passed to the COMPOSE level. The TASK level is enabled when working at TASK level and PTASK level.

The functions implemented by paraver at this level are:

- **Adding**: It adds all the input values and returns the sum.
- **Adding Sign**: It adds all the input values and returns if the sum is greater than zero.
- **Average**: It returns the average of the input values.
- **Maximum**: It returns the maximum value of all the input values.
- **Minimum**: It returns the minimum value of all the input values.
- **Thread i**: This function takes all the input values and returns the value of the thread i. If tasks have different number of threads, it could happen that a selected thread number not exists within a task, in this case, the value 0 is returned for this thread.

4.3.2.4 PTASK functions

The PTASK functions return values which refers to the Application level, these values are computed with all the values from the previous level, in this case this level is the TASK level. As the previous level, these functions compute a combined value which will be passed to the next level (COMPOSE level).

The functions implemented by paraver at this level are:

- **Adding**: It adds all the input values and returns the sum.
- **Adding Sign**: It adds all the input values and returns if the sum is greater than zero.
- **Average**: It returns the average of the input values.
- **Maximum**: It returns the maximum value of all the input values.
- **Minimum**: It returns the minimum value of all the input values.

4.3.2.5 Compose functions

The compose functions are applied at the top level, when all the process model levels have been computed.

We have two compose levels which are applied from bottom to top, first it is applied the COMPOSE 2, level which receives as input the value computed by the process model functions; and finally, is applied the COMPOSE 1 which receives as input the value returned by the previous compose level (see Figure 12 on page 57).

The two compose levels have the same functions and can be combined to obtain the final value that will be returned by the Semantic Module to the Representation Module.

By default, Paraver implements some functions which gives the possibility to change or remain the semantic value when its calculation through the process model has been computed. These functions are:

- **As Is**: Not change the value returned from the previous level. The value remains intact.
- **Sign**: This function returns the sign of the value returned from the previous level. This function return 1 (a running state) if the value is greater than zero and 0 if it is equal.
- **1-Sign**: This function is the complementary of the function Sign. It returns 0 (an idle state) if the value is positive and 1 otherwise.
- **Module+1**: This function returns the module of the value returned from the previous level. By default the divider is the greatest integer number on the machine which does not change the value. The value returned by this function will be within the interval [1...divider] because adds to the result a 1.
- **Module**: This function returns the module of the value returned from the previous level. By default the divider is the greatest integer number on the machine which does not change the value. The value returned by this function will be within the interval [0...(divider-1)].
- **Divide**: This function returns the division of the value returned from the previous level. By default the divider is 1 which not change the value.
- **Select Range**: This function returns the value returned from the previous level if it is in the selected range, if not returns 0. This function only lets to pass the values that are between the interval, the rest are converted to an idle state (value 0). When the user select this function a new window appears to select the range, the user has to select the maximum and minimum value. By default the maximum value is the greatest integer number on the machine and the minimum is 0.
- **Is In Range**: This function works more or less as Select Range but returns 1 if the value returned from the previous level if it is in the selected range, if not returns 0. When the user select this function appears a new window to select the range, the user have to select the maximum and minimum value. By default the maximum value is the greatest integer number on the machine and the minimum is 0.
- **Stacked Value**: This function works as a stack. Stores the values returning in each moment the top value. Values greater than 0 are pushed, when the function receives a value equal to 0 pop up the top of the stack and then, the top is returned. Note that the current (top of the stack) has been returned before the last change. The goal is to remember the previous states.

4.4 Representation module

The representation module is composed by three modules: the visualization module, the textual module and the analyzer module.

Before entering in the representation module, the trace file is pre-processed by the Filter and Semantic modules that actually select what is going to be displayed or analyzed by these modules.

The Visualization module is composed by the displaying windows (see chapter 10) plus some utilities to work with, like Zooming or Timing. This module will give us a graphical visualization of the trace file.

The Textual Module give us a textual representation of the trace files around specific points. It displays in text mode what we are seeing in a displaying window.

Finally, the Analyzer module lets us to get summarized information. Very detailed qualitative analysis can be done by properly selecting the Filter and Semantic modules combined settings.

4.4.1 Visualizer Module

The visualizer module allows the user to display the trace file in a displaying window. Each displaying window shows a particular view of the trace file with its own time interval and scale, and its object representation, even its trace file. The visualizer module has others utilities that work with the displaying window like zooming, timing or the global orders which works with all the opened windows at the same time.

Each displaying window will be associated with a level of visualization. This level fixes the type of object to work with and what will be displayed. There are four types of objects (PTASK, TASK, THREAD, CPU) one per level (Application, Task, Thread, Processor). By default, paraver works with all the objects in the level and all are displayed.

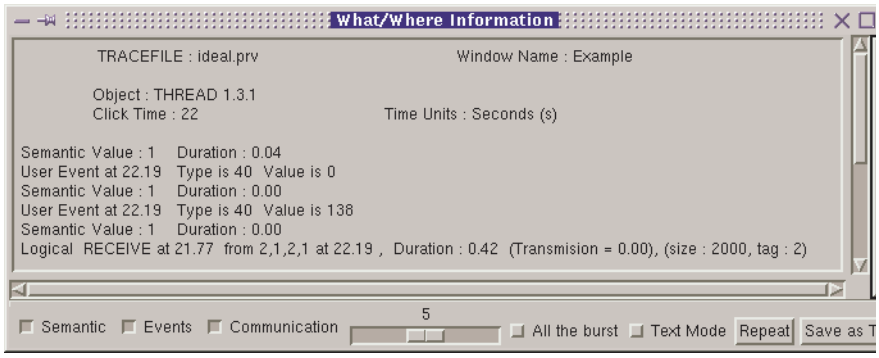
The timing utility computes the elapsed time between two points in the trace file. Both points, the beginning and the end, can be selected from any displaying window. The user must click the selected initial and final points in the drawing area. The timing results will be written in the text box and it shows the initial and final points selected plus the duration of the selected interval. This simple tool is very important to provide quantitative measures and it is implicitly used by several utilities like the zooming or even the static analyzer.

4.4.2 Textual module

Paraver offers a simple and quick way to get information about the tracefile from the displaying window. A click in the drawing area gets a textual display of the tracefile around the selected point.

This textual information is displayed in a window called What/Where window (Figure 18). This window has a Text Area, where the textual information is displayed, and a Control Area where the user can select what he/she wants to see.

Figure 18. Paraver What/Where Information



4.4.3 Analyzer module

This tool let us to analyze a subset or the full trace file. The selected traces are pre-processed by the Filter Module and Semantic Module modules, before entering in the Analyzer Module.

The Analyzer Module provides some predefined functions but also offers an interface to user programmed functions. This feature let to make the most accomplished analysis that the user wants.Paraver lets to make an analysis for all the trace file or a subset.

Following subsections describe the predefined functions.

4.4.3.1 Function Average Semantic Value

This function computes for each selected row the average value along the selected area. Adds the duration of each burst multiplied by their values and divide the result by the duration of the selected area. The Mathematical Formula for each row is:

$$\frac{\sum_{i=1}^{n\_bursts} (t_i \cdot value_i)}{Selected\ time}$$

Where:

- ***n_bursts*** is the number of bursts within the selected area for each row (if the selection cuts a burst, only the selected burst time is used to compute the result).
- t_i is the duration of burst i.
- $value_i$ is the value of burst i.
- ***Selected Time*** is the duration of the selected interval.

4.4.3.2 Function Average Burst

This function computes for each selected row the average duration time for all the selected bursts with a value greater than 0. Adds the duration of each complete burst and divide the result by the number of selected bursts. The Mathematical Formula for each row is:

$$\frac{\sum_{i=1}^{n\_bursts} (t_i)}{n\_bursts} \qquad where \qquad value_i > 0$$

Where ***n_bursts***, t_i and $value_i$ have the same meaning as in the previous section.

By default, all selected bursts will be used to compute the average value, but the user can select the maximum and minimum duration of the burst that will be considered. This feature is useful to filter outliers.

4.4.3.3 Function number of Burst

This function computes number of bursts in the selected area for each row with a value greater than zero. If the selected region cuts a burst with a value greater than zero, that burst will not be added to the result.

As the previous function, by default all the bursts will be used but, the user can select the minimum and the maximum length that will be considered.

4.4.3.4 Function standard deviation Burst

This function computes for each selected row the standard deviation of the duration time for all the selected bursts with a value greater than 0. The Mathematical Formula for each row is:

$$\sqrt{\sum_{i=1}^{n\_bursts} (t_i)^2 - \left[\sum_{i=1}^{n\_bursts} (t_i)\right]^2} \qquad where \qquad value_i > 0$$

Where ***n_bursts***, t_i and $value_i$ have the same meaning as in the previous section.

4.4.3.5 Function Time with Semantic Value

This function computes for each selected row the total duration time for all the selected bursts with a given range of values. The Mathematical Formula for each row is:

$$\sum_{i=1}^{n\_bursts} t_i \qquad where \qquad value_i \in \text{range}$$

Where ***n_bursts***, t_i and $value_i$ have the same meaning as in the previous section, and **range** is the selected range of values. As the previous functions, by default all selected bursts will be used to compute the value. The user can select the maximum and minimum duration of the burst that will be considered.

4.4.3.6 Function Average Message Size

This function computes for each selected row the average message size along the selected area. Adds the message size of each sent message and divide the result by the duration of the selected area. The Mathematical Formula for each row is:

$$\frac{\sum_{i=1}^{n\_messages} size_i}{n\_messages}$$

Where:

- ***n_messages*** s the number of messages in the selected area. Only logical sends are considered to compute the result.
- $size_i$ is the size for message i.

4.4.3.7 Function Number Bytes Sent/Received

This function computes for each selected row the number of send bytes in the selected area. Adds the message size of each sent message. The Mathematical Formula for each row is:

$$\sum_{i=1}^{n\_messages} size_i$$

4.4.3.8 Function Number Sends/Receives

This function computes the number of outgoing communications within the selected area. If logical and physical communications aren't filtered, the computed value will be the addition of logical sends/receives and physical sends/receives.

4.4.3.9 Function number Events

This function computes the number of events in the selected area.

4.4.3.10 Function Max/Min Semantic Val

This function computes the maximum/minimum value within the selected area.

4.4.3.11 User function

Paraver lets to include a user function to code any procedure. The function receives each record trace to analyze it, and returns a value. The result is the adding of all these values. The user routine should have to distinguish the three types of traces.

- State: There are two records in memory for each state trace. The first record means the trace beginning and is pointing to the other which is the trace ending. Both records have the same information, but the fields of begin/end times are inter-changed in the second one.
- Event: There is a record in memory for each event trace.
- Communication: There are four records in memory for each communication trace: the logical send record, the physical send record, the logical receive record and the physical receive record. All of these have the same information about the communication trace.

CHAPTER 5

Promenvir

Promenvir is an advanced Meta-Computing tool for performing stochastic analysis of generic physical systems. Stochastic analysis, whether applied to Structural Mechanics or Computational Fluid Mechanics, is a technique that enables one to take into account the scatter that is normally present in nearly all the data managed by engineers. Scatter normally shows up in various forms, the most typical ones being simple parameter uncertainty, such as density, thickness, or stiffness, or, in more complex cases, local property fluctuations, known under the name of random fields, which may be observed, for example, in composite materials such as concrete or ceramics. These uncertainties in the properties of physical systems, or even in the nature of the loading to which they are subjected (e.g. wind, waves or earthquakes) are reflected in what is known as non-repeatability.

A physical phenomenon is said to be repeatable (essentially deterministic) if each time the results of an experiment with this system are identical. Highly non-repeatable phenomena are, for example, explosions, shocks and impacts, earthquakes, the weather conditions over a certain area, etc. The uncertainties of interest to engineers appear in a probably more hidden form. The local fluctuations of fiber angles in composite material fabric, or the local thickness variations of stamped automobile parts are certainly less evident and less macroscopic manifestations of scatter and uncertainty than the previously cited examples, however, they are increasingly important in modern engineering practice. Today, the strive to build better and faster, and, most importantly, with a higher level of predictability, is pushing engineers to consider scatter as integrating part of their everyday practice.

Promenvir is an industrial tool that enables the engineer and scientist to easily and efficiently define and set up a stochastic problem and to execute it using advanced High Performance Computing resources. Stochastic engineering analyses are known to be CPU-intensive and therefore a new approach in solving these problems is mandatory. One such approach is through Meta-Computing.

This chapter focuses in the meta-computing approach employed for the internals of Promenvir, especially in the resource control and management. It is organized as follows: Section 5.1 describes the method to perform stochastic analysis and intro-

duces some important characteristics of the jobs. Section 5.2 describes the expected functionalities for a queue system. Section 5.3 describes the characteristics of Promenvir workload. Section 5.4 describes the execution environment for Promenvir. Section 5.5 describes the scheduler itself. Section 5.6 describes the specific implementation issues. And Section 5.7 concludes this chapter.

5.1 General description of Promenvir

Modern stochastic analysis is based on the Monte Carlo Simulation technique that requires the execution of series of typically hundreds of deterministic analyses, all of which are clones or replications of a nominal mother model. The idea of Monte Carlo Simulation is basically very simple:

1. Take a reference (nominal) model of the system.
2. Select a set of variables that we know to scatter within a certain range of tolerance (e.g. thickness and density).
3. For each variable pick randomly a value from the above range.
4. Replace the nominal value of each variable with the random one and execute an analysis.
5. Extract and store the value of the response parameters of interest (e.g. frequency).
6. Repeat steps 2 to 5 until the average values of the output parameters stabilize.

With modern Monte Carlo Simulation (MCS) algorithms many normally means around one hundred. This very simple technique of actually simulating many of all the possible combinations of certain parameters is very powerful. It can in fact enable the engineer to easily pinpoint the worst combinations of these parameters, to quantify the reliability of his system and, most importantly, it enables one to gain more insight into the nature of the system under study. MCS reveals the important fact that perfect deterministic models often hide important response mechanisms that can only be triggered if we introduce uncertainty in the model's parameters. This introduction of uncertainty is of paramount importance if one wants to reach highly realistic levels of response with his model.

It is clear from the above description of the basic philosophy of MCS that each deterministic simulation in step 4 is completely independent of all the others. Furthermore, these independent simulations can also be executed in arbitrary order, and even on different computers. Obviously the more computers we can involve in an MCS, the less time shall the analysis take. This important characteristic of MCS is known as intrinsic parallelism. Computing applications in which many separate computers (or computational resources) are used simultaneously to work on the same problem in a coarse grain parallelism context are known as Meta-Applications while the computers themselves constitute a Meta-Computer. It is clear that if we happen to have access to more computers (or CPUs) than the number of single analyses to execute, we can dedicate groups of computers (or CPUs) to each such analysis in a fine grain parallelism context. This will, of course, speedup the MCS even more. If, in exceptionally large and complex Monte Carlo analyses, we need access to remote, even geographically distributed computational resources,

Promenvir will provide all the necessary tools and mechanisms to cope with multi-national Wide Area Networks.

5.2 Queuing systems

The most important features of queuing system are:

- Use of a network of heterogeneous computers as a single system

  The queuing system hides to the user the available computers, and automatically launch jobs in any of the connected machines. For example, LSF unites a group of Unix and NT computers in a single system. Hosts from different vendors can be integrated into a seamless system.

- Time windows for jobs execution

  Time windows are a useful mean to control resource access such that user can disable access to some resources during certain times. In most cases, user workstations will be used for job execution during night, when the machine is available, and then restricting access during work time.

- Resource availability and limits

  Jobs require resources for execution. The queuing systems shall be able to define such resources, the limits of those resources and their current availability. Common resources may include memory, scratch area, swap space,… The queuing system should respect the limits and map the jobs to a host that satisfies the requirements. Current systems include the possibility to react according to the variation of the resources availability: jobs can be suspended; jobs can migrate to another host, …

- User interaction

  Users submit jobs, query for the job status and, in some special situations, suspend and resume the jobs execution. All these functions are supported with a graphical interface. A different graphical interface is provided for the user administrator issues: queue definition, current workload, system parameters, etc.

- Job dependencies

  Some batch jobs depend on the results of other jobs. For example, a series of jobs could process input data, run a simulation, generated images based on the simulation output and finally, record the images on a high-resolution film output device. Each step can only be performed when the previous step completes and all subsequent steps must be aborted if any step fail.

- Remote file access

  If there is no a shared file system space among all the machines in the queuing system, input files should be copied to the execution host before running the job, and the result files copied back to the submission host after the job completes.

- Job description

  Submission jobs are described as a list of command lines, usually included in a shell script file. Most of the queuing systems define a special syntax to specify

the job requirements in the shell script file, instead of fixing those requirements in the submission command line.

- Job scheduling

  Each queuing system offers a set of possible jobs scheduler. The simplest one is Fist-Come-First-Served (FCFS), but it is usually insufficient for an environment with many competing users. Other job scheduling policies group users and assign different priority to different users and hosts. These priorities can also vary among the different queues in the system.

5.3 Monte Carlo workload

The Monte Carlo analysis involves the execution of up to thousand executions of the same simulation, but using different input data files. In this section, an example of Monte Carlo analysis is outlined because the Promenvir queuing systems take advantage of the workload characteristics generated in Monte Carlo analysis.

The goal of the Monte Carlo method is to analyze the behavior of the model when some variation (scatter) is included in the input files, i.e. which is the influence of some input parameters versus some output data. For example, what will be the effect on a car structure in a crash analysis if the initial speed has some variation?

Promenvir, in the problem definition phase, allows the user to select which are the input variables and the output variables. The input variables are the random ones, and the user selects the density function for each. The output variables are those results that the user wants to analyze related to the variation of the input parameters. Once the input and output parameters has been selected, the user selects the total number of simulations to be executed. The number of jobs to be scheduled equals the number of simulations, and usually is below the thousands of simulations.

The Monte Carlo workload characteristics can be summarized as:

- All jobs are identical

  The same input filenames and the same output filenames will be used, and thus there is no chance to execute all simulations in the same directory, because output data will be lost. It is necessary to create a working area for each job. Each job will be assigned to a host and a directory within that host.

- Low variance on job resource demand

  All jobs request the same resources: disk requirements, memory, processor time, swap area and scratch area, because all jobs are going to execute the same analysis. However, the input data for each execution is different, and this can produce different requirements. For example, if one of the selected input variables refers to the time limit of a crash analysis, the total wall time may vary dramatically.

- Independent jobs

Each job is independent to all other jobs. There is no dependence chain in job execution. Only a single dependence is present, the Monte Carlo analysis is finished when all jobs are finished.

- Automatically generated

  All the jobs to be launched, usually up to thousand, are created at the same time. Once the user has selected all the input parameters and output results, Promenvir automatically generates one shell script file for each job. All the script files are identical except for a new file dependency, which contains the values for the input parameters for each specific simulation. The script file also includes the command lines for the solver execution. A command line to patch the input files with the random parameters is also included in the shell script. Finally, the execution of a filter command extracts the output data results from the output files.

- Output data size

  The size of the output data is usually low in most problems (car crash analysis, satellites...). Although the size of the output data is dependent on the problem definition, it usually takes up to a KB, because only the output results are sent back to the submission host.

- Coarse granularity

  Each job is described with a shell script file, that includes the execution of programs (solvers). The execution time of these programs can vary from seconds up to hours or days.

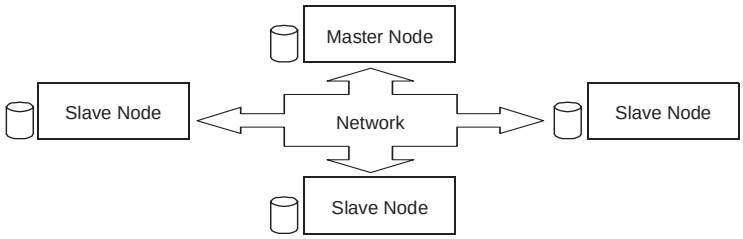
- More jobs than processors available

  The number of jobs is usually greater than the available number of hosts. The number of jobs is bounded to thousands, and the available platforms (in usual sites) are limited to one hundred (in the optimistic situation). Software licenses are also a limit for the available platforms, because only the hosts with a valid license should be considered as available hosts.

5.4 Unix-like platforms and architecture

The host requirements for Promenvir are limited to all Unix like platforms. Figure 19 represents the architecture used. There are two different types of nodes: master and slave.

Figure 19. Execution environment. Network connection



The master node is the responsible of managing all the executions in the different nodes (through the scheduler), it contains the input data and it will contain all the output results. This node is, usually, the workstation of the Promenvir user. The software for the master node is based in Unix system calls, C programming language, Motif graphic interface and sockets. The portability of the code is based in these restrictions. The software of the master node is responsible of:

- Distribute the workload all over the accessible machines
- Recognize the finalization of the work
- Display information of the accessibility and the current load of the available machines
- Suspend, restart and kill jobs

Each of the nodes accessible using the network, including sequential, vector and parallel machines, can be used as slave nodes. In each slave node, one or several solver execution can be performed. In each slave node, a daemon program is in charge of all launched executions. The daemon for the slave node is based in Unix system calls, C programming language and sockets. The daemon code has been tested in several platforms (SGI, HP, IBM, SUN, and DEC), thus allowing the user to employ all of them to solve the problem. The daemon is responsible of:

- Spawn the local jobs
- Recognize the finalization of each local job
- Suspend, restart and kill the local jobs
- Notify the master node about the finalization of each job

5.5 Workload scheduling

Promenvir has been designed to support heterogeneous and non-dedicated systems. Heterogeneous means for different vendors and different architectures (sequential, vector and parallel machines). The only restriction is the accessibility of the solver in that architecture. For example, if the user is analyzing a car crash using the Pamcrash an execution involving a Cray T3E and an Origin 2000 is possible if the Pamcrash code is available for both architectures. Non-dedicated systems means that the machine is not supposed to be exclusively used by Promenvir.

In order to allow the scheduler to select the best job scheduling, some parameters should be used to model the different platforms. As the platforms are from different vendors, with different architectures, with different memory size, these parameters must be normalized to allow a fair comparison.

Hardware manufacturers often compare processors using different benchmarks: LINPACK, SPEC,.... These benchmarks computes the MIPS (million of integer instructions per second), the scalar MFLOPS (million of float instructions per second) and vector MFLOPS. A different family of benchmarks can be used to model I/O rate, communication parameters, memory performance, and the whole system performance. Although all those parameters provide an initial value for the comparison between machines, Promenvir does not intend to use any of them because, in

the first hand, the machines are non-dedicated and, in the second hand, the large number of Promenvir jobs allow us to use more accurate estimators. For example, most jobs in a Monte Carlo analysis are identical, so the response time for a single job is estimated from previous executions response time. This already computed response time would be used to compare the current performance of all platforms running this specific problem.

The scheduling policy we have selected is FCFS. The reasons for such selection are:

1. Simplicity

   This is the simplest scheduling policy. The incoming jobs are queued until a host is available, then the job is assigned to that host. When the job is finished, the host becomes free.

2. Single user oriented

   Promenvir scheduler is oriented to a single user and scheduled jobs belong to a single Monte Carlo analysis. The unfair situation of different users and jobs with priorities is not present in our environment.

3. Fairness distribution in non-heterogeneous systems

   The number of jobs that Promenvir is going to send concurrently to a host is a user define parameter, usually equals the number of processors per hosts or equals to the number of solver licenses per host. With these premises, the scheduler maps as many jobs as possible when all jobs have been generated. When each job finishes, a new one is launched in the free host.

4. Adaptability to non-dedicated systems

   With this scheduler and the Promenvir workload, the influence of non-dedicated systems is also taken into account. If a host is overloaded, the current job will remain running for a long time, and thus this job will not be used to map new jobs until it becomes available. If a fixed scheduling is applied, then all jobs mapped to an overloaded host will be delayed, and this will produce a global delay. With the current scheduler, the overloaded hosts will be considered as a slow machine and, consequently, fewer jobs will be submitted to it.

   This mechanism also takes into account other influences than the number of running processes. In fact, if a Promenvir job requires a huge amount of disk I/O, the performance of the machine will decrease dramatically if another process in the system performs I/O in the same disk. In this situation, the processor of the host is not overloaded, but there is a bottleneck in the disk access.

5. Adaptability to host availability

   If a host comes down or it becomes not available because of the time windows, it will not be used for any other job. If a fixed scheduling is planed then a rescheduling should be applied. It also adapts to a new available host, just mapping the first pending job to it.

6. Traffic congestion recognition

   Promenvir uses Internet to copy the input files from the master node to the slave nodes, and to get back the results. As Promenvir includes these copies in the job, any traffic congestion in the network will be taken into account directly when scheduling the job. In these cases, the hosts with slowest network connection will receive fewer jobs as the scheduler recognizes them as slowest machines.

5.6 Implementation issues

For each job, the user defines in the problem definition phase which are the required input files, which are the command lines to execute, and which is the output file. Here follows a job description file, where some keywords are employed to define the different requirements. A job is fully defined with this syntax.

Figure 20. Job scripts

```
1  #& promenvir file
2  #& task number 12691_0_290
3  #& input Promenvir.Data.12691_0_290
4  #& input beam.PRJ
5  #& input beam.dat
6  #& input beam
7  #& output promenvir.out.12691_0_290
8  #& resources memory 10.0 Mb
9  #& resources temporal disk space 16.0 Mb
10  #& deadline now + 1
11  ./beam
```

The steps to execute the different jobs are:

1. File analysis

   Check the file syntax and analyze all parameters. These parameters include resource requirements and job dependencies. Before the host selection phase, all the jobs dependencies should be satisfied. For example, in Figure 20 different dependence and limits are:

   - Files: using the keyword “input”, the user can specify as input files as necessary. If those files are generated through the execution of a different script, this will produce a job dependence. With the keyword “output”, the user specifies the output files.
   - Resources: as memory, disk space, etc.
   - Execution limits: when to stop the execution of this job.

2. Host selection

   Select a host from the available host pool. The job requirements must be accomplished. If there is no an available host, the job will remain pending. The host's list is ordered according to previous jobs response time. The time windows are also managed in this point, avoiding scheduling jobs to non-accessible machines.

3. Job description file modification

   The job description file is modified to include the necessary statements to change to the working directory and to copy the input files from the master node to the slave node. Those statements use the rcp (Remote CoPy) or scp (Secure CoPy) command. The copy back command line is also appended to the end of the shell script file.

Figure 21. Job scripts after host selection

```
1  #& promenvir file
2  #& task number 12691_0_290
3  #& input Promenvir.Data.12691_0_290
4  #& input beam.PRJ
5  #& input beam.dat
6  #& input beam
7  #& output promenvir.out.12691_0_290
8  #& resources memory 10.0 Mb
9  #& resources temporal disk space 16.0 Mb
10  #& deadline now + 1
11  #& replicator 1
12  cd $HOME/.ProMenVir/work/12691_0_3
13  $EXEC/checking_rcp Promenvir.Data.12691_0_290 rcp promen 156.1.242.9 ~/.ProMenVir/
tasks/Promenvir.Data.12691_0_290
14  $HOME/.ProMenVir/bin/IRIX/checking_rcp beam.PRJ rcp promen 156.1.242.9 ~/IV/BEAM/
beam.PRJ
15  $HOME/.ProMenVir/bin/IRIX/checking_rcp beam.dat rcp promen 156.1.242.9 ~/IV/BEAM/
beam.dat
16  $HOME/.ProMenVir/bin/IRIX/checking_rcp beam rcp promen 156.1.242.9 ~/IV/BEAM/beam
17  $HOME/.ProMenVir/bin/IRIX/idg beam.PRJ Promenvir.Data.12691_0_290
18  ./beam
19  $HOME/.ProMenVir/bin/IRIX/ofp beam.PRJ ? promenvir.out.12691_0_290
20  rcp promenvir.out.12691_0_290 promen@156.1.242.9:~/.ProMenVir/log
21  rm -f task.12691_0_290 promenvir.out.12691_0_290 Promenvir.Data.12691_0_290
```

Input files for each execution are copied conditionally. If the file already exist from a previous execution, it is not necessary to copy it again. This improvement reduces the communication over the network, and it can be applied because of the workload characteristics, where all the input files for the different jobs are identical until files are patched with the job specific values.

Figure 21 contains the same example in Figure 20 but once the job has been selected and the script is modified with the necessary additional command lines. Line 12 changes to the working directory, the location where each job has an independent virtual machine to work with. From line 13 to 16, all the copy command lines are included. Line 17 corresponds to the cloning of the input template with the data values corresponding to this specific job. Line 18 is the solver execution (in this example, a simple command line; but user can specify any shell script). Line 19 is to filter from the output files the requested values. Line 20 is to copy back the output files. And, finally, line 21 is to clean-up the working directory. All the files are not removed, because the scheduler can select the same virtual machine for the execution of a new job, and then the files (for example, beam and beam.dat can be reused).

It is also important to remark that the copy statements are using a shell script. This allows to use rcp, scp or any other copy command.

4. Start job

A command line is build in the master node and send to the daemon at the slave node. It forks a new process and executes the shell script. The user, when selecting the available platforms, is able to define which is the mechanism to start jobs: different shells, submit the job to a local queue manager, …

5. Job finalization

   When the job exits, the daemon notifies the master node about the finalization.

5.7 Conclusions

The job scheduler included in Promenvir performs most of the functionalities of commercial queuing models, but it is oriented to a single user and to specific analysis. It allows the user to define and use a network of heterogeneous computers as a single system, to restrict access to host using time windows, to define resource availability and limits.

Portability is supported from different points: the user interaction is supported through graphical interface, based on Motif; job description is done using shell scripts and remote access and communication is supported with standard Unix utilities.

Finally, the scheduler itself based in FCFS and using the response time for the last executed jobs offers good scheduling mapping and performance. This latest feature can be introduced because of the Monte Carlo workload characteristics.

CHAPTER 6

Dimemas validation

It is necessary to perform Dimemas validation from two different points of view. First, to validate its capacity to reproduce the behavior of an application in the architecture we have used for instrumentation. And second, to validate the accuracy of the simulator for performance prediction over non-accessible architectures.

In both scenarios, it is require to obtain the values to fit the interconnection network (latency and bandwidth) and know about the specific implementation of the collective operations. It is always feasible to use vendors parameters, but for validation we choose values obtained in the running system.

Dimemas validation is based in NAS Parallel Benchmarks. This is a set of seven parallel programs, implemented on top of MPI. Those programs are prepared to provide execution time and performance (expressed in MFLOPS). These benchmarks are the most popular in bibliography for benchmarking parallel system, but there are two additional reasons for us to select this programs: first, they are prepared to work with different number of tasks, it is just a matter of recompilation; and second they also deal with several data sizes. Former characteristic modifies the communication pattern and the blocking times of the parallel execution. Latter one, modifies processor time consumption and messages sizes, thus modifying the whole application. So, whenever we specify a program, the full name includes the number of processors and the class (memory size). The different options for classes are: S(sample), W(Workstation), A, B, or C.

Different types of experiments have been performed in order to check the quality of the predictions and usefulness of Dimemas. They are classified in the following groups:

- Qualitative validation: The objective is to show graphically how Dimemas enables the user to tune and develop parallel programs even if no dedicated parallel machine is available.
- Stability validation: to quantitatively support the above statement.

- Parameter optimization for microbenchmarks: investigating the range of Latency and Bandwidth values that lead to fairly good prediction results for microbenchmarks that intensively stress some of the MPI calls.
- Verification of model assumptions: checking how well simple models in the core of Dimemas match with current experiments. Specifically:
  - Linear communication model: Latency and Bandwidth parameters.
  - Collective model as a function of number of processors.
- Quantitative validation: comparison of execution time and predicted time, using SGI O2000 to get the traces. The prediction time will be computed for SGI.

6.1 Network parameters

Latency and bandwidth are obtained with the Ping-Pong benchmark. The main loop of the program is shown in Figure 22. It consist in sending back and forward a message.

Figure 22. Ping-Pong benchmark

```
1MPI_Send (message, 0, MPI_CHAR, 1, TAG, MPI_COMM_WORLD);
2  MPI_Recv (message, 0, MPI_CHAR, 1, TAG, MPI_COMM_WORLD, &status);
3  for (k=0; k<COUNT; k++){
4       timer_clear (k);
5       timer_start (k);
6       for (j=0; j<MAX_ITER; j++){
7         MPI_Send (message, num_bytes, MPI_CHAR, 1, TAG, MPI_COMM_WORLD);
8         MPI_Recv (message, num_bytes, MPI_CHAR, 1, TAG, MPI_COMM_WORLD, &sta-
tus);
9       }
10      timer_stop (k);
```

Figure 22 code corresponds to the Ping thread. The initial two lines tries to get the best synchronization of the involved threads. Loop starting at line is to repeat the experiment several times, to determine if there is variation in the measurements. Subroutines calls 4, 5 and 10 annotates wall clock time, including block time and communication time. To conclude, loop in line 6 is to increase the granularity of a single measurement, in order to avoid any problem with clock precision.

If the necessary time for doing MAX_ITER communications is T, we can approximate the necessary time for each communication with

$$T_c = \frac{T}{2 \times MAXITER}$$

because each iteration of the internal loop performs two communications.

Table 8 presents the execution results for SGI Origin 2000, with 64 processors R10000 (resource available at CEPBA, European Center for Parallelism of Barcelona, of the Technical University of Catalonia).

TABLE 8. Ping-Pong benchmark results for SGI O2000

Bytes	Count	Sum	Average
0	128	0.002896077	0.000023
1	128	0.003044482	0.000024
2	128	0.003024237	0.000024
4	128	0.002309194	0.000018
8	128	0.002327217	0.000018
16	128	0.002348785	0.000018
32	128	0.002467115	0.000019
64	128	0.002684582	0.000021
128	128	0.003554742	0.000028
256	128	0.003944063	0.000031
512	128	0.004625256	0.000036
1024	128	0.004112000	0.000032
2048	128	0.005346608	0.000042
4096	128	0.008004908	0.000063
8192	128	0.016106396	0.000126
16384	128	0.028667020	0.000224
32768	128	0.055859816	0.000436
65536	128	0.104688040	0.000818
131072	128	0.189579208	0.001481
262144	128	0.388594132	0.003036
524288	128	0.807166692	0.006306
1048576	128	1.468302348	0.011471

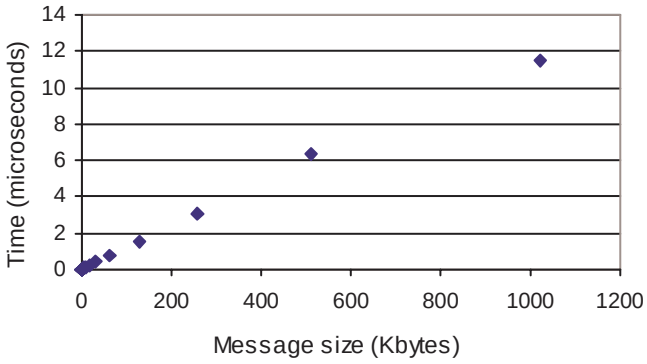
Figure 23 is the graphical representation of Table 8. Computing the best fit line, using the minimum square method, we obtain the following equation:

$$Time = 0{,}000044 + 1{,}1122 \times 10^{-8} \times size$$

Thus, bandwidth and latency values are 85.74 Mbytes/second and 44 μseconds. But, if we limit the message size of the Ping-Pong benchmark to 1024 Bytes, the

obtained parameters are 64Mbytes/second and 22 µseconds. This different values is significative, but not surprising, because different protocols are implemented within each MPI library to support different message sizes. This is the reason some bibliography concludes the lineal model based in latency and bandwidth is not a good approach. This is not a negative point for our proposal because we are interested in having an approximate model with an average accuracy. We propose to use the values according to the appropriate message sizes used by the application.

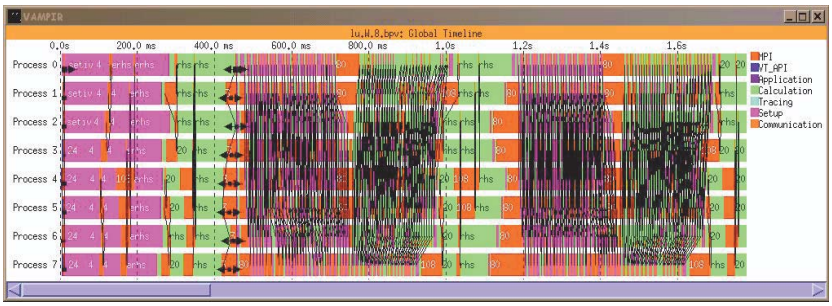
Figure 23. Ping-Pong benchmark results for SGI O2000



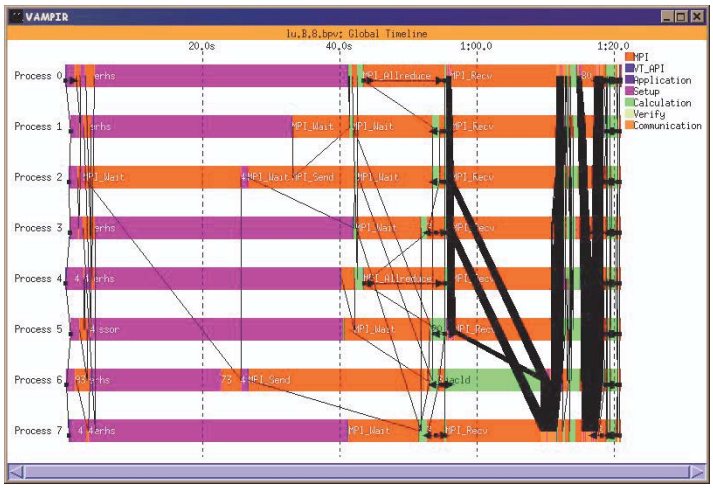
6.2 Qualitative validation

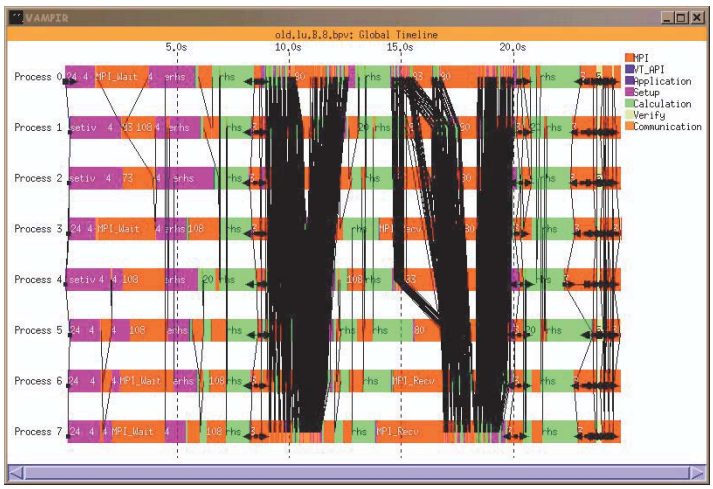
To show graphically the capability of Dimemas to rebuild the temporal behavior of an MPI program as if it had run on a dedicated parallel environment we run the NAS LU (class W) benchmark on a dedicated machine (SGI O2000). The resulting trace is shown using Vampir in the following figure.

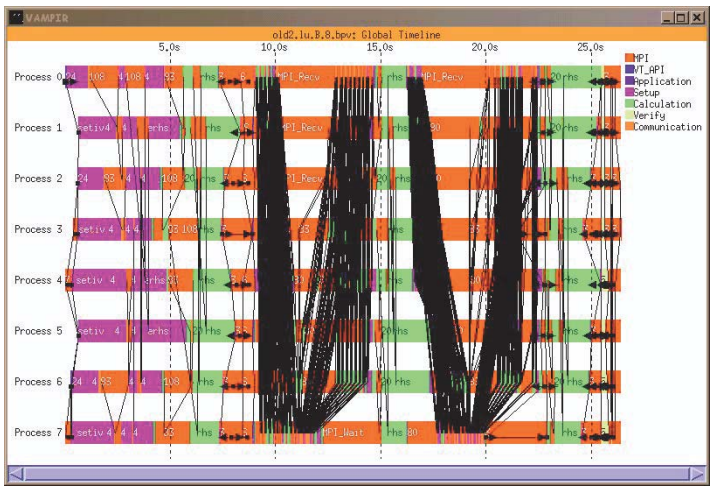
Figure 24. NAS LU (class W), on dedicated machine

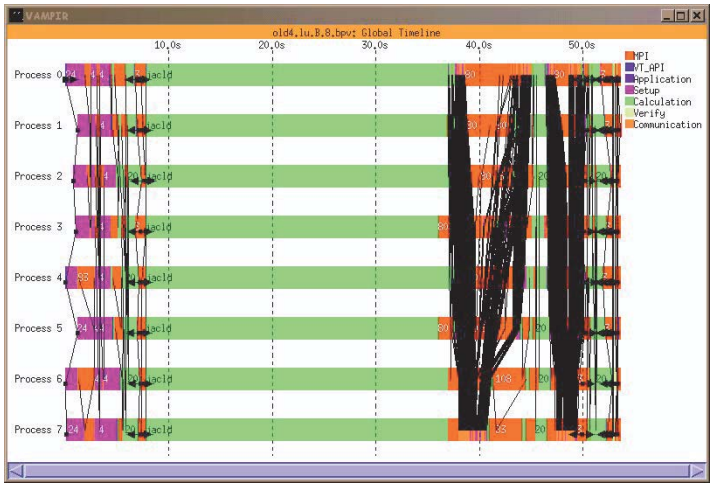


The same test case was run 4 times in a heavily loaded system. The four identical runs generated the following traces


VAMPIR
lu.B.8.bpv: Global Timeline
20.0s
40.0s
1:00.0
1:20.0
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
MPI
MT_API
Application
Setup
Calculation
Verify
Communication


VAMPIR
old.lu.B.8.bpv: Global Timeline
5.0s
10.0s
15.0s
20.0s
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
MPI
MT_API
Application
Setup
Calculation
Verify
Communication

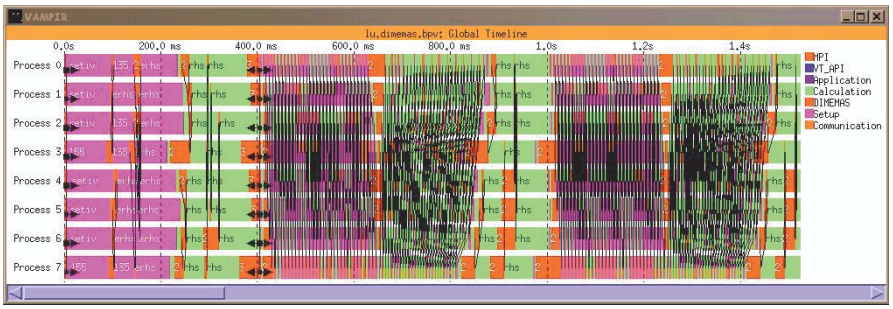

VAMPIR
old2.lu.B.8.bpv: Global Timeline
5.0s
10.0s
15.0s
20.0s
25.0s
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
MPI
MT_API
Application
Setup
Calculation
Verify
Communication



The four runs had totally different elapsed times and although the communication pattern are certainly the same, it is not possible to draw conclusions about what is the performance limiting factor or where should tuning effort be concentrated.

Figure 25 has been obtained by generating a Dimemas trace in the same highly loaded environment and then running Dimemas with a latency of 28 microseconds and bandwidth of 82 MB/s.

Figure 25. NAS LU (class W), predicted using Dimemas



As can be seen, the similarity with the current parallel run is good. The Vampir trace generated in this way can be used to analyze the application and identify possible optimizations.

We could like to state other type of qualitative validation coming from an end user statement: "I can identify better the structure of my application on the Dimemas generated Vampir trace than on a real Vampir trace".

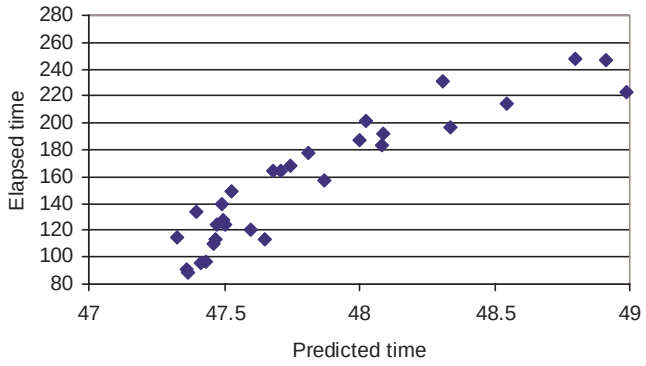
6.3 Stability

The question may arise of what would happen if we repeat the tracing and Dimemas process. Are the results stable enough? Will the huge variation observed with the visualization tool appear again?

It is certainly true that several instrumented runs will generate different Dimemas traces. Memory related issues are the major source of variation. Cache pollution generated by the multiprogramming OS scheduling policies will have the major effect, but also data placement issues and process migrations will be important in a NUMA machine.

To evaluate this effect we carried out an experiment where 30 runs of a benchmark (NAS LU on 32 processors) were run on a heavy loaded machine. For each of them, a Dimemas trace was generated and the elapsed time measured. In Figure 26 we plot the current elapsed time of the instrumented run versus the time Dimemas predicts it would have taken if run on 32 dedicated processors.

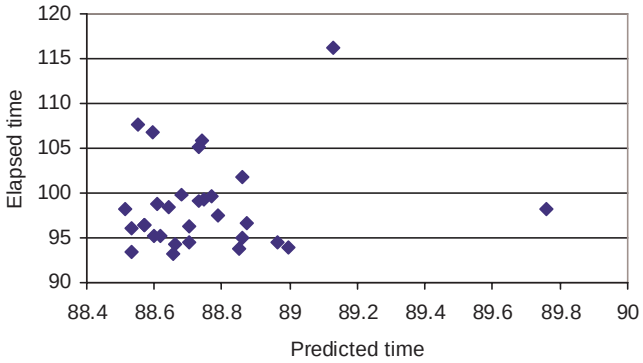
Figure 26. NAS LU, 32 processors, overloaded system, predicted versus elapsed time



As can be seen, there is a correlation between the current elapsed time and the predicted time, showing that the runs that experienced more delay also introduced more perturbation in the trace due to cache problems. The variability of the predicted time is nevertheless minimal compared to the variations in elapsed time (4% vs. 210%)

Figure 27 shows a similar experiment (LU) with 16 processors. In this case, the variability of predicted time is below 2%. The variability of elapsed time is above 20%. As can be seen the more processors are used, the more predicted and elapsed time variance is, but the variance for the predicted time is kept within a reasonable value for this benchmark.

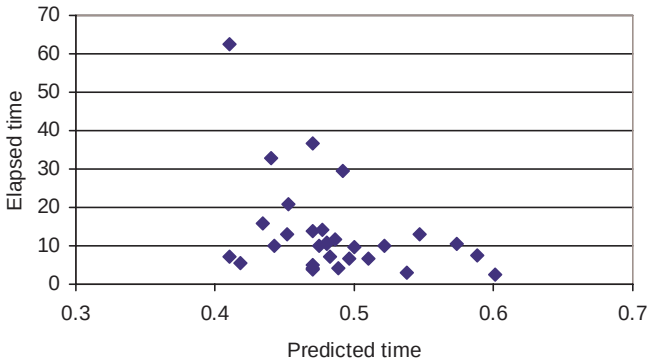
Figure 27. NAS LU, 16 processors, overloaded system, predicted versus elapsed time



Will other codes behave the same?

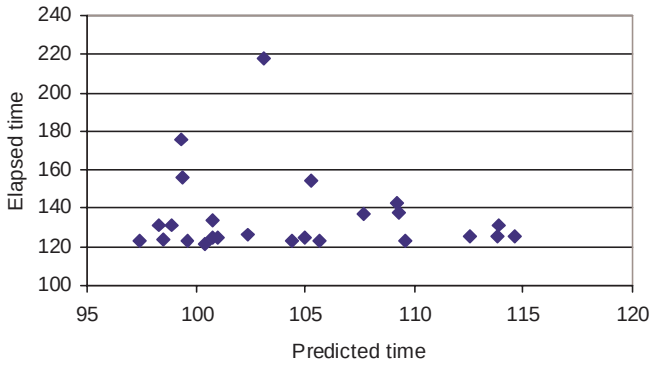
We repeated the same type of experiment with other benchmarks. The following plot corresponds to the Barrier benchmark from the NPB benchmark suite. The results show that the predicted range spans a variation of 39% while the variation of the elapsed time is 452% (256% without the outlier). Even if we eliminate the outlier taking more than 60 seconds, the variation of the real run is much higher than that of the predicted time. The variance of the prediction is nevertheless much higher that in the previous case (LU). This has to be attributed to the fact that the Barrier benchmark does not carry out any useful computation between successive MPI calls. The CPU burst that the Dimemas tracing generates are thus very small, and a minimal error in the clock precision reflects directly on the total predicted time.

Figure 28. Barrier benchmark, overloaded system, predicted versus elapsed time



Finally, we performed 25 samples of the SP benchmark with 16 processors. The elapsed time variability is 73% and the predicted time variability is 16%. SP is a benchmark with very small CPU bursts between communications. This is reflected in larger variability that the LU benchmark. We nevertheless consider that 16% is quite acceptable.

Figure 29. NAS SP, 16 processors, overloaded system, predicted versus elapsed time

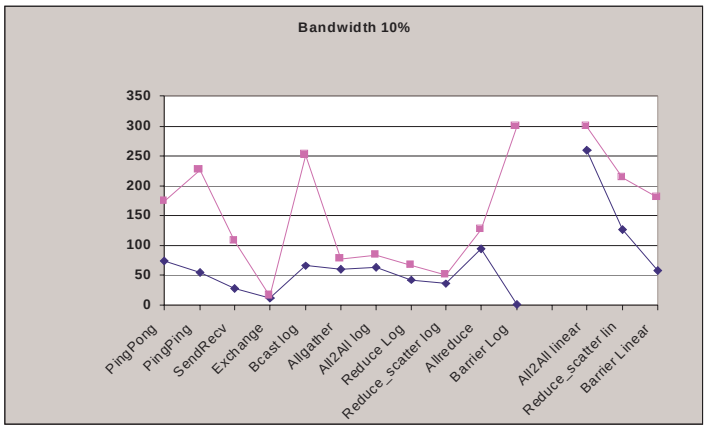


Our claim is that the quality and stability of results makes of Dimemas and adequate methodological tool for performance analysis of message passing programs.

6.4 Parameter optimization for micro-benchmarks

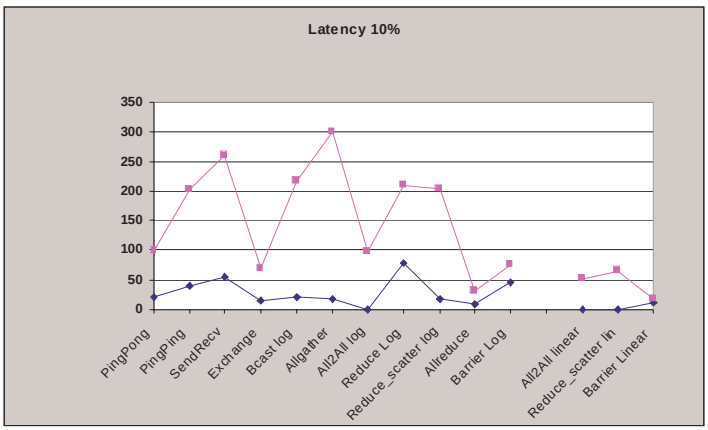
In order to evaluate the quality of the predictions, we performed an optimization process for different micro-benchmarks that intensively stress some of the communication primitives in MPI. Each of them was run with dedicated resources, a Vampir trace with dedicated resources was also obtained and then a Dimemas trace was obtained in a loaded system. Then, for each of the benchmarks a whole bunch of simulations, using Promenvir, was done trying to fit the elapsed time of the dedicated run by using the Dimemas trace and varying the latency and bandwidth parameters.

Figure 30. Fitting bandwidth to predict time with error <10%, simple collective communication model



The plots in Figure 30 and Figure 31 show the ranges (minimum in blue, maximum in pink) of latency and bandwidth that lead to predicted time within 10% of the dedicated elapsed time. On the X-axis, different benchmarks and collective models are presented. The first continuous lines show the results for logarithmic models of collective operations and the second part (last three points) the results when the model used is linear.

Figure 31. Fitting latency to predict time with error <10%, simple collective communication model



We were expecting to get a range of latencies and bandwidths that would make all benchmarks to be predicted within 10% accuracy. Clearly, Exchange and Allreduce need very different bandwidths than the other benchmarks, while something similar happens for Reduce with latencies. We also observe that SGI would object to publication of the most favored bandwidth (short above 50MB/s). Where is the problem? Is the Dimemas model very simple?

Looking at conceptual issues not considered in the above fit, the first one that arises is contention in the network. All our simulations were performed with unlimited bisection bandwidth. Even more, each node had one input and one output link, allowing for simultaneous sends and receives to take place. The results for the Exchange benchmark go into more reasonable values if the fit is repeated assuming a certain level of contention in the network. In our case, we repeated the study using only 8 buses. The results for the Ping-ping benchmark are sensitive to the ability of one node to have two simultaneous transfers concurrently. For this benchmark, it is possible to simulate the single link connection by assuming a single bus in the network. By doing so, the needed bandwidth approaches that of the Ping-Pong.

A second issue that arises is the conceptual structure of the collective operations. Many of them have two phases: first, some information/synchronization is collected (Fan in) and then the result is distributed (Fan out). Our original model does not consider this structure. Is it possible to extend the model in a simple way to account for this structure? How much benefit will be gained from that in terms of quality of prediction? And in terms of complexity of use?

6.5 Validation for microbenchmark and extended model of collective communication

By analyzing the semantics of the collective MPI primitives, a reasonable model would be:

TABLE 9. Input values for modeling collective communication

MPI call	Fan in model	Fan in size	Fan out model	Fan out size
MPI_Barrier	Lineal	Max.	0	
MPI_Bcast	Log	Max.	0	
MPI_Gather	Log	Mean	0	
MPI_Gatherv	Constant	Max.	Constant	Max.
MPI_Scatter	Log	Mean	0	
MPI_Scatterv	Constant	Max.	Constant	Max.
MPI_Allgather	Lineal	Min.	Lineal	Min.
MPI_Allgatherv	Lineal	Min.	Lineal	Min.
MPI_Alltoall	Lineal	Min.	Lineal	Min.
MPI_Alltoallv	Constant	Max.	Constant	Max.
MPI_Reduce	Log	Max.	0	
MPI_Allreduce	Log	Max.	Log	Max.
MPI_Reduce_Scatter	Log	Max.	Lineal	Min.
MPI_Scan	Log	Max.	Log	Max.

Where we have not tried to estimate the level of contention and assumed it is not present.

The resulting range of latencies and bandwidths that lead to less than 10% error in the prediction are presented in Figure 32 and Figure 33.

Figure 32. Fitting bandwidth to predict time with error <10%, extended collective communication model

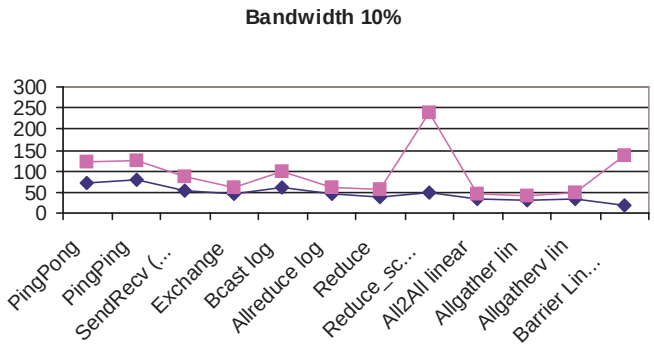
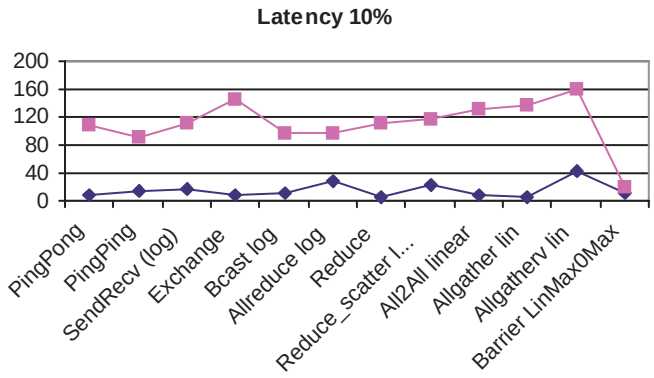


Figure 33. Fitting latency to predict time with error <10%, extended collective communication model



Our conclusion is that the proposed model is simple enough to capture acceptably well the performance of communication primitives.

6.6 Verification of model hypotheses

Before continuing, we could ask questions such as: Are the basic models correct? Is a linear model (latency and Bandwidth) appropriate? Do collective operations scale in a linear or logarithmic way?

6.6.1 Linear communication model

Figure 34 shows the real time per message reported by the Ping-Pong micro-benchmark and the linear model obtained from it. A linear regression model with all the data leads to the values L=-141.76 microseconds and BW=86.96. The negative latency is hardly acceptable from the conceptual point of view. A little bit of manual tuning leads to L=19 microseconds and BW=82 MB/s. From the figure, one would argue that the fit is good.

Figure 34. Ping-Pong real time and linear model

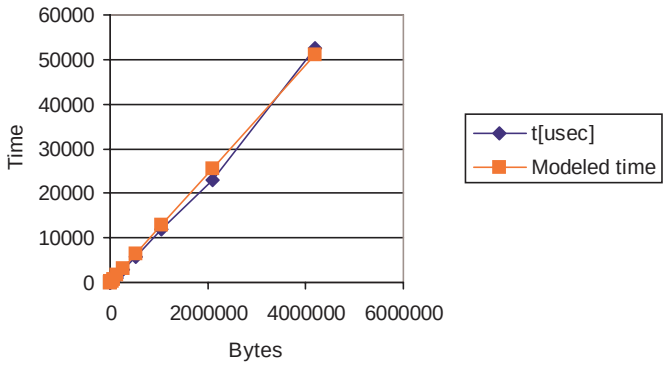
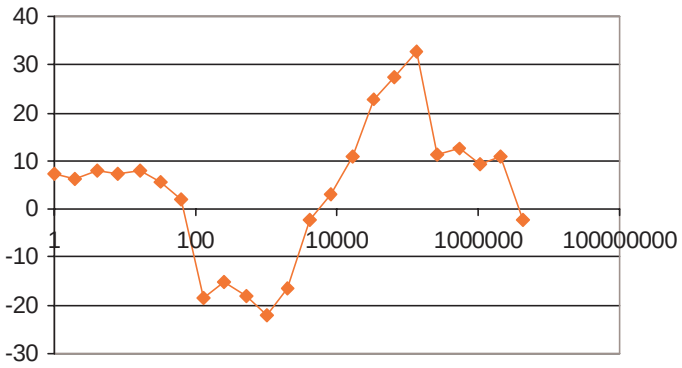


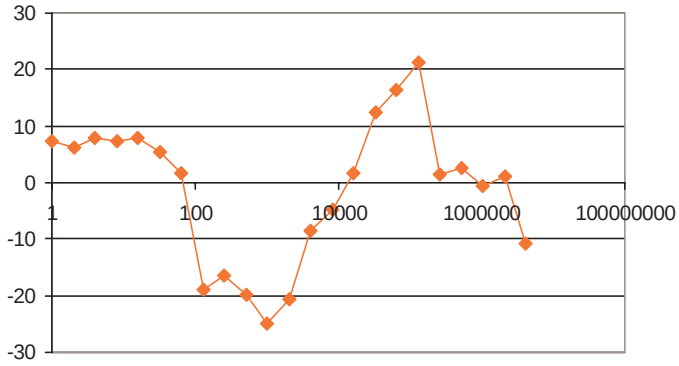
Figure 35 plots the percentage of error between the linear model and the reported time in a logarithmic message size scale. We can see that there are substantial differences at certain message sizes.

Figure 35. Percentage of error between the linear model and the reported time(L=19μs, B=82 MB/s)



If we change the bandwidth to 90 MB/s, the plot of errors is shown in Figure 36.

Figure 36. Percentage of error between the linear model and the reported time (L=19μs, B=90 MB/s)



Similar variations in the quality of the linear model as a function of message size are observed for many other primitives. The question that may be risen is, should we move away from the linear model? Our opinion is that certainly NOT.

6.6.2 Scalability model

How do collective operations scale? Let us consider the Barrier benchmark, run on a dedicated machine, varying the number of processes between 2 and 32. We can fit the values by a linear or logarithmic expression. The results show that none of them fits very well. Considering that the value for 32 processors as an outlier and fitting again, the model seems to be better approximated by a linear fit. In any case, some questions may arise: what should be done if one tries to predict the performance for a 32 processors application? What happens for 64 processors?

Figure 37. Execution time vs. number of processors for Barrier benchmark

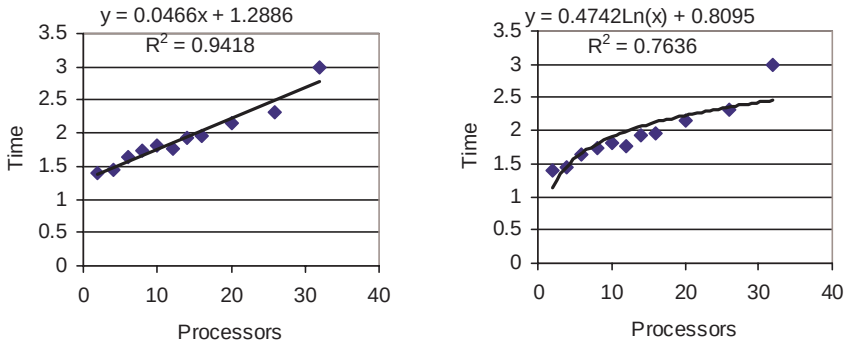
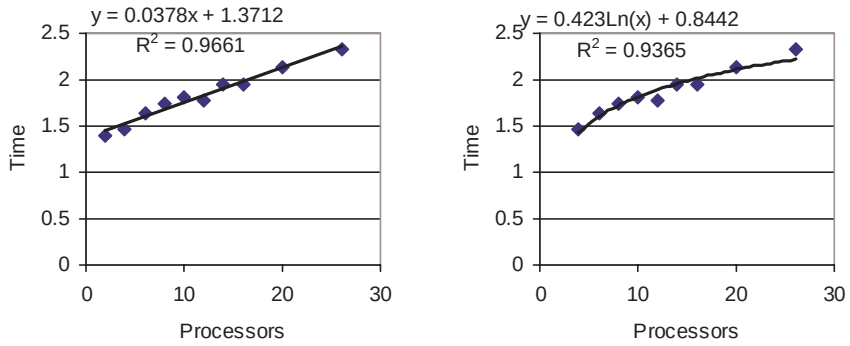
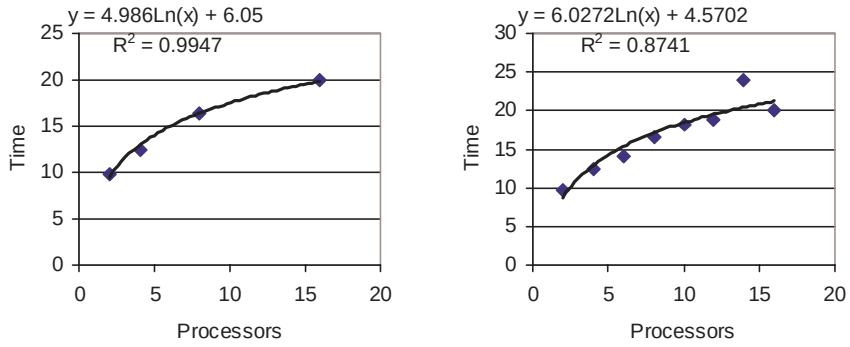


Figure 38. Execution time vs. number of processors for Barrier benchmark, without outlier



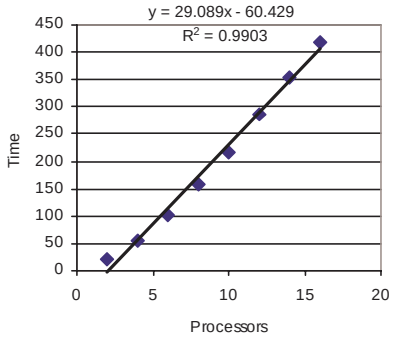
Doing a similar study for the Broadcast benchmark and using only numbers of processors equal to powers of two, we see how a logarithmic fit is good. A bit different fit is obtained if data for more numbers of processors is used, and the fit has some more error.

Figure 39. Execution time vs. number of processors for Broadcast benchmark



The time for an All to All primitive is fitted well by a linear model.

Figure 40. Execution time vs. number of processors for All to All benchmark



These results show that different primitives may have different scalability models but in general, a very concerned user of Dimemas may want to use a different detailed model for each primitive. The proposed models in Table 9 provides such functionality.

6.7 Quantitative validation

In this section, we present the prediction quality of Dimemas as a comparison on real execution time and predicted time.

6.7.1 Measuring SGI, prediction for SGI

We have selected the benchmarks BT, CG, FFT, IS, LU, MG and SP, with the classes W and A. The number of tasks for the experimentation is 8, 16 and 32 for CG, FFT, IS, LU and MG; and 9, 16 and 25 for BT and SP (because both benchmarks require a exact square root number of tasks).

The parameters we will use for this validation are:

- Latency = 25 μseconds
- Bandwidth =87.5 Mbytes/seconds
- 1 half-duplex link
- 16 buses
- Collective operation model: refer to Table 9

Figure 41 presents the predicted time versus the real execution time. Each point represents a different experiment (benchmark, class, tasks). The lineal model fits very well with this data representation, showing that prediction and measuring data are very close. The problem in this representation is that most benchmarks results are located in the same area, close to origins.

Figure 41. Predicted time vs. real execution

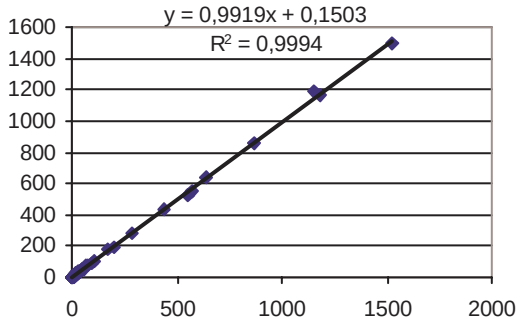


Figure 42 presents the same results but with a different graphic. In the X axis, we present the 42 different benchmarks. Dedicated is the application time in a dedicated environment. Instrumented is the application time for the same application but tracing the execution. And, finally, prediction is the predicted time for the application.The accuracy seems to be adequate once again, because once again some benchmarks are having a big execution time, and have a lot of influence in the graphical representation.

Figure 42. Predicted, Instrumentation and Dedicated time

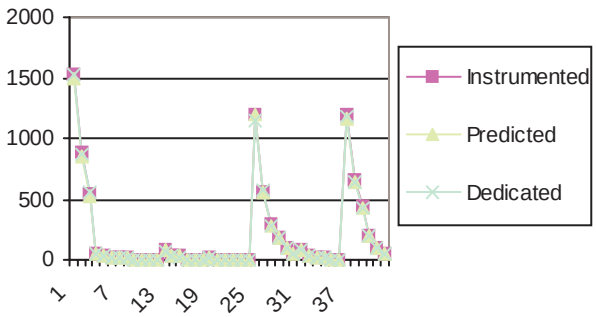
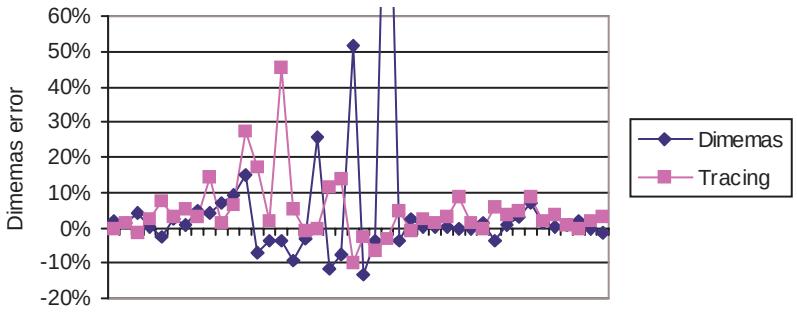


Figure 43 presents the same results are presented as error. Most of the benchmarks are predicted with less than a 10% of error. The point out of the graphic takes the value 150% error. This error and those over 10% refer to really short executions (less than five seconds). Thus the real difference between execution and prediction is negligible.

This figure also shows the error introduce by the tracing facilities. To measure this error, we have included the instrumentation routines, and execute the application in standalone mode, thus the difference in execution time is due to the tracing facilities. This error is due to the influence of the tracing facilities in the execution time, and in some cases this error is also important. This extra error source reduces, in

some form, the error introduced by Dimemas, as part of Dimemas error can be explained by the tracing error.

Figure 43. Dimemas error in prediction of SGI O2000



CHAPTER 7

Methodology for the analysis of application performance

This chapter describes the methodology we propose to analyze application performance, to locate those factors that limit the performance, and to predict the application behavior in different architecture configurations.

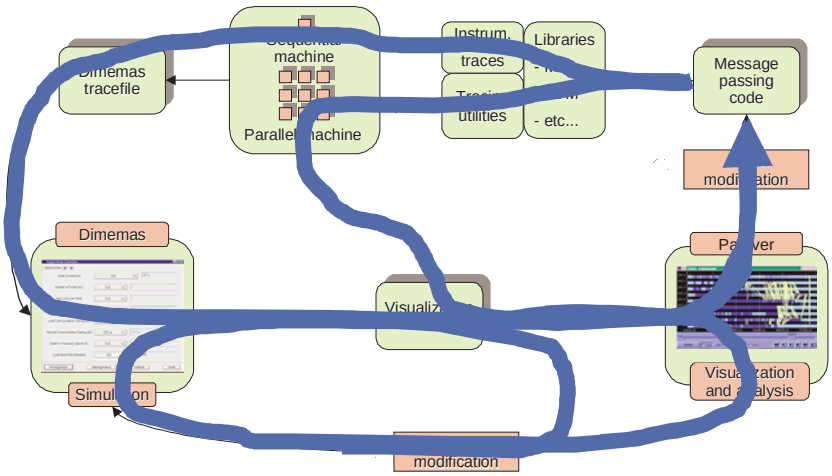
This methodology is based in two concepts: application characterization and system characterization. In Chapter 3, we propose a mechanism for application characterization. It is based in recording in a tracefile all the relevant information of the application behavior, but not including information of the host used for the measurements. Different examples in Chapter 6 demonstrate the independence between the information recorded in the tracefile and the computer used for execution. Also in Chapter 3, we present the model for system characterization, based in several parameters as latency, bandwidth (for the lineal communication model), processor relative speed (for processor comparison), number of links from each node to the network, number of buses, and model for collective communications. The correctness of these model assumptions and the accuracy is demonstrated in Chapter 6.

At this point, we have the opportunity to predict the application behavior when some of the architectural parameters or some characteristics of the application are changed, without the requirement of running again the application in the same or in a different platform.

The methodology is based in the execution of two loops shown in Figure 44, both of them located in the bottom of the Figure. The first one, the smallest one, only includes the simulator and the modifications of different parameters. It is based in running several times the simulator with different configuration parameters. All those different executions allow the location of the parameters that limit the application performance. The second loop, the one including the simulator and the visualization tool, gives the opportunity to analyze the application behavior when using different configuration parameters, but not limiting this analysis to execution time. The different possibilities of analysis provided by the visualization tool allows the user to acquire valuable information of the application and its utilization of the system resources. The visualization of the different simulations allows the user to understand much better the application and to prepare the next simulations.

Previous to this innovative methodology, there were two mechanisms to analyze performance on message passing applications. The first one was comparison of the execution time and the second one was tracefile generation for a visualization tool like Paraver. Both options requires execution of the application on a dedicated machine, because other application interference would modify the application performance, and the visualization can not eliminate this influence, and the user will understand it included in the application behavior. The requirement of code modification and new execution is the second disadvantage of these mechanisms of performance analysis, because it is quite time consuming to modify the application, it is expensive to execute it again in a dedicated environment, and in some cases these modifications and analysis are not satisfactory. And, as third disadvantage, it is not possible to predict the application performance on different platforms, as it is necessary to execute the application in the platform we want to analyze. In this last group of disadvantages, we can include the impossibility to perform analysis of application dependencies on different parameters.

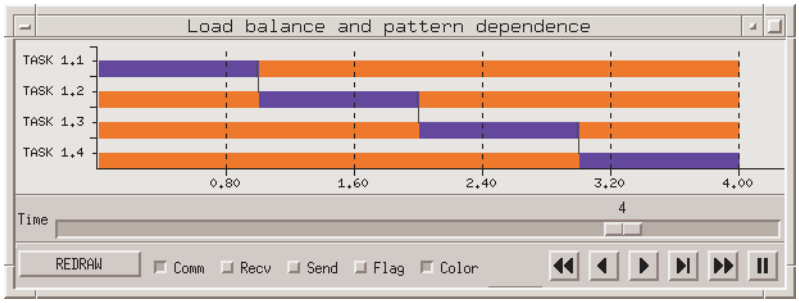
Figure 44. Dimemas and Paraver interoperability for performance analysis



What are the performance aspects our methodology allows the user to analyze? What is the procedure of this methodology? Is it applicable to all message passing applications and computers? These are the questions we will focus in the remaining part of this Chapter.

Is my application having load balance problems? Is there some important precedence in the communication pattern of my application? These questions arise when the user starts analyzing the application performance. A perfect load balance application is the application where all tasks employ nearly the same processor time. Figure 45 represents a perfect load balance application, because all tasks consumes 1 ms. An application with important precedence in the communication pattern, is an application where tasks block for message reception, and then the whole execution is delayed. Figure 45 also represents an application with important precedence in the communication pattern. In this case, although the application is perfectly load balance, the overall application performance is poor (close to 1/number of tasks) because the communication precedences produce blocks in the tasks.

Figure 45. Load balance application



Once the application is running properly for a given architecture (the obtained performance parameters are satisfactory to the user) it is important to quantify the performance in different machines and networks. The next steps of the methodology focus in the following questions: Is my application sensible to communication parameters? Is my application sensitive to latency? Shall I group communications (if possible)? Do my application require a fast network to obtain good performance? Is my application sensible to network contention or to network resources? It is reasonable to execute this application in an ethernet based network of workstations, or it is preferable to use a dedicated machine?

We will demonstrate the methodology using the LU application, class B, with 8 tasks and a 2 iterations, and the same application and class but using 32 tasks and 2 iterations. We will use the same naming as NAS benchmarks, and LU.B.8 stands for the application LU, problem size B, 8 tasks; and LU.B.32 stands for the same application and model size but using 32 tasks. LU is a communication intensive application in some areas, and processor intensive in the rest. We select class B because of the big data size, and big messages. We made this selection because the tracefile is more realistic, closer to real applications with big data sizes and big messages. We also select number of tasks to be 8 and 32 to have small and big number of tasks. Number of iterations is two, because all NAS benchmarks repeats the same iteration several times to increase the duration of the problem and to eliminate the application startup time, but we are interested in application performance and this can be analyzed in just two iterations.

This chapter is organized as follows. Section 7.1 analyzes the load balance problem and application dependencies. Section 7.2 analyzes the influence of latency is application execution time. The analysis of the influence of bandwidth is application performance is analyzed in Section 7.3. And finally, Section 7.4 analyzes the influence of contention because of network resources, in this case contention produced by the limited number of buses in network.

7.1 Load balance and application dependencies

The methodology we propose is composed of different steps and different analysis at each step. The first step is focused in analyzing the application load balance and the communication pattern dependencies. In order to obtain a maximum application performance, it is important to reduce to the minimum the blocking

time of the different tasks, as it increases the overall application time. Blocking time happens in message passing applications when a message is requested before it is sent (or just before the message is ready for reception).

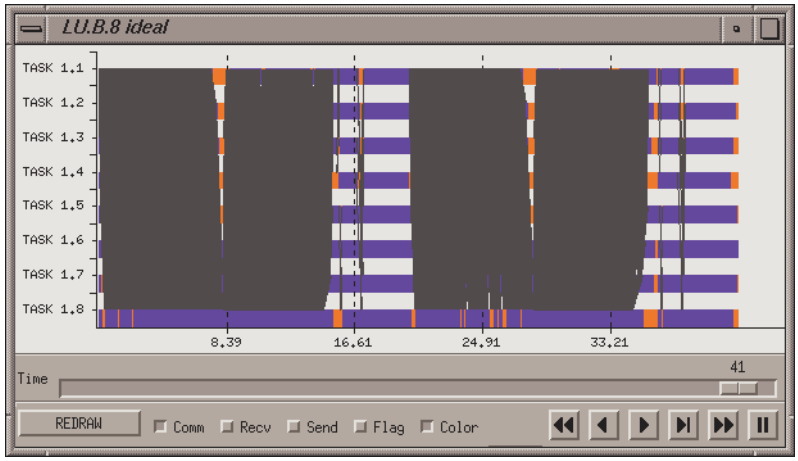
To analyze the load balance and pattern dependence problem, we provide the following parameters to the system characterization:

- Infinite bandwidth
- Null latency
- No resource contention

We use the term “ideal network” for this system characterization, because the modelled network is perfect and it does not include any delay in the execution of the application. The results we obtain with Dimemas using this ideal network are only dependent on the application characterization.

Figure 46 corresponds to the representation of two complete iterations of LU.B.8. In blue is represented application execution, in orange, block time, and vertical lines represent communications. Collective operations are not graphically represented, but in this benchmark there are only two collective communications, both of them close to the end of the execution.

Figure 46. LU.B.8



In Figure 46, and also in all Figures of this section, the communications are displayed as vertical lines or as oblique lines. Vertical lines represent a communication from a sender task to a receiver task, where the receiver task tries to receive the message before the sender task sends it. Those messages produce blocking in the receiver task. Oblique lines represent communications from a sender task sending the messages before the receiver task requests them. The message is sent and buffered in the receiver node, ready for delivery to the task. The initial point of the communication line is the send time, and the final point is the reception time. Vertical lines are an effect of using ideal network parameters.

There are two important blocking areas in the whole execution. One of this areas is zoomed in Figure 47. One can observe that tasks number 8 (in Figure with label

'Task 1.8') gets messages from tasks 4 and 7, then 4 from 3 and task 7 from 3 and 6. This creates a dependency communication chain, ensuring that task 8 is always the last to conclude computation. All other tasks request messages from task 8, and then they remain blocked, because task 8 is not ready for sending messages.

Figure 47. LU.B.8, zooming the communication intensive area

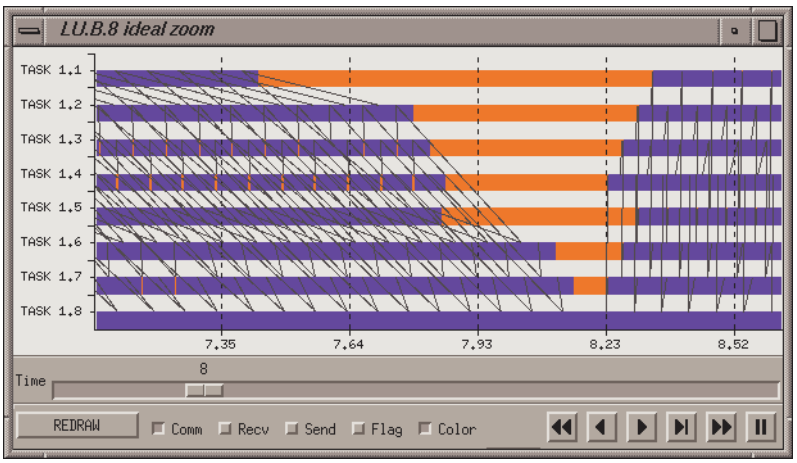


Figure 48 represents two iterations of LU.B.32, and Figure 49 the zoom for the communication intensive area. One can observe from these Figures, that increasing the number of tasks but maintaining the same problem size, the total number of messages is increased, because the communication pattern is fixed. As this communication pattern produces some blocking, when using more tasks, the total blocking time increases, and then the overall application performance is worst.

Figure 48. **LU.B.32**

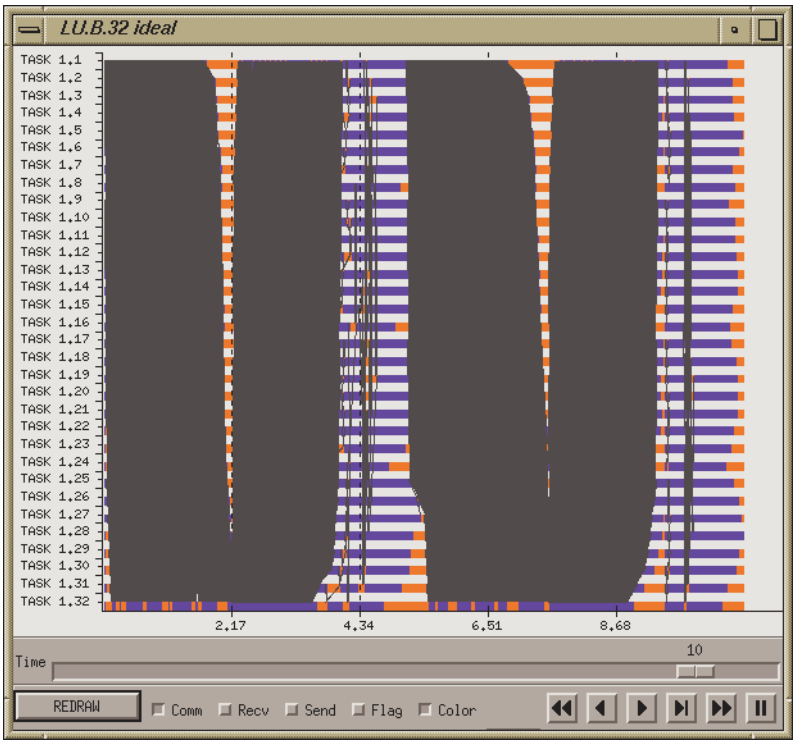


Figure 49. LU.B.32, zooming the communication intensive area

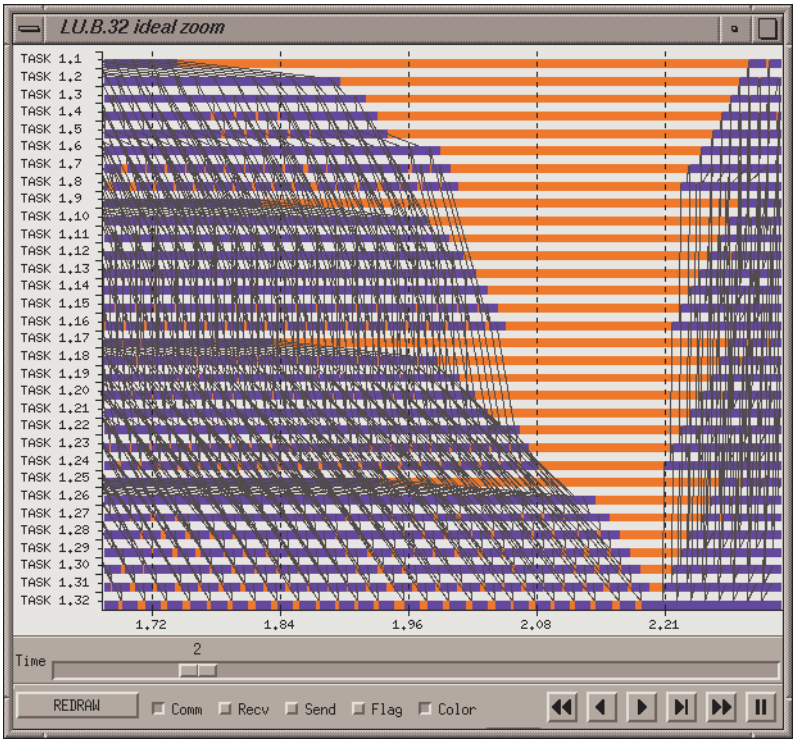


Figure 50. LU.B.8, application profile

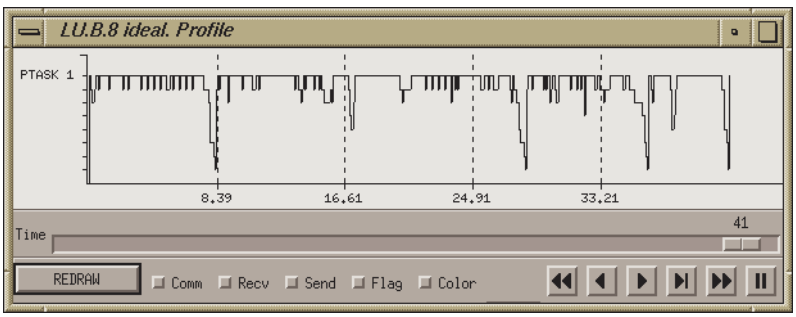
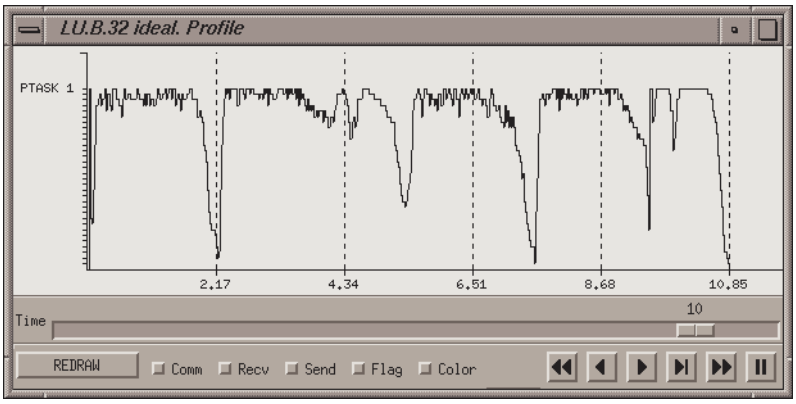


Figure 50 presents the application profile for LU.B.8 using the same scaling as Figure 46. There are four different locations where the level of parallelism decreases. Two correspond to the communication intensive area (at 8.39 and 27 of execution time) and are produced because of the communication pattern of the application. The other two correspond to collective operations (at 35 and 41.5 of execution time) and are produced by task synchronization. Figure 51 also presents the application profile but for 32 tasks application. There are no many differences between Figure 50 and Figure 51, but the latter has one additional loss of performance (at 5 of the execution time), reducing the overall parallelism to 10.

Figure 51. LU.B.32, application profile



7.2 Message grouping

One of the more important parameters in message passing environments is latency. It applies to any message being sent, and thus affects directly to application performance. Is our application sensible to latency? Do we must group messages to reduce its number, and then the total latency time?

The method we propose is to repeat several Dimemas executions with different latency values, but using constant values for other parameters (bandwidth, buses and links). And then, for some of the analyzed latencies observe the special influence in the application performance using the visualization tool. This second part of the method allows the user to analyze in detail the influence of latency in application performance. This method is based in the smallest loop (bottom left) of Figure 44.

The first part of the analysis is intended to measure the influence of network latency in application execution time. Figure 52 shows this influence for different values of network latency. We can observe significant influences for values greater than 1 millisecond. Figure 53 shows the same analysis for LU.B.32. The behavior of influence for 32 tasks and 8 tasks is quite similar (Figure 52 and Figure 53), but in LU.B.8 the total execution time goes from 41s to 72s (less than 100% increase), and in LU.B.32 the application uses from 10s to 53s (more than 500% increase). From this analysis, we can conclude that the influence of latency time is really more important in LU.B.32 than in LU.B.8.

Figure 52. Influence of latency to application time. Application LU.B.8

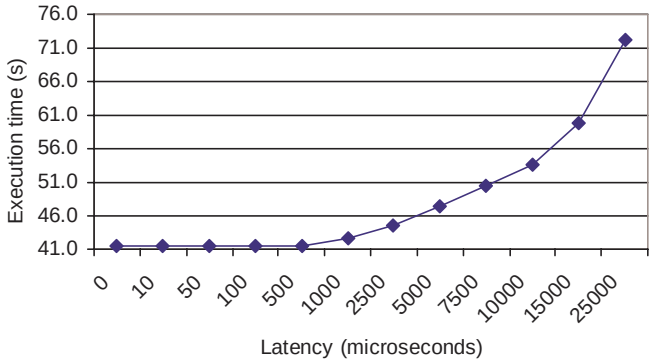
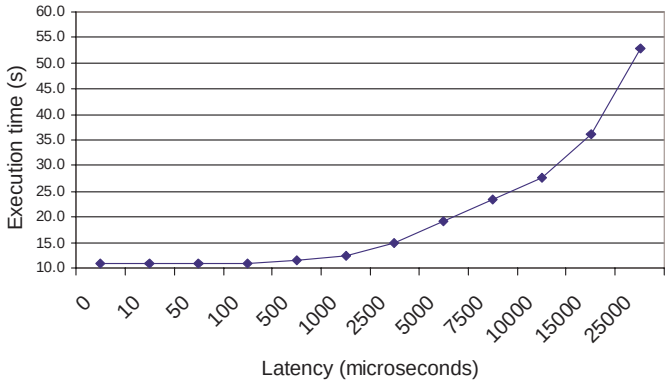


Figure 53. Influence of latency to application time. Application LU.B.32



As a second part of the method, we focus the analysis in the visualization. Figure 54 and Figure 55 show the visualization of the prediction when the latency is set to 10 milliseconds. The first one is comparable to Figure 46, except by the communication intensive area that is larger because all those messages produce latency utilization. The second is comparable to Figure 47, zooming the communication intensive area, and allow the analysis of latency utilization, and its delay effects in all tasks. Furthermore, Figure 55 show in yellow all the occurrences that task number 8 incurs in latency utilization, one for each message.

Figure 54. LU.B.8, 10 ms latency

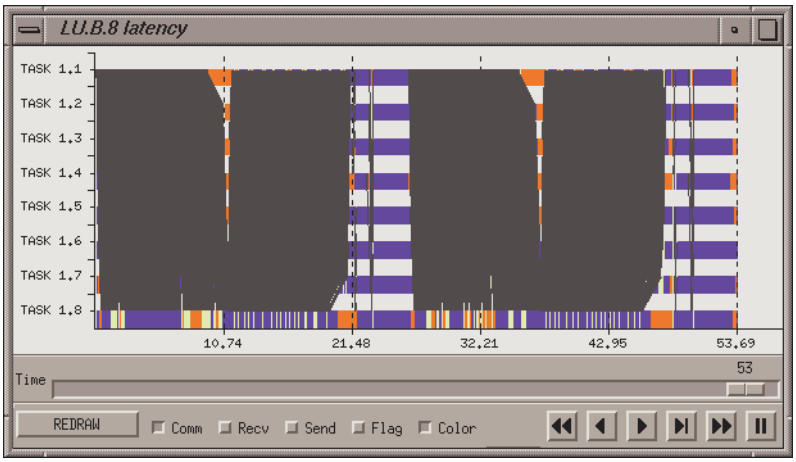
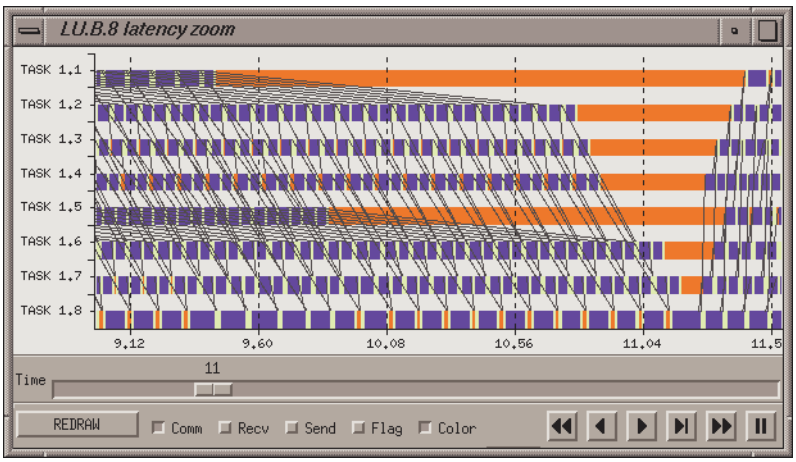
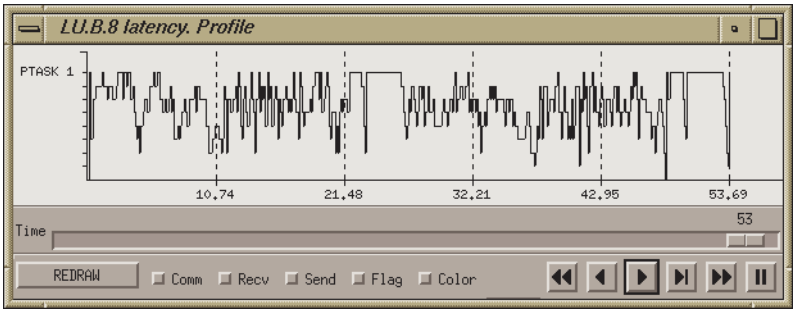


Figure 55. LU.B.8, 10 ms latency, zooming the communication intensive area



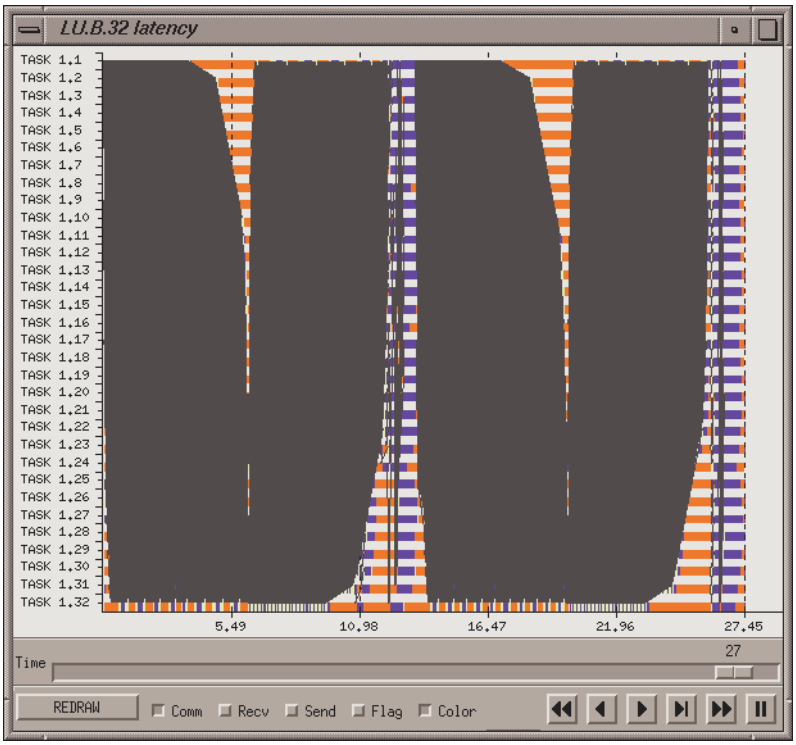
We can compare also the total execution time in Figure 54 and Figure 46. Including latency time in the simulator, the application takes 12.1 extra seconds for finalization. But, a single task is sending and receiving 1010 messages as an average, inducing 10.1 seconds (1010*10 milliseconds) of overhead. The difference (2.1 seconds) is motivated by the different blocking schema produced with the new parameters. The explanation for this new blocking schema is: if there is a penalty for sending a message, the receiver get the message later and thus the total blocking time for this receiver task is greater than in the execution without latency.

Figure 56. LU.B.8, 10 ms latency, application profile



If we compare Figure 56 and Figure 50, total execution time is not the only result that has changed. We can conclude that for high latency values, LU using 8 tasks increases the total execution time and the blocking time. Thus, it can be interesting to group messages and reduce the total number of communications included in this LU implementation.

Figure 57. LU.B.32, 10 ms latency



We also present figures for the same application but running 32 tasks (Figure 57, Figure 58 and Figure 59). Both Gantt diagrams demonstrate the big influence of latency in the application behavior, specifically in the intensive communication area. The blocking time for each time is increased dramatically. Figure 59 completes the analysis, where one can observe that overall performance of the application is reduced dramatically compared to Figure 51. In the current prediction, using

10 milliseconds latency, the average level of parallelism is clearly below the value obtained with ideal network parameters. With this analysis, we can conclude that it is not convenient to execute this application, on 32 tasks, in a system with a high latency network.

Figure 58. LU.B.32, 10 ms latency, zooming the communication intensive area

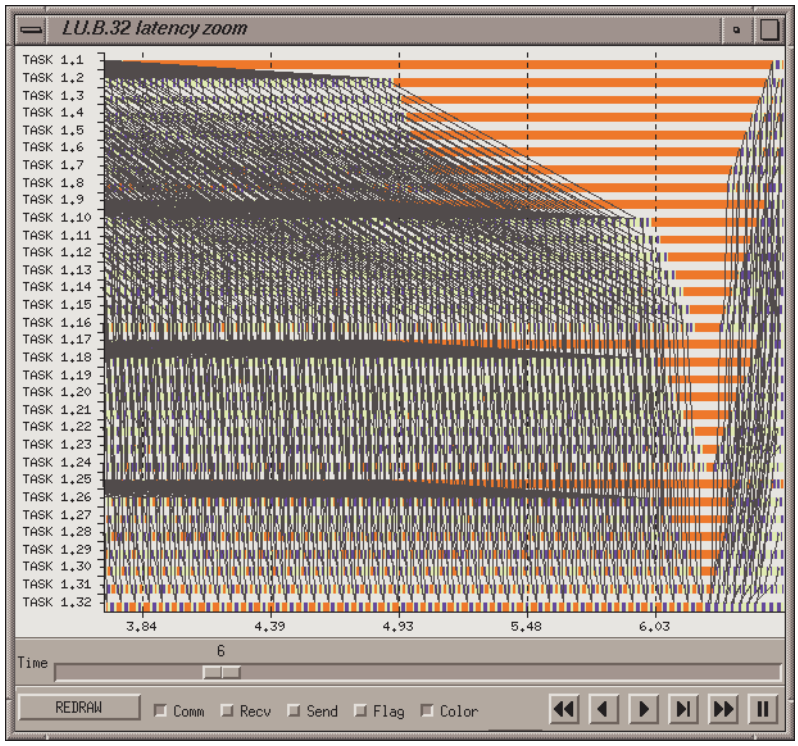
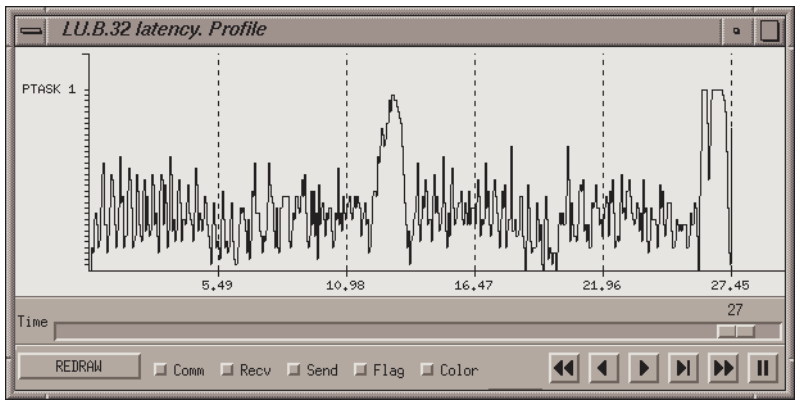


Figure 59. LU.B.32, 10 ms latency, activity summary



To conclude the analysis of the influence of latency in application execution time, we can compare execution time for the different combinations, presented in Table 10. The results shows that in ideal network the speed up was close to ideal

(4), but using a 10 milliseconds latency the speed up is reduced to 2. The reason is that 32 task application uses much more messages (1010 per task versus 657 per task, thus a total of 8080 for LU.B.8 and 21008 for LU.B.32), and each of them requires a latency.

As both applications run the same problem size but using different number of processors, we recommend to use the 8 tasks version when running in a network with big latencies, otherwise the overall performance is reduced.

TABLE 10. Latency influence analysis. LU execution time comparison (in seconds)

	Ideal network	Latency = 10ms
LU.B.8	41.5 s	53.6 s
LU.B.32	10.8 s	27.6 s

7.3 Bandwidth influence analysis

Another important parameter in message passing environments is network bandwidth, because if fixes the delivery time for a message. This delivery time depends on message size. The questions we want to address are the following. Is our application sensible to bandwidth? Do we have to split messages to reduce message size and then speed the delivery of each message?

The method we propose is the same presented in Section 7.2, but using different values for bandwidth instead of different values for latency. The model assumptions are ideal latency (0 microseconds), no links contention and no bus contention.

The first part of the analysis is intended to analyze the influence of different values of bandwidth in application time. Figure 60 and Figure 61 show this influence for LU.B.8 and LU.B.32 applications, respectively. In both examples exists this influence in application behavior, for bandwidth values below 5 Mbytes/second. Although both Figures are similar, it is important to remark that execution time with 0.5 Mbytes per second is 12% slower in LU.B.8, but 28.6% slower for LU.B.32.

Figure 60. Influence of bandwidth in application time. Application LU.B.8

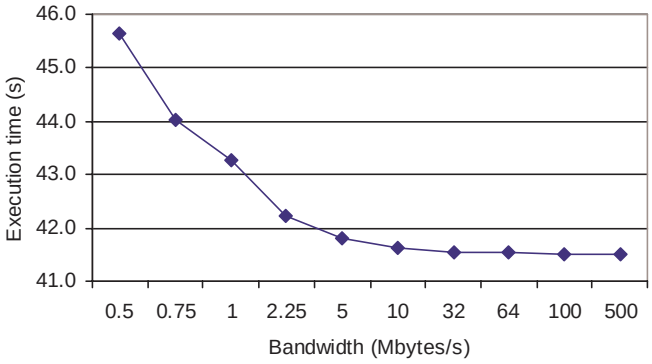
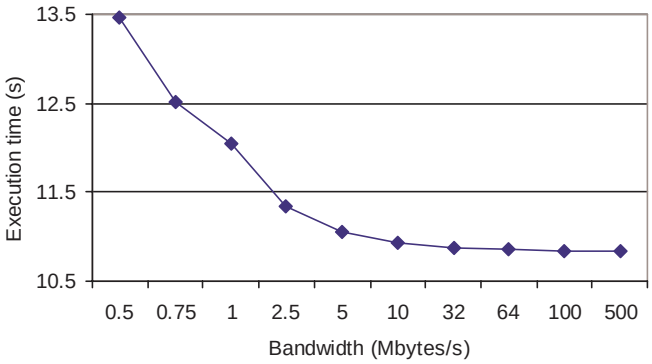


Figure 61. Influence of bandwidth in application time. Application LU.B.32



The second part of the method focus the analysis in the visualization tool. Figure 62 shows the visualization of the prediction when using a 0.7 MBytes/second bandwidth. We have to compare this figure with Figure 46. The difference in both figures is not concentrated in the communication intensive area (as it was in the latency analysis) because these areas really compare similar. But there are some big messages after the communication intensive area (close to second 17 and second 38) that make those figures look different. A significant amount of time is required for delivery because the small used bandwidth and small message size. In Figure 62, we can observe that those big messages produce extra blocking in the receiver task because of the necessary time for delivery (two red arrows point to these areas).

Figure 62. LU.B.8, 0.7 MB/s bandwidth

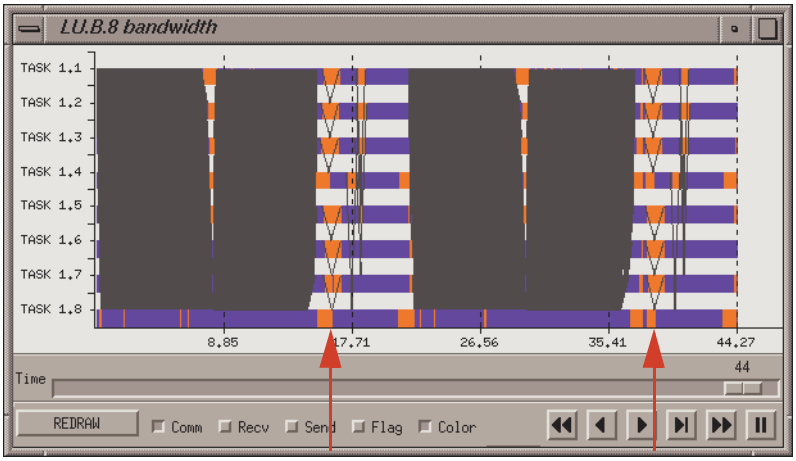


Figure 63 compared to Figure 50 on page 103 shows there are two locations in the execution where the parallelism level does not reach the optimal value. This information, and the presented in Figure 62, shows the influence of big messages in the application execution (notice that arrows in Figure 62 correspond to the location in Figure 63 where there is some loose of performance).

Figure 63. LU.B.8, 0.7 MB/s bandwidth, application profile

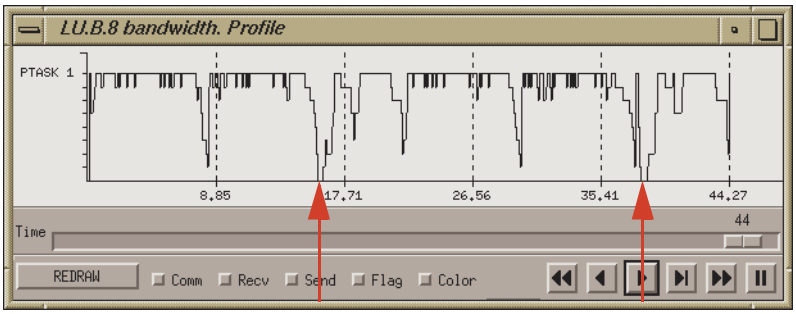


Figure 64 and Figure 65 refer to LU application but using 32 tasks with a low bandwidth network. These figures are comparable to Figure 48 on page 102 and Figure 51 on page 104, respectively. Figure 64 shows clearly all the blocking times produced by slow message delivery, tasks are ready to get the message but this is travelling through the network. We remark these blocking times with the red arrows. Figure 65 gives information about the loose of performance of this application when running in an environment with small bandwidth. The profile presented in Figure 51 on page 104 corresponds to an application reasonably load balance, but once we execute the same application in a different environment (a real one, with an small network bandwidth) it does not behave as well as before.

Figure 64. LU.B.32, 0.7 MB/s bandwidth

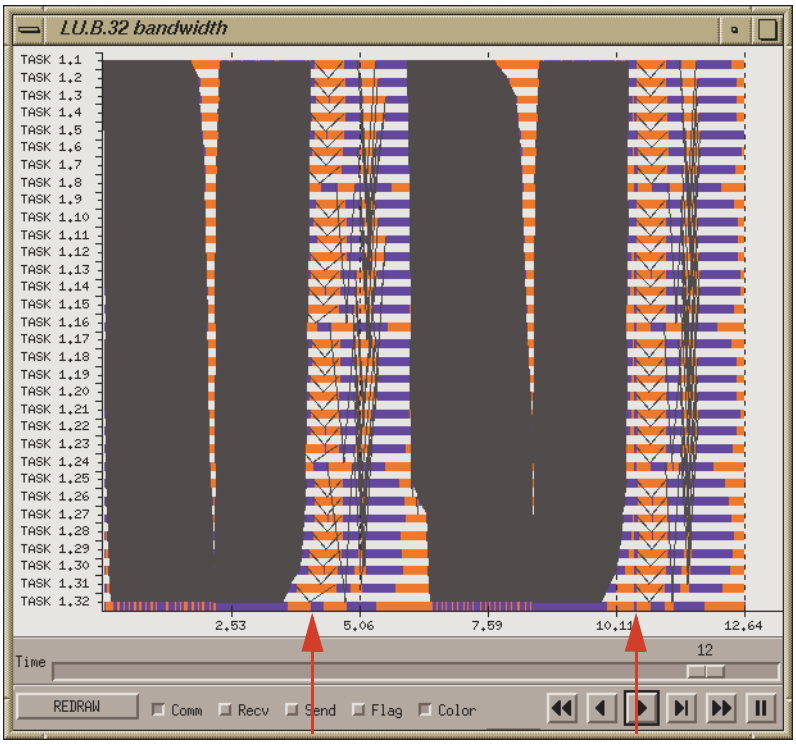
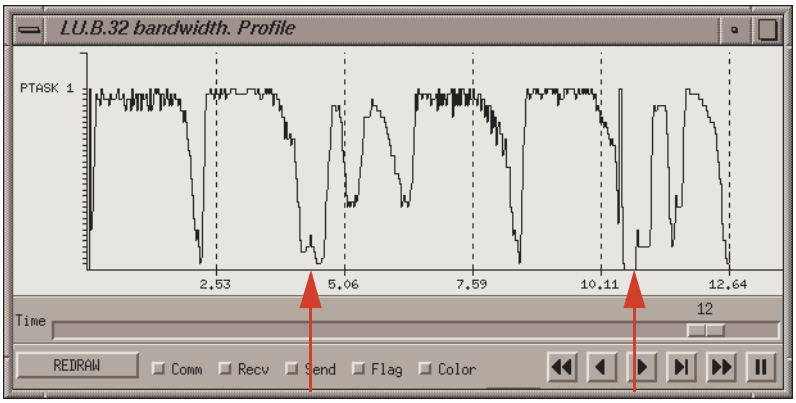


Figure 65. LU.B.32, 0.7 MB/s bandwidth, activity summary



It is evident, at it is shown using Dimemas, that bandwidth influences application time, not only for messages being delayed in delivery, but also in the blocking time waiting for messages. If possible, it will be convenient for this application to perform some extra computation at that time, as it is not only important to balance all tasks within an application but also to take into account the execution environment and the communication dependence chain.

7.4 Analysis when having resource contention

In previous sections, we have analyzed the application itself and the influence that network parameters have. An extended study of the influence of network resources can provide a better understanding of application time variation in real execution environments. With this information, the user shall be able to define/select a proper execution environment for its application, or to define some architectural parameters to limit the application time. This is the last step of the methodology we propose for analysis of message passing applications.

The system model we use to perform this analysis has the following network parameters, and we call it ‘contention network’:

- Latency: 10 milliseconds
- Bandwidth: 0.7 MBytes/second
- Bus connectivity: single bus
- Node to network connection: 1 full duplex link

Figure 66, Figure 67, and Figure 68 compares to Figure 46 on page 100, Figure 47 on page 101, and Figure 50 on page 103, respectively. The bigger differences are located in the non-intensive communication area, because larger messages (refer to Section 7.3) require the resources more time (bus and links), thus all other messages are delayed until the resources become available. The initial time for the second iteration, using ideal network is close to second 20 but in current analysis it starts at second 33.

Figure 66. LU.B.8, contention network

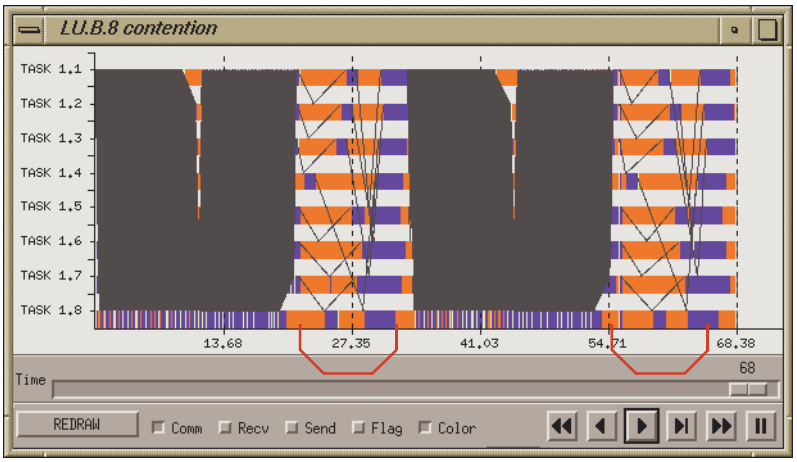


Figure 67. LU.B.8, contention network, zooming the communication intensive area

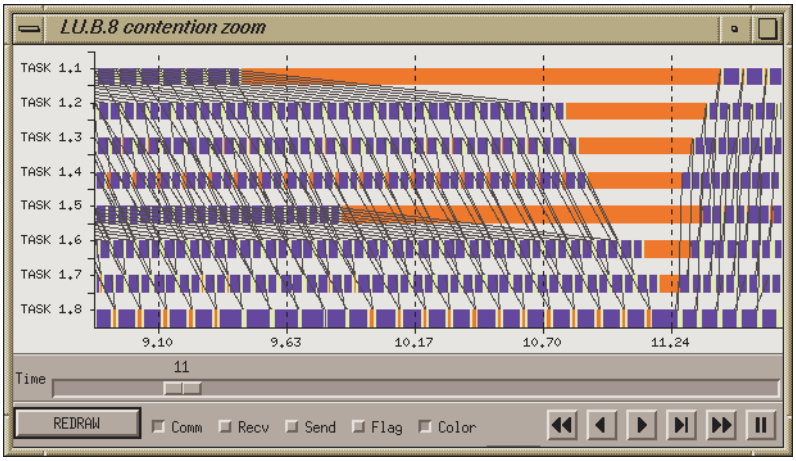


Figure 68. LU.B.8, contention network, application profile

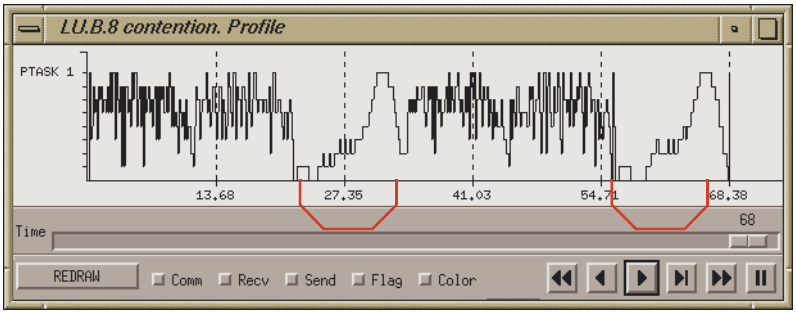
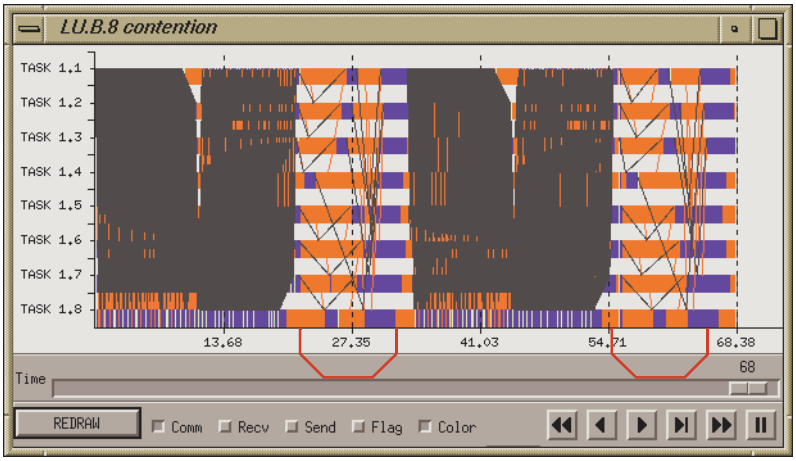


Figure 69 displays the sample application but in this case red lines represent the physical communication. We set up the analysis for a contention-based network, with a single bus; thus only one message can be delivered simultaneously. The graphical representation allows us to observe how the resource (the bus) is shared among all tasks. Of course, this is much clear in the area with big messages, because those require more communication time.

Figure 69. LU.B.8, contention network, physical and logical communication



The analysis facilities provided by Paraver, allow the analysis of different options. For example, Figure 70 displays the instantaneous bus queue length, that is the number of messages blocked because of bus contention. The maximum value is 5, produced in the non-communication intensive area. This is motivated by larger messages that use the bus more time than shorter ones, which are located in the communication intensive area.

Figure 70. LU.B.8, contention network, bus queue length

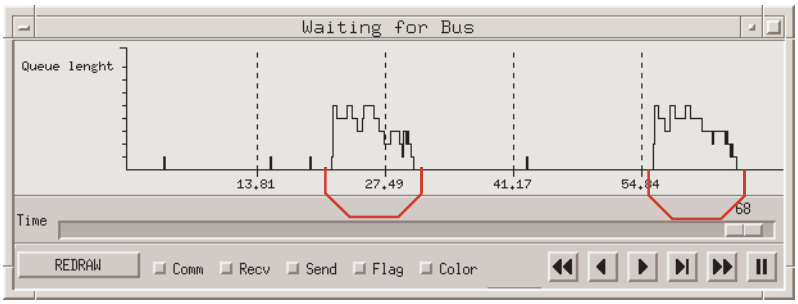
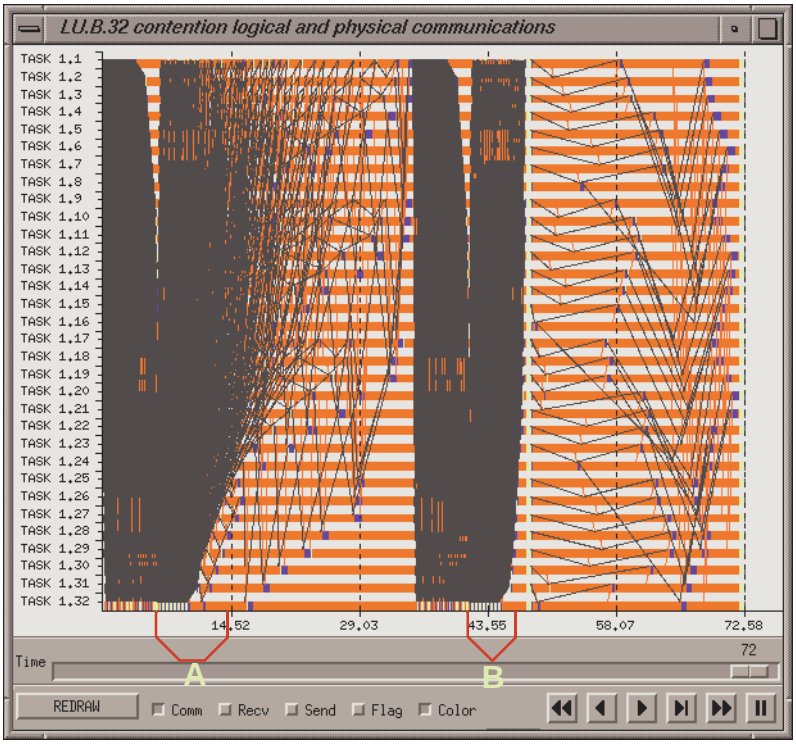


Figure 71. LU, class B, 32 tasks, contention network, physical and logical communication

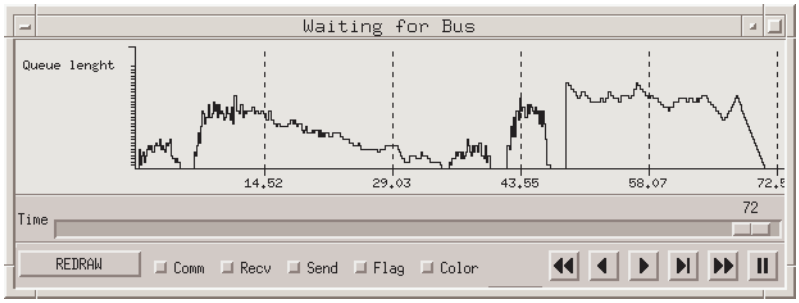


Same information relative to the 32 tasks case is presented in Figure 71 and Figure 72. At a first glance comparing to Figure 57 on page 107, but using contention network and visualization of logical and physical communication. Orange color, block time, is the larger one for all tasks because of resource contention.

It is also important to notice the different order of message delivery between the two iterations. Initially, both iterations are identical and one can expect to obtain the same message ordering. In Figure 72, we remark with letters A and B the final area of the two identical iterations, but they do really look completely different. Dimemas uses FCFS (First Come First Served) for resource assignment. Any small variation in the execution of the application is present in the tracefile, and Dimemas assign resources upon this timing, which may produce big delays because of resource contention. This also happens in real execution, where resources are assigned in demand bases but not taking into account the communication chain of the application.

Bus contention produces the real problem for this application (Figure 72). Compared to the execution with 8 tasks, we obtain a value close to 10 for the average queue length. Although 10/32 is relatively smaller than 4/8, the difference is that for 32 tasks, the bus is producing contention for nearly all execution time.

Figure 72. LU.B.32, contention network, bus queue length



7.5 Conclusions

In this chapter, we have presented a methodology to analyze application performance, to locate those factors that limit the performance, and to predict the application behavior in different architecture configurations. This methodology is based in the utilization of a simulator, Dimemas, and a visualization tool, Paraver. Using these tools, the user gets a better understanding of the parallel application and the influence of the architectural parameters in its performance.

This methodology provides answers to the following questions: Is my application having load balance problems? Is there any important precedence in the communication pattern of my application? Is my application sensible to communication parameters? Is my application sensitive to latency? Shall I group communications (if possible)? Do my application require a fast network to obtain good performance? Is my application sensible to network contention or to network resources?

To conclude the analysis of this chapter we can compare the application time and processor utilization of both applications (LU.B.32 and LU.B.8) in all the scenarios we have defined in our methodology.

TABLE 11. Comparison for different system parameters for LU, class B.

	Network model	Execution time	Processor utilization	Slowdown	Speedup
8 tasks	Ideal	41.513	94.72%		
	Latency	53.689	73.36%	0.77	
	Bandwidth	44.268	88.88%	0.94	
	Contention	68.384	58.00%	0.61	
32 tasks	Ideal	10.848	88.54%		3.83
	Latency	27.613	35.03%	0.39	1.94
	Bandwidth	12.644	76.14%	0.86	3.50
	Contention	72.584	13.68%	0.15	0.94

The analysis of Table 11 is completely related to this application and can not be extended to other applications, because it depends on the communication pattern,

the total amount of messages, the size of these messages and the load balance of the application.

First column in Table 11 is the total application time. The second one is the average processor utilization of all the tasks. Third column is the slowdown of the application, computed as the ratio of the application time on ideal network and the application time in the current scenario. It provides information of the loss of performance because of the network environment. Last column is the application speed up, computed as the ratio of the application time of the 8 task application and the 32 task application, when the analysis is performed in the same scenario.

Latency has big influence in both examples (8 and 32 tasks), because in both cases there is a lot of messages, but for the second one (LU.B.32) the influence is much higher (the slowdown is 0.39) because there are much more messages than in LU.B.8.

Bandwidth does not have the same influence as latency. For 8 tasks, the influence is important because message size is quite significant, and the delivery time is enlarged. But using 32 tasks, each message size is smaller and the influence is not as important as in the case of latency.

When bus contention is considered together with the 32 tasks cases provides really bad results because of the combination of high latency, poor bandwidth and bus contention. Slowdown is 0.15 for LU.B.32. For LU.B.8, the slowdown is significant but not that big as for LU.B.32.

The speedup analysis shows that the application obtains good results in ideal network scenario, but as we include the new scenarios, the speedup is reduced dramatically. We must mention the 0.94 speedup for contention network, that it means that the application is slowest than LU.B.8, using more processors and solving the same problem.

CHAPTER 8

Analyzing the influence of process scheduling

The origin of Dimemas was the analysis of scheduling policies in SMP systems, and this chapter will focus in this topic. Although it was the former objective, at this moment it is a smaller part of the work, but it remains as a possible extension of this work.

Early computers, those that work with batch systems, received a sequence of jobs/programs to execute in order (batch order). The inconvenient was the loose of performance because processor is often idle. The problem was the speed of I/O devices compared to the processor speed. Multiprogrammed batch systems are the solution to this problem, where several jobs are ready for computation. The operating system picks up a job and begin to execute it. Eventually, the job may wait for some I/O. In a non-multiprogrammed system; the processor would sit idle. In a multiprogrammed system, the operating system switches to and executes a new job. When that job needs to wait, the processor switches to another job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as there is always some job to execute, the CPU will never be idle. The extension to multiprogramming is time-sharing. Multiple jobs are executed by the processor switching between them, but the switches occur so frequently that the users interact with each program while it is running.

Parallel systems allow two different programming models: shared memory and message passing. Space partitioning was the first mechanism provided to share a parallel system. Each application is mapped in a subset of processors and uses it in dedicated bases. This mechanism allows executing concurrently in the same system several applications, but not using all resources. It can be compared to batch systems. Logical extension is to allow time partitioning (comparable to time-sharing). Each processor can switch to different jobs, and a subset of those jobs forms an application. The same process scheduler was applied to these new systems, and some problems arise. The most important one is the problem of the time spent acquiring access to a critical region, because the job handling the right may not be running at that time and will no free it until the processor switch it in. In this case, the requesting job can be switched out because it can be comparable to an I/O block. Gang scheduling is the solution proposed to this problem, ensuring all jobs

of an application are switched on at a time. But this method is reasonable applicable to shared memory system, but not in distributed systems because the synchronization of all involved scheduler systems is time consuming.

Our work focuses in the analysis of different scheduling policies for distributed memory machine systems, using the message passing programming model. For this evaluation, we will present three different case studies. These studies show the evolution of our work, starting from synthetic workload and simple scheduling policies, to more complicated workload (using mixtures of NAS benchmarks) and some specialized schedulers.

The remaining part of this chapter is organized in a section for each of the three different case studies, and finally some conclusions on the work performed in processor scheduling.

8.1 Short term scheduling and dependence chain

The main objective of this work on scheduling policies has been to evaluate the efficiency of short term scheduling policies in system and application performance. A parallel machine is a very complicated system, where the final performance depends on a large numbers of parameters. Not only there are many parameters, but the interaction or coupling between them is very intricate and may have important effects in performance. The approach we follow in this section, based on trace driven simulations, tries to isolate specific aspects that are then evaluated in a more controlled way than the execution on a real system. By using a simple model of the architecture we concentrate on some aspects which we consider of high relevance. Our interest concentrates on the relationship between short term scheduling and the dependence chain (synchronization) in the application.

8.1.1 Workload

Even with a simple model of the architecture and scheduling policies, there is a high dependence of the results on the application characteristics. This was expected and became apparent in some preliminary experiments. It was thus decided to carry out a first set of experiments on very simple workloads consisting of synthetic applications. In this way, the application behavior is also under control, and the effect of the scheduling policies can be better analyzed.

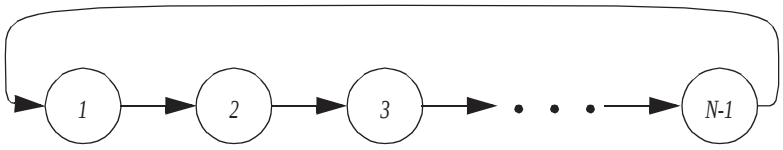
Traces from real programs will then be used to analyze the efficiency of the different schedulers on real applications. The objective here is to find out whether the effects identified on synthetic workloads appear also on real workloads and what is their relative importance.

8.1.1.1 Synthetic traces

Many scientific applications on message passing oriented machines use a predefined communication scheme. Frequently encountered schemes are line, ring, binary tree, torus,...

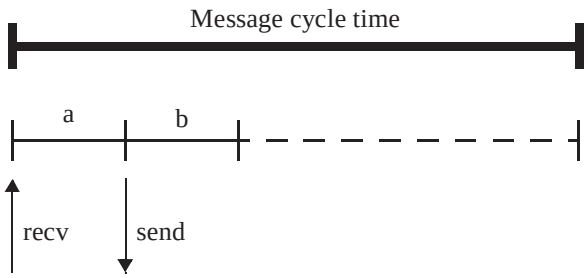
For our synthetic workload we have chosen a ring based communication scheme, on Figure 73: task *i* sends messages to task *(i + 1)%N*, where *N* is the number of tasks of the application. This communication scheme is widely used and is simple enough to understand the influence of communication to application and system performance.

Figure 73. Ring communication



The computation and communication sequence followed by all the processes within the synthetic application is shown in Figure 74. Each process follows an infinite loop with four phases: it receives a message from the previous process, computes for a while, sends a message to the next process in the ring and computes for some additional time. This pattern may correspond to the behavior of an iterative algorithm. To start up the ring, processor 1 sends a first message without receiving any.

Figure 74. Task compute and communication

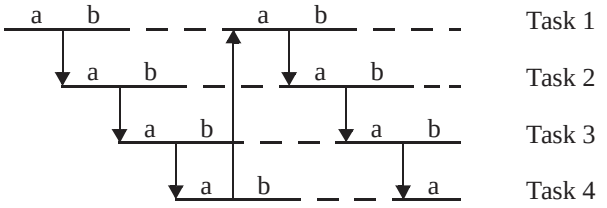


These synthetic applications can be characterized by three parameters:

1. Total number of tasks within the application, *N*.
2. Execution time between a message is received and next message is sent, *a*.
3. Processor utilization percentage (demand) obtained from a single task if running exclusively on a single processor, *d*%.

We will call message cycle time, *mct*, for such an application as the time between two consecutive messages arrive to the same task. The message cycle time will be minimum, *m_mct*, when each task is run on a different processor, the communication assumes null latency and infinite bandwidth. Under these conditions, Figure 75 shows an example, with *N*= 4 tasks, *d*= 50% of utilization and *a* microseconds between the reception of a message and the sent of a new one.

Figure 75. 4 tasks



Computational time spent after sending the message, b, can be computed as:

$$b = a \times \left(\frac{d \times N}{100} - 1 \right)$$

In this synthetic load, each task performs a given computation (between receive and send) which is in the critical path of the application, and the rest of the computation until the next receive is used to tune the desired CPU consumption of the application. This model limit the computation times between receive and send to one that achieves 100% utilization. We do not consider for the moment larger computation times after the send as this would put all the code in the critical path and we are interested in analyzing the interaction between applications with parts in the critical path and parts outside it.

As seen is Figure 75, minimum message cycle time for synthetic applications can be easily computed as:

$$\text{m\_mct} = a \times N = \frac{a + b}{\left(\frac{d}{100} \right)}$$

For a given CPU demand *d*, an application can have a small or large *a*. In the first case, the execution consists of many small CPU burst and very frequent communications. In the second one, communications are not very frequent and CPU bursts are large. We will refer to the first case as a more critical application as the critical path of a single iteration is much shorter. Blocking for a given amount of time one communication of the first application is relatively more important than doing it to the second application.

8.1.1.2 Real program traces

Seven traces have been used for the measurement of the effect of the scheduling policies on real applications. Three of them come from the porting effort described in APPARC OpSa4 [76]. The other four come from solvers developed by UPC within ESPRIT project IDENTIFY. All of them have been generated for 8 processors.

From the APPARC, the three traces are:

1. **bm6**

   Trace of 10 iterations of the CG solver for a 35937 unknowns problem. The execution time of this trace alone is 11.7 seconds and the utilization is very low (U = 15%).

2. **femsolv**

   The trace includes the whole solver of this application. The problem size is 900. The execution time of the trace is 4.6 seconds and although it is quite parallel it has a certain amount of unbalance. The utilization of processor 1 is 66% while the other seven achieve 97%.

3. **parfem**

   This trace corresponds to the parallel matrix assembly and the first 10 iterations of the CG solver applied to a problem size of 17632 unknowns. The total duration if run alone is 6.4 seconds. The first part of the trace (6 s), which corresponds to the matrix assembly, is totally parallel although there is a certain amount of unbalance (in the last 0.7 seconds). The solution part is much shorter (0.3 seconds) than the assembly part, has a substantial amount of communication and a good average parallelism (U = 79%).

The IDENTIFY project consisted in the parallelization of a CFD code. The solvers developed are based on domain decomposition. The four traces used correspond to the solution of a problem of 48384 unknowns with a BiCGS iterative solver. All of them contain 10 iterations. The traces are:

4. **ilu bisect**

   The solver uses an ILU preconditioner and the mesh was partitioned with a geometric bisection algorithm. The execution time of this trace alone is 14.7 s. with an utilization per processor of 55%. The amount of computation is well balanced among processors. The block data distribution used and the solution of triangular systems required by the preconditioner produces the low processor utilization.

5. **dia bisect**

   The same partitioning method but a diagonal preconditioner is used. The preconditioner is numerically worse than the ILU preconditioner, but requires less communication and computation per iteration and is more parallel. The total duration of the trace is 3.2 s. and the average utilization 96% with very good load balance.

6. **ilu malone**

   The same solver as in the first case but the mesh is partitioned with an spectral method. The total duration of the trace is 12.2 s. with an average utilization of 65.5%, although there is a certain level of unbalance between processors due to the partitioning of the mesh.

7. **dia malone**

   As in the previous case but using diagonal preconditioner. The execution time of this trace alone is 3.7 s. achieving an average utilization of 90% due to the unbalance originated by the partitioning.

8.1.2 System model

Our first objective has been to concentrate on the interaction of fine grain scheduling and message passing applications. To this end we have simplified the model used within the simulator and the scheduling policies to those described in the following sections.

The system we are simulating corresponds to a closed network with just one server. The server represents the ensemble of processors that run the parallel applications. Each customer represents one application. As in a typical closed queuing model, we are interested in looking at the effect of the different policies on performance indices such as throughput or fairness, which analyze the system behavior when stressed by the tight competition between applications. When one application finishes, other one of the same type is started.

Our model has a single server representing the computational subsystem, as we do not consider applications to perform any I/O operations. This is another issue for future consideration. Overlapping computation and I/O is the reason while multiprogramming was introduced in sequential systems. Modeling some type of I/O in our experiments would certainly favor our hypothesis on the usefulness of time sharing in parallel systems. Nevertheless, we are for the moment interested in the overlaps achievable between computation and communication within the processing part of the parallel applications themselves.

An open issue for an immediate future is to consider an open system where applications of different types arrive to the system with a specific time distribution. In this environment, our approach of fine grain sharing will certainly lead to better average turnaround time than the typical batch oriented system. Related works [70] that analyze different space partitioning approaches recommend dynamic policies where short jobs arriving to the system are allocated some processors as soon as possible. It is an open question how a fine grain time sharing can achieve similar effects without the redistribution cost.

8.1.2.1 Communication

Although Dimemas allows us to simulate a wide range of message passing computer architectures, we have limited our experiments to a few ones. This reduction is proposed since, at the moment, we are interested in understanding applications behavior and low level scheduling policies, and not in the influence of the message passing mechanism nor in the network connectivity. We have decided to focus on the following communication characteristics:

- No rendezvous

  A sender thread never waits for its partner, just sends the message and the communication mechanism stores it, on the destination processor, until the receiver requests it.

- No latency time

  When a thread calls to the communication routine, data transmission begins without any latency overhead. The thread can also continue immediately, not incurring any cost in copying the message to or from system buffers.

- Infinite bandwidth

  Transmission is instantaneous no matter the length of the message. A message arrives to its destination node (processor) at the same time it is sent.

- Asynchronous

  The sender thread doesn't have to wait until the communication mechanism delivers the message. It is not relevant given the above two settings.

- Full processor connectivity

  There is no routing delay nor conflicts in the network, neither due to the number of links neither due to the number of buses.

No rendezvous and asynchronous communication have been chosen because this is the mechanism most commonly used in message passing libraries (pvm3, 3P Parlib,...).

Other assumptions were done in order to reduce the set of parameters that influence the results. The instantaneous communication assumptions are non realistic as in many cases the communication time is important and is certainly one of the most important causes of low efficiency of an application. For our experiments, this assumption is going to produce pessimistic results. In fact, one of the sources of system throughput improvement that multiprogramming offers is the overlap between communication and computation. We are nevertheless interested in the effects of the scheduling policies on the applications only considering their load unbalances and dependence relationships. We consider that the delays caused by those two features are key to devising policies which lead to an efficient system.

8.1.2.2 Scheduling policies

We will consider in this study short term scheduling policies that are applied locally at each node without global coordination. We have chosen to study the following scheduling policies/algorithms:

1. FCFS, First Come First Serve (FIFO, First In First Out)

   Ready threads are kept in a FIFO queue. When the running thread requests a communication which requires blocking (message not yet received), it blocks and the first thread in the ready queue is dispatched. When the communication ends, the thread is put at the end of the ready queue.

2. Round robin, RR

   It is similar to FCFS scheduling, but preemption is added to switch between threads in a circular way. The quantum assigned to a dispatched thread is 1 ms. in our experiments. In case a thread blocks, it looses the rest of its quantum, and a new thread is scheduled from the ready queue. When a thread unblocks it is queued at the end of the ready queue.

3. Fixed priority, PF

   A priority is associated to each Ptask, all the threads within the same Ptask have the same priority, and the processor is allocated to the thread with higher priority. Between threads of equal priority, the previously described round robin algorithm is applied. When a thread unblocks at the end of a communication it is inserted at the end of the appropriate queue. No preemption is applied.

4. Preemptive fixed priority, PP

   This algorithm is like the previous one, but incorporates preemption. When a thread of higher priority than the running one appears on the ready queue

(unblocks from a communication), the scheduler is activated and the processor is assigned to the higher priority thread.

5. Critical path, CP

   We propose this algorithm trying to find a good processor utilization and small delay in application response time. The basic idea is to favor threads that are in the critical path of their application. It's based on preemptive dynamic priorities. All applications have the same basic priority, but priority is increased when a thread is going to get into the critical path of the application if it where alone. The priority is reduced when leaving the critical path.

   Two variations are used in the case of equal priorities. One is to apply FIFO (CP FIFO) and the other is round robin (CP RR).

   This algorithm is rather ideal to experiment with the system behavior. It is implemented in our simulation environment by including records in the trace file that specify the priority of the thread. For the synthetic traces these records are easily introduced by the trace generator. For real traces, these records are introduced by one of the analysis modules integrated into Paraver.

   This algorithm uses the critical path of the application when run alone. Observe that when the application is multiprogrammed with other one, its critical path may change as a result of the scheduling itself. We don't make efforts currently to dynamically estimate which are the computations effectively in the critical path.

For scheduling policies with priorities, we will use the notation PF_1 (for Priority Fixed) when application 1 has higher priority than application 2, and PF_2 for the opposite situation.

8.1.2.3 Performance indices

The figures we will present for each experiment will correspond to one of the following:

1. Execution time, *T*

   Correspond to the message cycle time for the synthetic applications and the total application execution time for the real applications.

   We will refer as T^{alone} to the execution time of one application when executed alone, and T^{mp} when executed under multiprogramming. The generic *mp* superscript can be substituted by the name of the scheduling policy. The superscripts will be dropped when not misleading.

   In the multiprogrammed case, a subindex *i* will be used to specify the application number. In the case of policies that give priority to a specific application, other subindex will be used before the previous one indicating the favored application.

   This subindex notation applies also to the other performance indices.

2. Throughput, *Th*

   Number of jobs finished per time unit.

   For the synthetic loads, a job will correspond to a single iteration and the time unit will be 1 second.

In the analysis we can refer to three different throughput presentations: individual application throughput, system throughput (sum of the individual throughputs) and throughput ratios between a multiprogrammed and a batch execution for the same application workload.

3. Slowdown, *S*

   If several applications are going to be executed concurrently each of them should expect a certain slowdown compared to its execution alone. We will use a slowdown index defined as:

$$S = \frac{T^{mp}}{T^{alone}}$$

   This value will always be $S>=1$ and represents a multiplicative factor of the performance obtained for one application when another one is multiprogrammed with it.

4. Slowdown ratio, *SR*

   This value shows the fairness degree of two applications and is defined as:

$$SR = \frac{max(S_i)}{min(S_i)}$$

   The scheduling will be fair when slowdown ratio is equal to one. Big values of *SR* mean that one of the applications is being disturbed by the other one more than what a fair share of the resources would expect. In this equation, we employ the slowdown (S_i) of all the applications involved in the current analysis.

8.1.3 Experiments

In this section we will describe the experiments carried out and the objective for each of them.

8.1.3.1 Synthetic applications

We introduce the following notation to specify a synthetic application and the configuration with which it is run:

$$(a,d\%)[p_1,p_2, ..., p_n]$$

where:

- *a*, is the compute time (in microseconds) between a message is received for a single thread and the next send is requested.
- *d*, is the percentage of utilization if each thread was run alone in one processor. It represents the computation demand of the application.
- $[p_1, p_2,..., p_n]$ is the mapping for this Ptask. Current Ptask has *n* tasks and Task *i* is mapped into processor number p_i.

Table 12 and Table 13 enumerate the experiments to be carried out with synthetic loads. There are two basic groups.

TABLE 12. Multiprogramming experiments

First application	Second application
(4200, X%) [1, 2, 3, 4, 5, 6, 7, 8]	(4200, X%) [1, 2, 3, 4, 5, 6, 7, 8]
(4200, 60%) [1, 2, 3, 4, 5, 6, 7, 8]	(4200, X%) [1, 2, 3, 4, 5, 6, 7, 8]
(4200, 60%) [1, 2, 3, 4, 5, 6, 7, 8]	(X, 60%) [1, 2, 3, 4, 5, 6, 7, 8]

The first one (Table 12) corresponds to the multiprogrammed execution of two applications with the same or different characteristics. In this table, X stands for the parameter we will modify to study its influence in the whole performance. In the first experiment, both applications are the same and the parameter that varies is the CPU requirement of each of them. The other two experiments fix the first application and the variation is introduced in the second application. First (second experiment) the CPU demand of the second application is varied keeping the critical path equal in both applications. Then (third experiment) the critical path of the second application is varied while keeping the CPU demand equal in both applications.

The objective is to analyze the interaction of the scheduling policies with the CPU demands of applications that have synchronization and with the differences in critical paths of the applications.

A second group of experiments is enumerated in Table 13. It corresponds to a single application that is folded by putting two tasks in the same processor. Two different mappings are considered: non interleaved (exps. 1 and 2) and interleaved (exps. 3 and 4). For each of them the CPU demand of each task is varied in one experiment and the critical path in other.

TABLE 13. Application folding experiments

Application	Mapping
(4200, X%)	[1, 1, 2, 2, 3, 3, 4, 4]
(X, 60%)	[1, 1, 2, 2, 3, 3, 4, 4]
(4200, X%)	[1, 2, 3, 4, 1, 2, 3, 4]
(X, 60%)	[1, 2, 3, 4, 1, 2, 3, 4]

The objective is check the influence of the scheduling when there is dependence between the tasks competing for the same processor. These experiments are also interesting to evaluate the effect on one application in case it is started with a large number of processors and has to be partially migrated to fit into a smaller processor set. This situation would naturally arise in cases where the scheduling policy of the operating system dynamically partitions the processors among applications.

8.1.3.2 Real applications

The final group of experiments evaluated is related to multiprogramming real applications. In this group we have evaluated the interaction between the real

applications described in Section 8.1.1.2 with each other. We also evaluate the effect of folding one such application.

8.1.4 Results

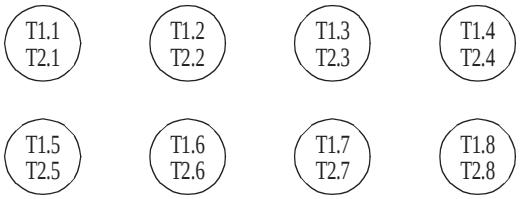
In this section we will present the results obtained for the different experiments. The data in a tabulated form can be found in [76]. These tables can be useful for some fine grain comparisons.

8.1.4.1 Multiprogrammed synthetic applications

2 x (4200, X%) [1, 2, 3, 4, 5, 6, 7, 8]

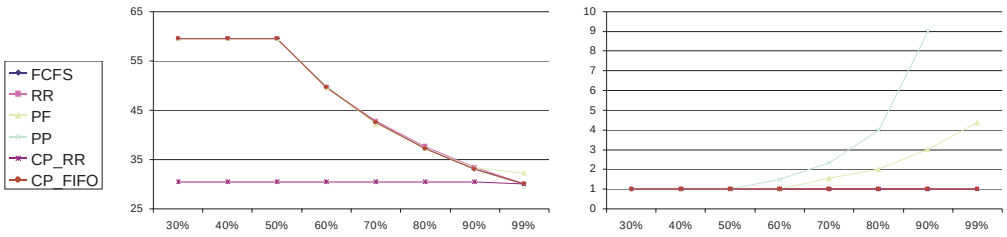
This experiment tries to understand the applications behavior when running two applications concurrently with different scheduler algorithms. In Figure 76 the mapping of tasks to processors is presented: task number *i* is mapped on processor number *i* for both applications.

Figure 76. Mapping 2x[1, 2, 3, 4, 5, 6, 7, 8]



A first effect we wanted to analyze is the synchronization between applications caused by sharing processors and the ability of the scheduling policies to desynchronize the applications. If we observe the behavior of our synthetic application in time it is apparent that the active processors is a set of contiguous processors which shifts along the ring. When loading two concurrent applications there is the possibility that the active sets of both applications overlap in time during the whole execution which would lead to some processors having active threads from both applications and some processors idling.

Figure 77. 2 x (4200, X%) [1, 2, 3, 4, 5, 6, 7, 8]. System throughput (left) and slowdown ratio (right)



The main effect observed in Figure 77 is that all policies except CP_RR achieve maximum throughput when *d*<=50%. This means that CP_RR does not desynchro-

nize the applications while all the other scheduling policies do (observe that both applications start synchronized).

If we look at the Paraver plots, it is apparent that even if all other policies achieve the desynchronization, the time taken by each of them is different. In the cases where there is no preemption in the experiment (FIFO, FP, PP, CP_FIFO) the desynchronization is achieved very rapidly. With RR both applications start competing for the same set of processors but if one application sends a message to the next processor, its processing in the receiving processor will start without competition for a while (until the other application sends the message). This effect is additive and as a result, one of the applications is able to take the lead and put its set of active processors ahead of the set of the other application. The time to achieve desynchronization is thus dependent on the quantum compared to the critical path (parameter *a* in this trace).

CP_RR makes the effect described of one application obtaining a small non additive. In fact, when one application sends a message it leaves the critical section in that processor and thus the other application gets priority and immediately sends its message.

Figure 77 shows how important this synchronization can be. For our two application system it halves the system throughput. The experiment shows that on a macroscopic level most of the policies tend to desynchronize applications, but on a microscopic level, very fine grain sharing is not recommended. Quantum should not be much lower than the computation time between communications in parts of the application which are close to the critical path.

Slowdown ratio is also presented in Figure 77. Fixed priority algorithms are naturally unfair when the needs of both applications are higher than the available resources (d <= 50%). The higher the demand of both applications is more unfairness they show. Nevertheless, this factor is not infinity (as would be in the case of a single processor with two sequential applications) because the inability on one application to use all the resources in the system gives some chances to the application with less priority. In this respect there is a great difference between PF and PP. The fact of PF not being preemptive guarantees that the application with less priority is able to progress until it blocks each time it gets the CPU.

This result shows the possible interest of medium term schedulers based on PF in which the priority of each application is periodically modified to achieve long term fairness as in share schedulers.

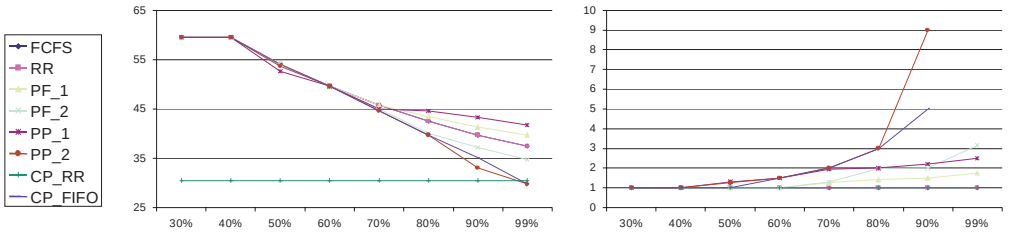
All other policies achieve a perfectly fair behavior, although as seen before there may be a great difference between them in system throughput.

(4200, 60%) [1, 2, 3, 4, 5, 6, 7, 8] and (4200, X%) [1, 2, 3, 4, 5, 6, 7, 8]

In this experiment, the first application is fixed (4200, 60%) and for the second one, the CPU demand is variable (4200, X%). Both applications have the same critical path and the mapping is the same as in the previous section.

Similarly to the previous experiment, the FIFO and RR policies have a perfect behavior in terms of fairness and very good in terms of throughput. The behavior is besides that identical for both of them.

Figure 78. (4200, 60%) [1, 2, 3, 4, 5, 6, 7, 8] and (4200, X%) [1, 2, 3, 4, 5, 6, 7, 8]. System throughput (left) and slowdown ratio (right)



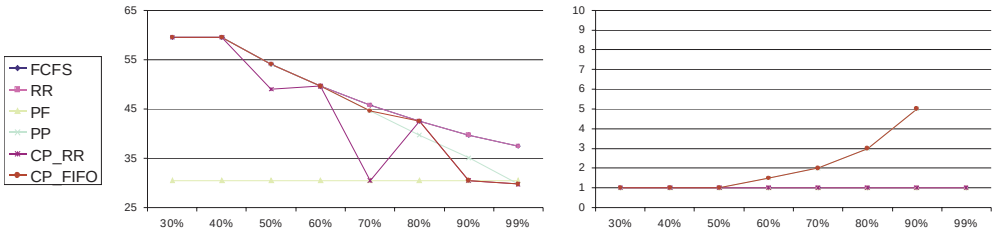
The same synchronization effect described for the previous experiment appears in Figure 78 for this case. The CP_RR algorithm obtains a constant system throughput equal to half the throughput achievable when there are enough resources (d_2<=40% in this case) for both applications if the scheduling policy achieves desynchronization. For CP_RR both applications always overlap critical path execution, increasing the minimum cycle time for those applications.

In the case where competition for resources is unavoidable (d_2>=40%) the comparison of priority policies shows that, in general, it is better to give priority to applications with less resource demands. This results also in a more fair policy. Even if the trend is valid for both PF and PP, the effect is stronger for PP. Between them, PF is more fair than PP and in the cases of giving preference to the less demanding application it also leads to better system throughput.

The behavior of the CP_FIFO policy is marginally better than the one of PP_2. Even if the actual values are better in CP_FIFO, the trend is very similar: poor throughput and high unfairness when the second application increases its demand. The reason is that CP_FIFO is in fact a preemptive policy, where an application that goes into its critical section takes over the CPU. By this mechanism the second application steals the processor and given that the send is non blocking, it goes on with it for the non critical part of the computation. If the application has a high *d*, a new message will soon come and it will again steal the processor. Even if the result is similar to PP the reason is somewhat different.

Given the results in this experiment, we performed an additional measure in which the priority of the application which is not in the critical path depends actually on how critical is that part. The priority used is a linear function of the laxity of the code. The more critical a computation is, the higher priority it has. We call these policies CP_RR_DEP and CP_FIFO_DEP, and the results are shown in Figure 79. The behavior of CP_FIFO_DEP is the same as CP_FIFO. The differences appear in CP_RR_DEP with respect to CP_RR but only for some configurations. These correspond to cases where the new policy achieves the desynchronization of the resource requests of the two applications. From the understanding of the effects presented in these experiments, we think that it would be interesting to try policies where there is no sharing (FIFO) in the critical path to avoid synchronization and more and more fine grain sharing as we leave the critical path.

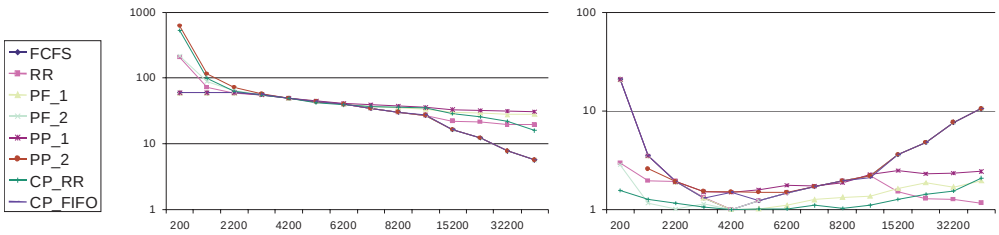
Figure 79. (4200, 60%) [1, 2, 3, 4, 5, 6, 7, 8] and (4200, X%) [1, 2, 3, 4, 5, 6, 7, 8]. System throughput (left) and slowdown ratio (right)



(4200, 60%) [1, 2, 3, 4, 5, 6, 7, 8] and (X, 60%) [1, 2, 3, 4, 5, 6, 7, 8]

This experiment analyzes the influence of the scheduling policies when applications of different critical paths are multiprogrammed. Both applications have the same pattern and CPU consumption when run alone, but while the period of application 1 is kept constant (a = 4200), the period of application 2 ranges from one twentieth to 10 times the period of application 1 (a = 200..44200). We will use the terms bigger or higher critical path when comparing two applications referring to the length of the critical path per iteration (8*a in our experiment).

Figure 80. (4200, 60%) [1, 2, 3, 4, 5, 6, 7, 8] and (X, 60%) [1, 2, 3, 4, 5, 6, 7, 8]. System throughput (left) and slowdown ratio (right)



The behavior of the system should be symmetric around the point where both applications have the same critical path (period). There is nevertheless one reason for not being totally symmetric which is related to the fixed quantum of 1000µs that is used in some of the policies. The characteristic times (a) of application 2 in some cases are explicitly below the quantum value to expose the effect of this parameter on very fine grain applications. The difference lies in whether a policy gives immediate control to the critical code when a thread unblocks or forces it to wait for one quantum. The difference can be very important if the CPU bursts of an application are smaller than the quantum.

In Figure 80 system throughput can be analyzed. When an application has a much smaller critical path than another one, the policies FIFO and CP_FIFO lead to a very bad throughput and a very high slowdown ratio. Under these policies, both applications achieve the same throughput, close to that of the worse application when run alone. The application with smaller critical path has to wait a lot each time the other application gets the CPU. Both policies give identical results because of a synchronization effect by which tasks of the short application always receive a message (and thus enter their critical path) when the other application is already in the critical path and thus the priority has no effect.

RR and CP_RR give the best system throughput and slowdown ratio. This ratio can nevertheless be above two, which is important. Similarly to RR, CP_RR gives chances of progressing to the more critical application when it is in the non critical path. The differences between the two are related to the non symmetric effect of the quantum. When the code in the critical path of one of the applications is small relative to the quantum, the CP_RR policy is much better. When both applications have computations in the critical path bigger than the quantum, this effect is not relatively so important. Nevertheless, CP_RR is in general (but not always) better.

Fixed priority policies, where the priority is given to the application with higher critical path, give the worse throughput and slowdown ratio. The application with frequent communications and tight dependences is given very few chances of progressing. When the priority is given to the application with smaller critical path, somewhat better throughputs than RR are obtained at the expense of somewhat higher slowdown ratios.

The same variation of CP_RR_DEP and CP_FIFO_DEP as in the previous experiment has been tried. CP_FIFO_DEP is equivalent to CP_FIFO. CP_RR_DEP lies between RR and CP_RR (and with worse slowdown ratio than CP_RR) when *a* for the second application is close to the quantum. When *a* is large, it is equivalent to CP_RR.

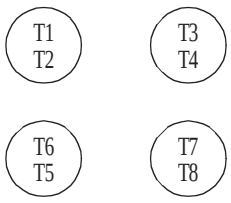
The differences between policies can reach an order of magnitude in system throughput and slowdown ratio. This is an indication of the importance of a proper scheduling in the case of important differences in the criticality of the applications. The scheduling policy should assign priorities depending on the length of the CPU burst between communication.

8.1.4.2 Folded synthetic applications

(4200, X%) [1, 1, 2, 2, 3, 3, 4, 4]

One single application mapped into four processors. Half of the tasks are mapped on the same processor as the next/previous tasks. This is a widely used mapping algorithm that tries to minimize processor communication when not enough processors (one per task) are available. Figure 81 represent four processors and the mapped tasks.

Figure 81. Mapping [1, 1, 2, 2, 3, 3, 4, 4]



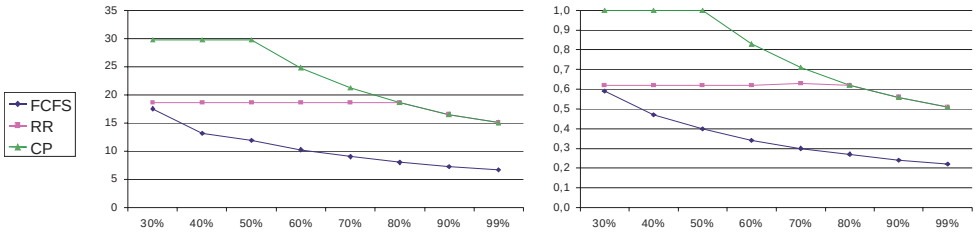
In this experiment, only three algorithms are evaluated. Fixed priority and preemptive fixed priority are not evaluated as we only have one application, so all ready threads would have the same application fixed priority and this corresponds to the RR policy.

Figure 82 shows the performance obtained by the different algorithms. In this experiment there is a linear relation between throughput and slowdown, so plots in Figure 82 displays the same information.

FCFS is the worst algorithm for all the cases. For the values of d ranging from 30% to 80%, the best algorithm is CP. Let us explain this behavior.

Using FCFS a running thread is not preempted after sending a message to the next thread. If both threads share a processor, the receiving one must wait until the first one finishes, thus delaying the flow of control through the critical path. This effect causes an enormous serialization of the computation.

Figure 82. (4200, X%) [1, 1, 2, 2, 3, 3, 4, 4]. System throughput (left) and slowdown ratio (right)



With RR a similar problem appears. In this case, the processor is time shared by the two threads resulting in a high delay in sending a message to the next processor so that it can not start computing. The effect is not as bad as in FCFS but leads to poor resource utilization. As can be seen in Figure 82, when the CPU demand of each task is low this can halve the application performance compared to the maximum achievable. This effect corresponds to the synchronization effect between resource requirements of different applications described in the previous section.

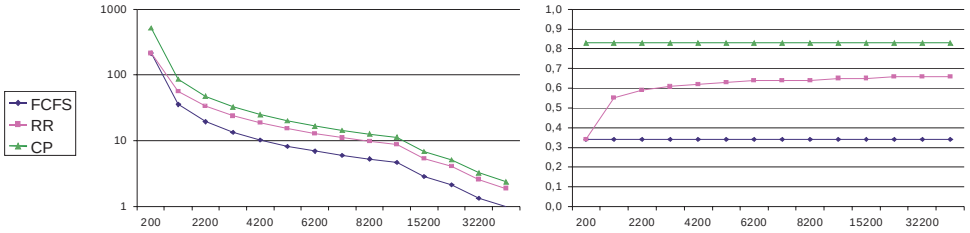
The CP algorithm gives priority to threads in the critical path in front of threads outside it. In this experiment this allows to forward messages in the critical path as soon as possible thus maximizing the performance of the application. When the CPU demand of each task is very high, CP does not improve over RR because the processor is fully utilized.

(X, 60%) [1, 1, 2, 2, 3, 3, 4, 4]

This experiment is similar to the previous one and the objective is to observe how the critical path of the application influences the performance obtained with the different policies.

As previous section, Figure 83 shows that CP is the best algorithm, FCFS is the worst one and RR obtains intermediate performance indicators.

Figure 83. (X, 60%) [1, 1, 2, 2, 3, 3, 4, 4]. System throughput (left) and slowdown ratio (right)



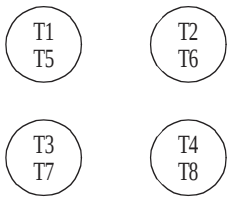
The problem of FCFS is the same as before, it introduces sequential execution by not allowing computation to proceed to a new processor until the previous one has finished a large part of its work.

An important effect pointed out by Figure 83 is that for small values of *a*, RR approximates its performance to FCFS, which is very poor. Computation times are shorter than the quantum and do not let the RR mechanism to act.

When *a* is larger than the quantum, RR stabilizes. The effect described in the previous section for RR appears. The value reached in this case is not very close to the one obtained by CP because the demand of the different tasks is not high enough.

(4200, X%) [1, 2, 3, 4, 1, 2, 3, 4]

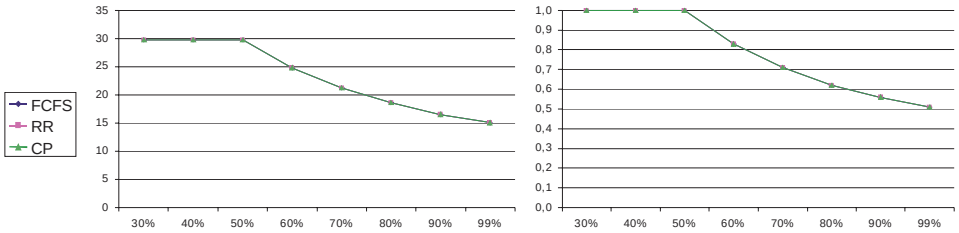
Figure 84. Mapping [1, 2, 3, 4, 1, 2, 3, 4]



The mapping of tasks to processors in this experiment is interleaved as shown in Figure 84. As mentioned at the beginning of this section, a frequently used approach when scheduling a set of tasks to a smaller number of processors tries to maps tasks which communicate in the same processor, in order to minimize communication time. As shown in previous subsections that may introduce reductions in the parallelism achieved due to interference in processor usage. The objective of the interleaved mapping is to let the communications actually to forward some computation request to a different processor trying to maximize parallelism.

The results in Figure 85 show that all the scheduling algorithms achieve the maximum performance.

Figure 85. (4200, X%) [1, 2, 3, 4, 1, 2, 3, 4]. System throughput (left) and slowdown ratio (right)

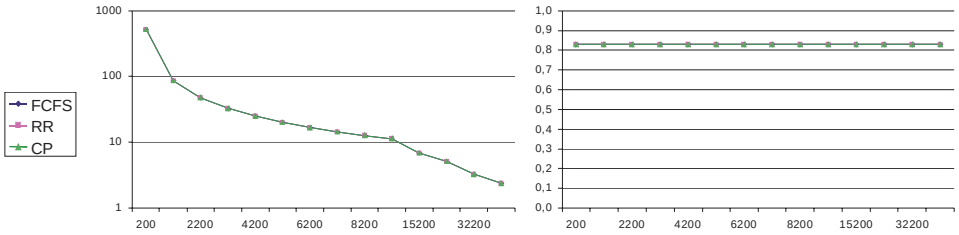


This experiment, as explained in Section 8.1.2, has been simulated using null communication parameters and full network connectivity. We thus don't take into account the actual difference in time between local and remote communication. In a real system, the balance between the degree of parallelism achieved and communication time would determine which of the mapping philosophy is better. These experiments nevertheless demonstrate that the problems originated by the scheduling in the case of folding an application (or explicitly using excess parallelism as a way to improve the processor utilization of an application) should not be disregarded.

(X, 60%) [1, 2, 3, 4, 1, 2, 3, 4]

In this experiment the parameter *a* is varied with the interleaved mapping. The results are the same as in the previous section: all the algorithms lead to the sane performance, achieving the maximum parallelism.

Figure 86. (X, 60%) [1, 2, 3, 4, 1, 2, 3, 4]. System throughput (left) and slowdown ratio (right)



8.1.4.3 Multiprogrammed real applications

In this section we present results of some experiments where two real applications are multiprogrammed with the same mapping of Figure 76 on page 129 used for the synthetic applications.

Figure 87. FIFO (left) and RR (right) on real application. Execution time (up), system throughput (middle) and slowdown (down).

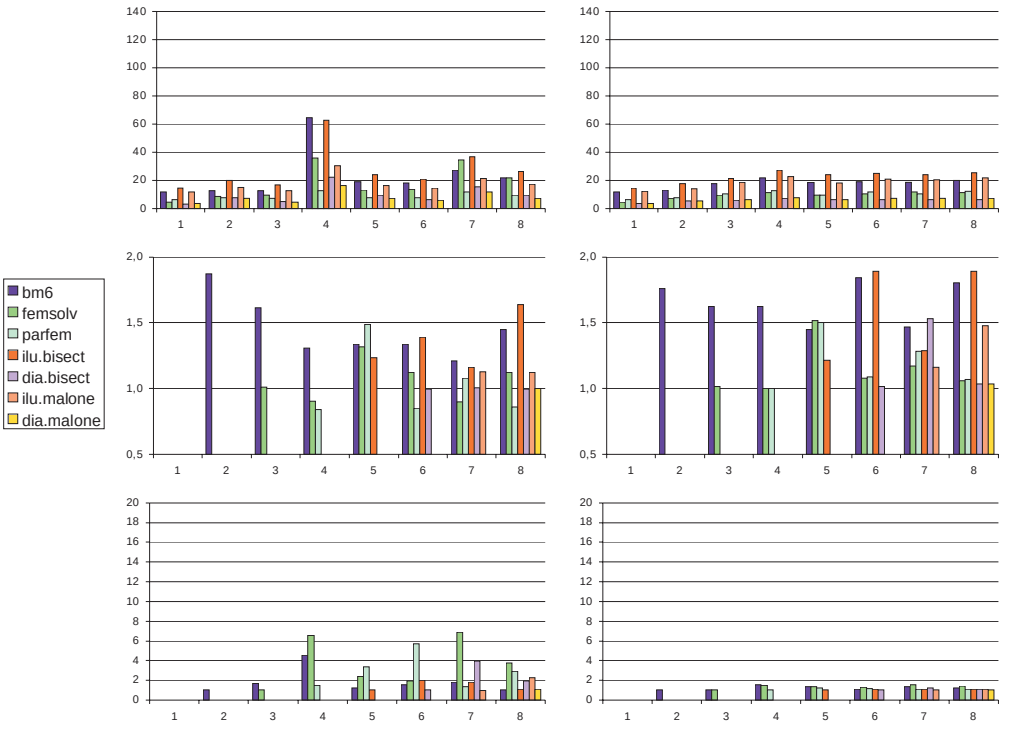


Figure 87 presents the result in terms of execution times for FIFO and RR. The results are presented as groups of bars each of which measures the performance index for a given application combination. The execution time for real applications when run alone is presented in bars with abscissa 1. The order of the bars corresponds to the numbering of the applications as presented in Section 8.1.1.2 and also displayed from top to bottom in Figure 87.

Each of the following groups of bars represents the execution time for the different applications when running multiprogrammed with one specific application. For bars with abscissa between *i* and *i+1*, application *i* is used as specific application.

From Figure 87 it appears that FIFO works quite similar to RR, except for application number three (parfem). Parfem, as described in Section 8.1.1.2 has an important parallel computational time (80% or more of its execution time) without any communication. Using FIFO with this application delay the other application until parfem finishes the current computational time.

In Figure 87, the normalized system throughput is also presented. Each bar represents the ratio between the system throughput and the batch throughput.

In these figures, the group of bars with abscissa between *i* and *i+1* represents the throughput ratio obtained when running applications from *1* to *i* concurrently with application *i*. Note that no repeated information is represented, this means for

example that the results of running *bm6* and *ilu.malone* is only shown in interval *5-6* and not in interval *1-2*.

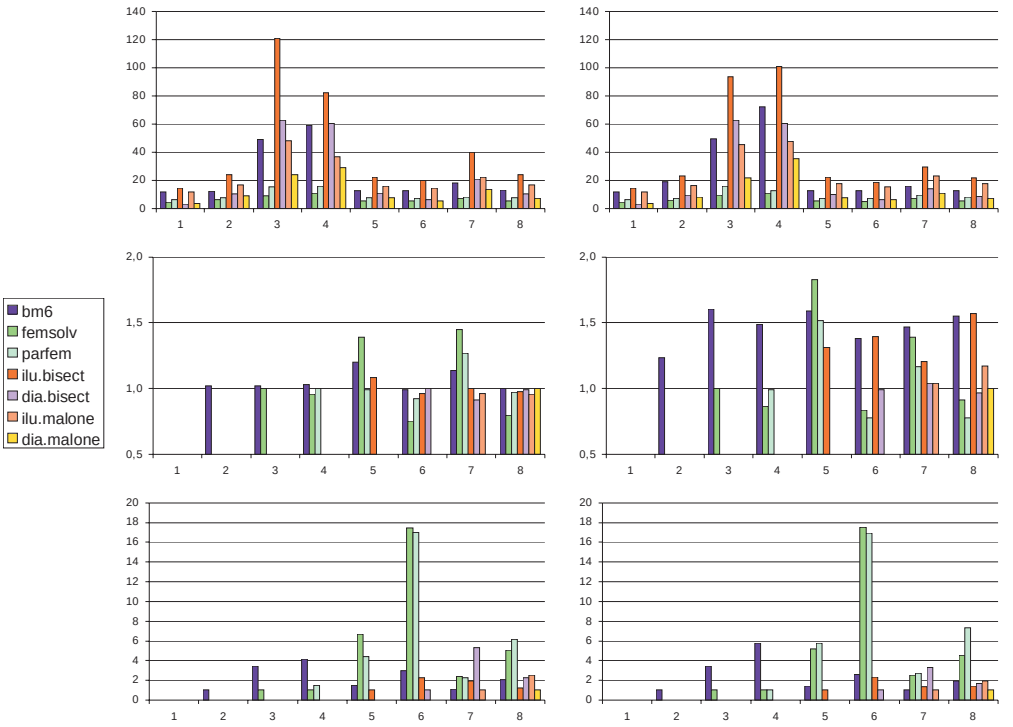
From Figure 87 we observe that the multiprogrammed execution of two applications always obtains better throughput than their batch execution with the RR algorithm. This is not true for the FIFO scheduling. There are combinations for which RR does not achieve improvement over batch, although we believe that if communication was not instantaneous RR would obtain improvements over batch.

Comparing the two policies, RR is in general better although there are cases where FIFO is better in terms of throughput. This happens when one of the applications in the mix is application *1* that has very little parallelism. FIFO is much worse in terms of fairness as shown in Figure 87.

On the average, an improvement of 50% in throughput is obtained for the mixes used in this experiment with RR which supports our hypothesis that fine grain multiprogramming is profitable.

We have also performed the experiment with CP_RR and CP_FIFO. The results are shown in Figure 88. CP FIFO is fairly bad in terms of throughput. CP_RR is in general worse than RR but actually better in some cases. Both policies are very bad in terms of fairness.

Figure 88. CP_FIFO (left) and CP_RR (right) on real application. Execution time (up), system throughput (middle) and slowdown (down).



One problem with these CP policies when applied to real applications which are not so regular as the synthetic one is that there may be computations which are close to be in the critical path and they are not given high priority. As a result, they will probably get into the critical path in the multiprogrammed execution.

8.1.4.4 Folded real applications

In this section we present the results obtained when folding application 4 on to 8 processors. The same non interleaved and interleaved mappings of Figure 81 and Figure 84.

The results are shown in Table 14. The first row in the table corresponds to the performance of the application on 8 processors and is given as a reference.

TABLE 14. Execution time and processor utilization of application ilu.bisect when folded on 4 processors

Policy	Mapping	Execution time	Processor utilization
FIFO	on 8 processes	14.74 s	54%
FIFO	non interleaved	23.16 s	70%
FIFO	interleaved	19.76 s	80%
RR	non interleaved	23.50 s	68%
RR	interleaved	20.90 s	76%
CP_FIFO	interleaved	19.08 s	84%
CP_RR	interleaved	20.90 s	76%

The real application has not such a uniform pattern as the synthetic one, and the effects there observed (Section 8.1.4.2) also appear here with a less abrupt impact on performance. We can see that by folding the application we go from a processor utilization of 54 to values between 68 and 85. Differences between the performance obtained with the different policies are very significant. They do not nevertheless reach the level obtained with the synthetic application.

For a given policy, interleaving produces better performance by improving the possibility of parallel execution. As described in Section 8.1.4.2, if we consider that a message carries with it some work to be done, it is interesting to send it to other processor that might be idle.

For the non interleaved mapping, FIFO gives a slightly better performance than RR. This does not correspond to the results in Section 8.1.4.2. For the interleaved mapping, FIFO is also better than RR. The pattern of this application does not shift circularly between processors but does a scan back and forward. Recall from Section 8.1.4.2 that RR achieved the desynchronization after several iterations of the circular pattern. From this point of view, in this real application the RR system is always in a transitory situation and is not able to avoid overlaps that the more drastic separation of resources enforce it by FIFO achieves much faster.

The CP policies have also been tried. While CP FIFO improves a bit over FIFO, CP RR behaves as RR. The CP policies for real applications are very restricted because only one thread in one application is prioritized. The communications will propagate the delays of the other threads to the whole application, included the high priority thread.

The detailed analysis of the behavior of the folded execution with Paraver show that for all the policies presented there are moments where it would be better not to apply round robin or to select the other thread. This makes us think that further improvements should be possible.

8.1.5 Conclusions

The work carried out in this task of the project has produced a set of tools and tuned a running platform which have been used to evaluate short term scheduling policies in Distributed Memory Machines.

The study concentrates basically in the study of the effects that appear when simple policies are used. The interference between two independent applications is studied. We have shown that simple policies such as round robin can achieve 50% better system throughput than batch execution for the mix of real traces used. Very pessimistic simulation conditions have been selected on purpose to check the validity of fine grain multiprogramming.

We also studied the interference between tasks of the same application if more than one task is mapped on each processor. The results show the substantial importance of the differences in performance achievable with different policies. FIFO policies tend to be better in real applications. The trade off between mapping which favor local communication and those where communicating tasks are put in different processors has also been identified. The first one tries to minimize communication time and the second one for increasing parallelism.

We have identified several effects that appear when several threads share processors in a message passing machine. Further effort is nevertheless required in order to devise and test policies that take profit of the effects observed, taking profit of the good ones and avoiding the risks that have also been identified.

Other side effects of this work are the open line of using the simulation and visualization tools to support program development and tuning for users concentrating on single applications. In this line, integration of a compilation tool would enable the evaluation of code restructuring methods without the need of a machine that actually supports them. For example, different communication semantics, different local operation orders within the application, communication schedules, different process models,... From our point of view, the comparison of how these programming model variations interfere with the operating system scheduling policies is very interesting.

8.2 Short term scheduling and dependence chain

This second case study is an extension of the one described in Section 8.1. From the point of view of workload, we incorporate the utilization of some benchmarks from NAS. From the point of view of scheduling we propose three different variations of the same algorithm that tries to locate the application critical path dynamically. The analysis in this case study also incorporates 4 task, 8 task and 16 task applications. And finally, it also incorporates analysis of dependencies on network bandwidth and the time slice of the process scheduling policies.

8.2.1 Workload

In this section, we describe the three applications employed of the process scheduling evaluation. All three are par of the NAS Parallel Benchmarks [71], included in PARKBENCH [72]. The NAS version we use is based in the communication library PVM. The necessary Dimemas traces were obtained using two dual ALPHA workstations.

8.2.1.1 Multigrid, MG

Corresponds to four iterations of the V-cycle multigrid algorithm are used to obtain an approximate solution υ to the discrete Poisson problem

$$\nabla^2 u = v$$

on a 256x256x256 grid with periodic boundary conditions. This is a simplified multigrid kernel. It requires highly structured long distance communication and tests both short and long distance data communication.

Figure 89. Multigrid, MG

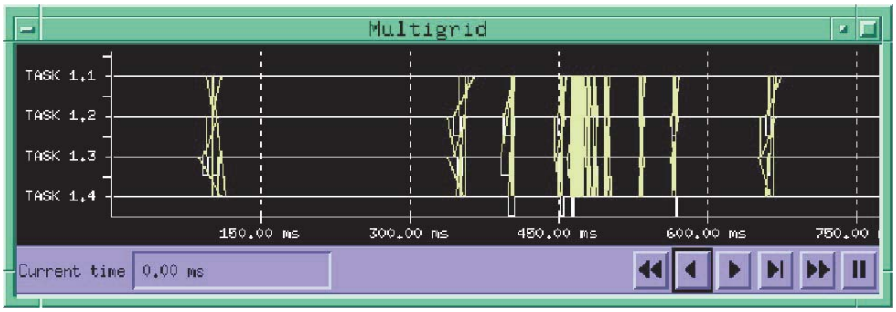


Figure 89 shows a Paraver view of the whole execution of Multigrid, for one of the four iterations. This visualization corresponds to four tasks simulated using ideal communications (zero latency and infinite bandwidth). Each task has two possible values (active/non active) and we also include the communications between tasks.

From a static analysis of the application and using ideal communication parameters, we can provide summary information for the application when using 4, 8 and

16 tasks. The information in each cell corresponds to the average values and the standard deviation of all the tasks. This information is presented in Table 15.

TABLE 15. Static summary information of MG

	Utilization	Send	Receive	Parallelism
4 tasks	(94.70, 1.41)	(180, 9.80)	(180, 5.66)	3.89
8 tasks	(94.25, 1.83)	(194, 24.00)	(194, 19.36)	7.58
16 tasks	(96.69, 0.79)	(169, 34.83)	(169, 30.02)	15.55

From information in Table 15 and Figure 89, we can conclude this application has coarse grain parallelism, because there are more computational time compared to the number of communications. From the analysis, we can also conclude this is a well balanced application, where all the tasks performs the same amount of work.

8.2.1.2 Conjugate Gradient, CG

Solving an unstructured sparse linear system by the conjugate gradient method. A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication.

Figure 90. Conjugate Gradient, CG

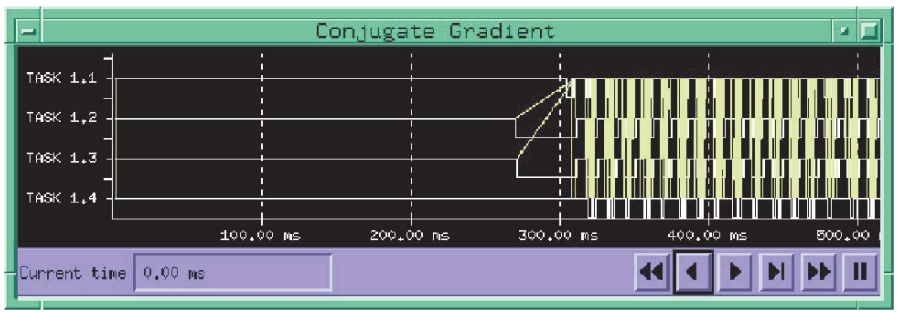


Figure 91. Conjugate Gradient, CG. Zoom in the communication area

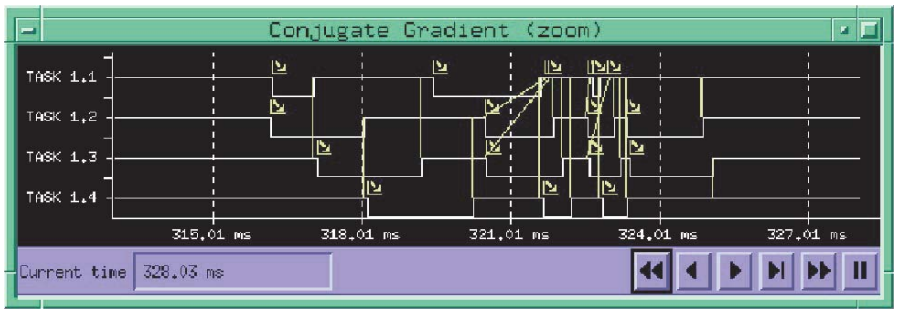


Table 16 contains the static analysis information of the application when executed using 4, 9 and 16 tasks., using ideal communication parameters.

TABLE 16. Static summary information of CG

	Utilization	Send	Receive	Parallelism
4 tasks	(69.00, 10.13)	(1801.50, 958.79)	(1801.50, 705.38)	2.78
8 tasks	(50.44, 14.23)	(2221.78, 2208.63)	(2221.78, 1912.3)	4.58
16 tasks	(39.56, 15.60)	(2398.16, 3265.97)	(2398.16, 2941.43)	6.42

For this application we provide two Figures. Figure 90 shows the initial phase of the execution and some iterations when using 4 tasks. In this figure, we can observe there is a long and balanced initialization part involving all tasks. After this initialization phase, there is a communication intensive area, repeated until application finalization. Figure 91 is a zoom of this communication area (one iteration) for CG application. In this Figure 91, we can observe some unbalanced situations of the application, because some tasks are blocked while others are still working.

From the information of Table 16, Figure 90 and Figure 91, we can conclude this is a low grain application from the point of view of parallelism, because there are many communications and the computation time between communications is small. The degree of parallelism also increases while increasing the number of tasks.

8.2.1.3 LU

LU is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) to solve a block lower triangular-block upper triangular system of equations resulting from an unfactored implicit finite-difference discretization of the Navier-Stokes equations in three dimensions. Although the described algorithm is not optimal for the application at hand, it is being retained as a benchmark because it is very sensitive to the small-message communication performance of an MPI implementation. It is the only benchmark in the NPB 2.0 suite that sends a large numbers of very small (40 byte) messages.

Figure 92. LU complete iteration

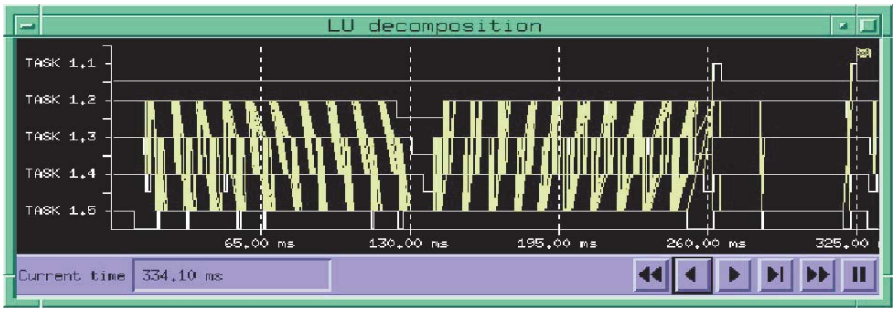


Figure 92 shows a complete iteration of the LU execution using 4 tasks. The flag at the end of task 1 bar represents the end of the iteration, and has been included in the application for performing analysis using Paraver.

This application is not using the programming model SPMD (Single Program Multiple Data) as can be observed in Figure 92, because task 1 is performing the work of a master task. For this reason, Table 17 information has been divided in two values: the information related to task 1 and the average information of the remaining tasks. This is to avoid the interference of the data values of task 1 in the whole analysis. For other tasks, it is important to remark the high value of processor utilization they obtain, as well as the high number of communications. There is a clear communication pattern, with most of the communications being grouped. The parallelism level is not very high, but it increases as the number of task increases (not proportional, but lineal relation).

TABLE 17. Static summary information of LU

	Utilization	Send	Receive	Parallelism
4 tasks	(1.00, 0.00) (91.75, 1.50)	(0, 0.00) (5169.50, 147.00)	(98, 0.00) (5145.00, 98.00)	2.78
8 tasks	(3.00, 0.00) (74.65, 16.35)	(0, 0.00) (5206.25, 1522.66)	(98, 0.00) (5194.00, 1531.26)	4.58
16 tasks	(5.60, 0.00) (60.25, 17.86)	(0, 0.00) (4011.88, 1416.30)	(98, 0.00) (4005.75, 1418.29)	6.42

8.2.2 Process scheduling policies

In this section we describe the scheduling policies we employ in this study. We have included four different policies, starting from basic ones (FCFS and Round Robin), including also Unix-like policy and finally we propose a priority based one, we called it BOOST, that tries to recognize the application critical path in execution time.

The basic policies have been described in previous sections of this work. We describe in the following sections the two new approaches.

8.2.2.1 DECAY, Unix-like process scheduling

This is the process scheduling algorithm implemented in most of the Unix systems. It is based in priorities and time slices. Each thread in the system has a priority assigned. The first thread to execute is the one with higher priority. In the same priority, time slices are applied. The priorities are modified according the processor utilization of each thread.

Compared to FCFS and Round Robin, DECAY incorporates a new possibility in process scheduling. This is preemption. If the running thread has worst priority than a ready to run thread, and the Dimemas flag *priority preemptive* is set, the running thread will be preempted from execution.

Dimemas allows the modification of several parameters to tune the scheduling policy: maximum priority, minimum priority, time interval to recompute processor utilization, time interval to compute average workload of processors, and time interval

of priority readjustment. The following equations show the computation of priorities. Refer to [73] for an explanation of Unix schedulers.

$$prio \in [p_{min}, p_{max}]$$

$$prio = u_{thread} + baseprio_{thread}$$

$$u_{thread} = u_{thread} + 1$$

$$u_{thread} = \frac{2 \times load_{cpu}}{2 \times load_{cpu} + 1} \times u_{thread}$$

8.2.2.2 BOOST, priority based process scheduling

In the previous schedulers of this case study, FCFS and Round Robin have fixed priorities; but DECAY modifies the priority according to the resource requirements of every thread. In BOOST scheduling policy the priority is modified according how a thread gets block/not block when receiving messages. We have included three different parameterization of this policy:

1. BOOST$_1$

$$priority = \begin{cases} priority + delta_1 & \text{if receiver arrives first} \\ priority - delta_2 & \text{if receiver arrives last} \end{cases}$$

   If the receiver thread arrives to the receive statement before the message is located in the incoming message queue of the node, we decrease the priority of the thread. The logic is: this thread was faster than the message, next time it can run slower.

   But if the message is already in the queue, we increase the priority to speed the execution of this thread.

2. BOOST$_2$

$$priority = \begin{cases} priority - delta_1 & \text{if receiver arrives last} \\ priority + delta_2 & \text{if receiver arrives first} \end{cases}$$

   The equations are identical, but the logic has changed. We increase the priority if the receiver arrives and the message was ready to delivery, because that message was urgent (already local to the node) so we must speed the execution of the thread (consuming the message information).

   Opposite, if the receiver arrives before the message, then the message is not so urgent. In this situation, the receiver thread can proceed slower, assuming this message is not in the critical path of the application.

3. BOOST$_3$

$$priority = \begin{cases} priority - delta_1 & \text{if receiver arrives last, applied to receiver} \\ priority + delta_2 & \text{if receiver arrives first, applied to receiver} \\ priority + delta_3 & \text{if receiver arrives last, applied to sender} \end{cases}$$

This is a small variation of $BOOST_2$. We include a new equation to affect the sender priority. If the message arrives later than the receiver, we increase the priority of the sender with the logic that the message was important for the receiver, it may happen that this message was located in the critical path of the application.

8.2.3 Evaluation

In this section we specify all the conditions we employ for the measurements of the different experiments as well as the performance indices we employ for the comparison of the different scheduling policies.

- All applications require more than 3 seconds on a real machine. The execution of these applications generates the traces to feed Dimemas. The rationale is to provide enough time to the DECAY policy to react to the application behavior. Otherwise, the application should be finished before the scheduler modifies any priority. We model a closed network, to guarantee that during all simulation time, all three applications are running concurrently.
- LU mapping is [1, 1, 2, 3, 4] for the 5 tasks execution. As task 1 does not use much processor time, it will not interfere in the whole execution and the whole utilization of processor 1.
- For CG benchmark running in 9 tasks, we map the last one to a node completely dedicated to the last task. This is an advantage to this task/application, because this task will never be delayed in its execution, but this will not affect significantly the whole analysis because other tasks are sharing processors. In any case, we will take this in mind when analyzing this special experiment.
- For the communication semantics we employ asynchronous and buffering communication.

Let us define the following system parameters:

- Application response time in a dedicated machine (Ta_i)
- Application response time in a shared environment (Tc_i)
- Slowdown for application (S_i) and for the whole system (S). The value for S, if it is greater than 1, means a gain over batch execution.

$$S_i = \frac{Ta_i}{Tc_i}$$

$$S = \sum_{i=1}^{3} S_i$$

The same results can be obtained with the following equations:

$$T_{max} = max(Tc_1, Tc_2, Tc_3)$$

$$N_i = \frac{T_{max}}{Tc_i}$$

$$T_{sec} = \sum_{i=1}^{3} (Ta_i \times N_i)$$

$$S = \frac{T_{sec}}{T_{max}}$$

To compare the fairness of the scheduling policy to all the applications. we will use the standard deviation of the partial slowdown of each application.

$$stddev = \left(\frac{\sum_{i=1}^{3} (S_i - \bar{S})^2}{2} \right)^{(1/2)}$$

8.2.4 Experiments

8.2.4.1 Applications with 4 tasks

Figure 93. Speedup (left) and Fairness (right) using ideal communication parameters, modifying the time slice (in μs)

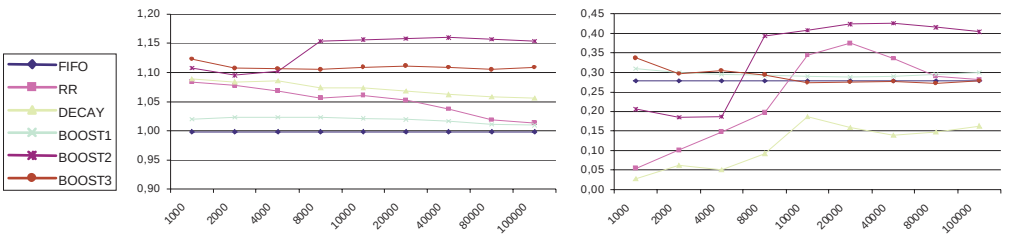


Figure 93 shows the speedup and the fairness of the process scheduling policies, when using 4 tasks applications, ideal communication parameters and different values for the time slice parameter.

From this figure, we can observe that processor sharing produce benefits, because speedup is greater than one, except for FCFS policy. It is important to remark that although the speedup results for FCFS and Batch execution are similar, the fairness of both solutions is quite different. Using FCFS policy, there are much more executions of LU and MG than CG, thus CG has been treated on unfair basis.

It is also important to remark that time slice variation does not affect speedup, but it really affects the fairness distribution of resources between the different applications. For larger time slices, the worst fairness.

Figure 94. Speedup (left) and Fairness (right) modifying the communication bandwidth. From top to bottom, different time slices (1ms, 10ms, 100ms).

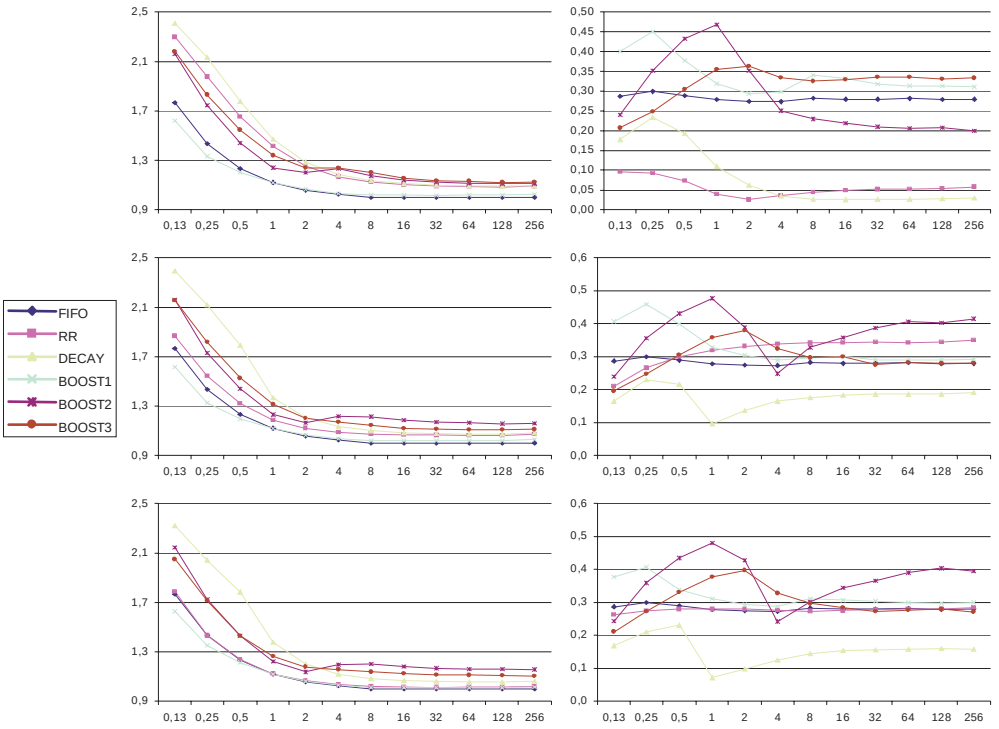
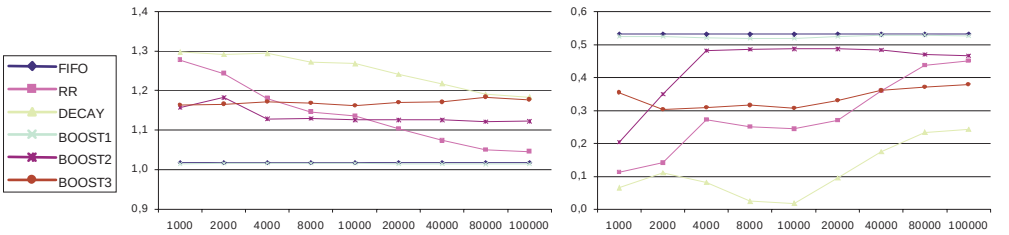


Figure 94 shows the results for the experiments on processor sharing for 4 tasks applications, for different scheduling policies and modifying the communication bandwidth parameter. Each row contains a pair of figures correspond to speedup and fairness for a given time slice (1ms, 10ms, 100ms). From this figure we can extract the following comments:

- For small bandwidth, DECAY obtains better performance and better fairness than any BOOST policy. For small time slice values, RR performs similar to DECAY, but distributing resources more equitable.
- $BOOST_2$ and $BOOST_3$ performs better than DECAY for bandwidth greater than 4MB/s.
- In most cases, DECAY is the most equitative policy.
- $BOOST_1$ does not introduce any improvement, although we expect some. In most cases, it does not provide different results than FCFS. It is possible that the reason for this unexpected result is that priority modification affects the future behavior of the application, but based in the previous utilization, instead of giving importance to the messages.

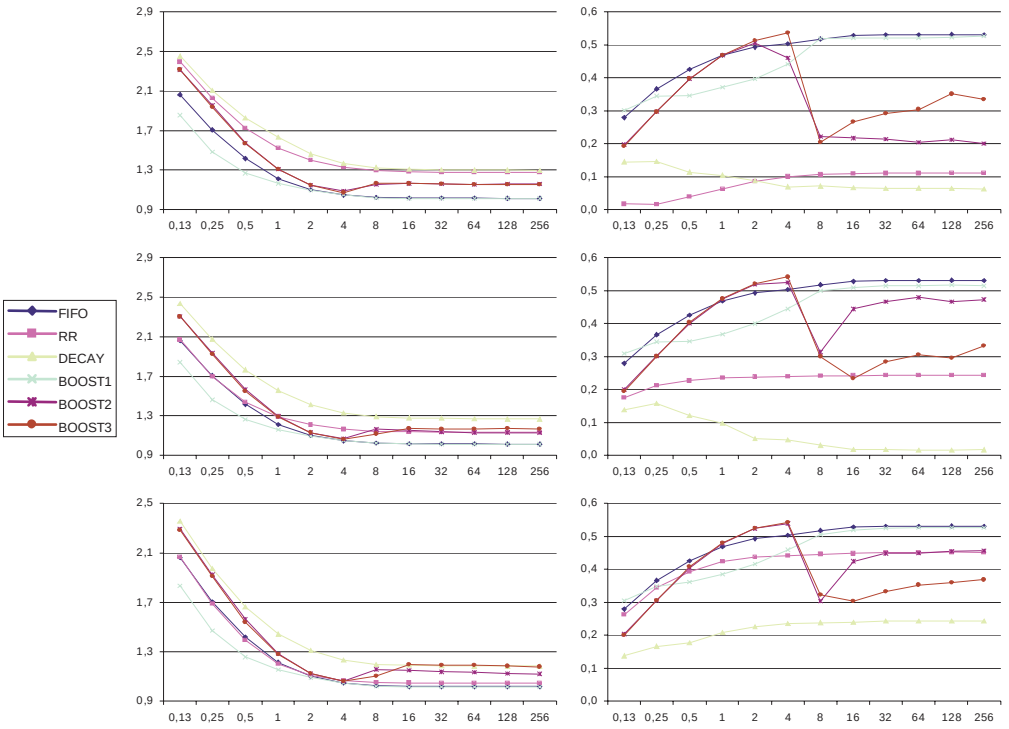
8.2.4.2 Applications with 8 tasks

Figure 95. Speedup (left) and Fairness (right) using ideal communication parameters, modifying the time slice (in μs)



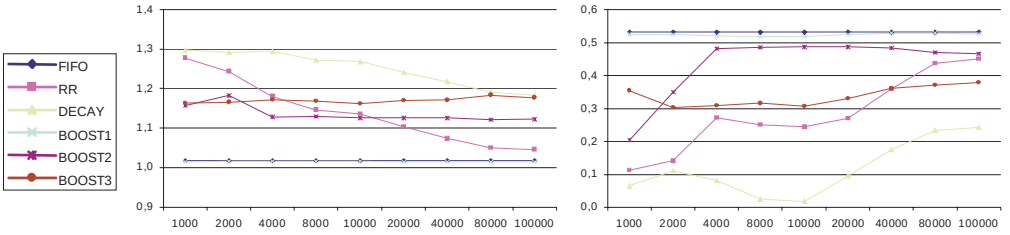
- FCFS and $BOOST_1$ are confirmed as the worst processor scheduling policy evaluated. This is valid for performance and fairness parameters.
- DECAY obtains good results, but specially for small time slice.
- RR obtain worst results as we increase the time slice.
- $BOOST_2$ and $BOOST_3$ are not really influenced in time slice variation. The problem is there are only three application with different priorities each, and time slice is only applicable when two threads have the same priority level.

Figure 96. Speedup (left) and Fairness (right) modifying the communication bandwidth. From top to bottom, different time slices (1ms, 10ms, 100ms).



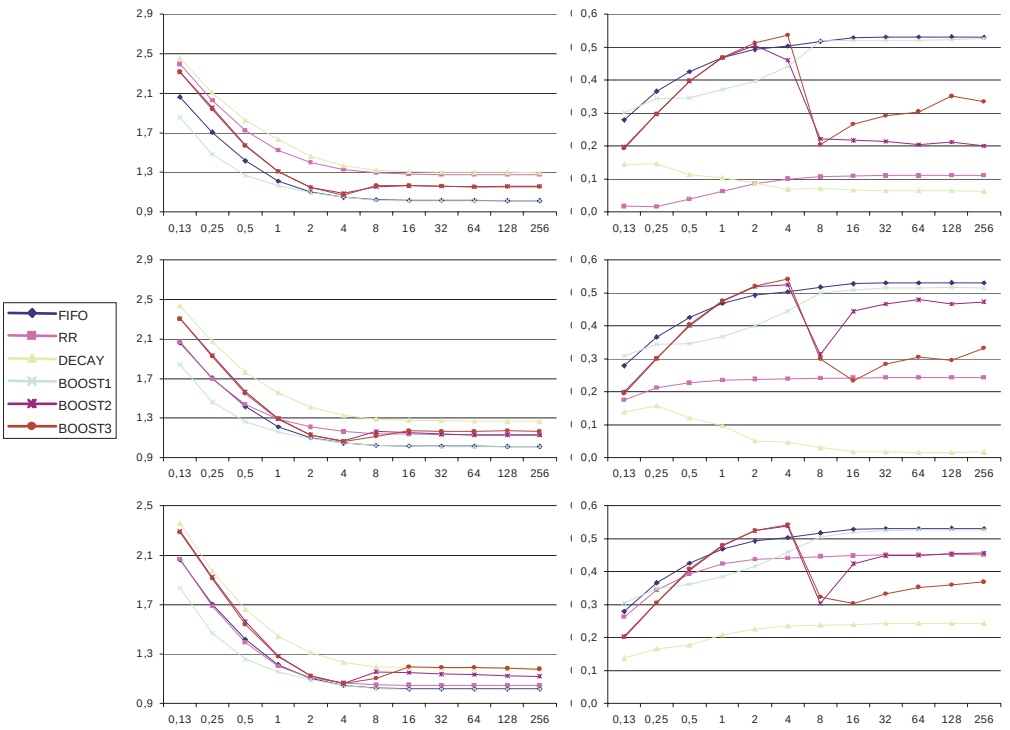
8.2.4.3 Applications with 16 tasks

Figure 97. Speedup (left) and Fairness (right) using ideal communication parameters, modifying the time slice (in μs)



- The best scheduling policy in terms of equitativity is DECAY.
- Processor sharing is a good choice, as performance is better than running in batch mode.
- For small values of bandwidth, BOOST policies behaves acceptable, although they only take into account the theoretical importance of a message.
- As we increase communication bandwidth and the number of tasks per application, the best scheduling policy is DECAY.

Figure 98. Speedup (left) and Fairness (right) modifying the communication bandwidth. From top to bottom, different time slices (1ms, 10ms, 100ms).



8.2.5 Conclusions

From the previous examples, we can conclude the following:

- The methodology described in this section is valid for scheduling policies analysis. It mixes several applications, and takes into account the speedup and the fairness obtained using each policy.
- From the point of view of the different scheduling policies, the results remark that $BOOST_1$ is not a good policy (poor speedup and unfair). We do think this is because this policy uses a bad logic. From the whole set of experiments, it is clear that the most important characteristic for distributed applications are the messages, and the contention they produce when being delivered later than expected. $BOOST_2$ and $BOOST_3$ try to get advantage of this characteristic, and obtain better results than $BOOST_1$. But still not significantly better than Round Robin or Unix-like policies. The difference between $BOOST_2$ and $BOOST_3$ does not introduce any gain in performance. This indicates, in some sense, that once a message has been delivered, it is unimportant to the task if the message arrives on time or late. To this task it becomes important the next message to be received. It will be a goof chance to know in advantage the next message characteristics. This will be a topic for future analysis.
- Round Robin and Unix-like policies are, for this group of applications, the best policies, in terms of performance and fairness. It is preferable to use small time-slices.

8.3 Basic scheduling policies study for HPC and NOW environment

In this section we will focus the analysis in two basic scheduling policies, but using a more elaborated workload and using different performance indices.

The policies we use for this study are: FCFS (First Come First Serve), a running tasks only frees the processor if it uses the receive communication primitive for a message that is not located in the node (the task must block until the message reaches the node); and RR (Round Robin), a running tasks may leave the processor if the time slice assigned to this task has finished or the task blocks itself receiving a message. In this last option of RR, the remaining time slice is not reassigned when the message reaches the node; in fact, the task must wait for a new and complete time slice. The time slice used is 10 ms. This time slice is short enough to allow a good share of the processor and long enough to reduce the impact of the context switch time.

In Section 8.3.1 we describe the workload selection, the characteristics of each benchmark selected and the simulation environment. Section 8.3.2 describes which parameters we use for the global system analysis. Section 8.3.3 contains the result analysis. And, finally, Section 8.3.4 summarizes the section goals.

8.3.1 Workload and simulation environment

The workload used in our work comes from some mixture of different NAS parallel benchmarks (NPB). We have used a PVM version of the NAS codes.

We have selected 4 from the 8 NAS benchmarks (CG, IS, LU, and MG) and we have mixture them in a three-elements group, obtaining four different workloads (named B1, B2, B3, and B4). Table 18 contains the workload names and the applications included in each workload.

We have used the sample class problem size because we want to analyze the effect of time sharing on low granularity applications. We are also interested in mixing applications that achieve high and low processor utilization.

The application traces for this work have been obtained in a Power Challenge with 12 R10000 processors and using the PVM 3.10 socket based version. Table 19 contains some relevant application characteristics: average number of bytes per message, number of messages, application time, average processor utilization and number of messages per second. The last three have been obtained for two different architecture environments: NOW (Network Of Workstations) and HPC (High Performance Computers). The communication parameters for both environments are presented in Table 20. Bus contention means that only one message can use the network concurrently. 1 link means that there are only one input and one output attachments from each node to the communication network, and the communication network is supposed to allow full connectivity among nodes. Each node has a unique processor.

TABLE 18. Workloads names and the applications included in each workload. Four different workloads using four applications from the NPB.

Workload	Application 1	Application 2	Application 3
B1	CG	IS	LU
B2	CG	IS	MG
B3	CG	LU	MG
B4	IS	LU	MG

TABLE 19. Application characteristics for the NAS codes: CG, IS, LU, and MG. This table shows the results obtained when the application is simulated in a dedicated system with the properties for NOW and HPC computers. Time is measured in seconds.

	Bytes per message	Num. mess.	NOW			HPC		
			time	%cpu	mess/s	time	%cpu	mess/s
FFT 4	229378.6	48	42.387596	12.57	1.13	5.794899	91.17	8.28
FFT 8	61168.4	105	9.161998	33.35	11.46	3.269270	90.56	32.12
IS4	272730.9	121	139.025563	13.81	0.87	25.586458	74.62	4.74

TABLE 19. Application characteristics for the NAS codes: CG, IS, LU, and MG. This table shows the results obtained when the application is simulated in a dedicated system with the properties for NOW and HPC computers. Time is measured in seconds.

	Bytes per message	Num. mess.	NOW			HPC		
			time	%cpu	mess/s	time	%cpu	mess/s
IS8	79399.5	243	20.815442	51.02	11.68	17.806472	58.42	13.65
LU5	152.4	5290	15.730039	82.10	336.30	8.791915	93.19	601.69
LU9	151.4	5330	13.873308	69.50	384.17	6.393461	74.97	833.63
MG4	9240.8	316	6.522070	31.81	48.49	1.915991	99.75	165.06
MG8	10175.2	164	2.102434	62.06	77.89	1.233535	93.83	132.75

TABLE 20. Communication parameters for the NOW and HPC environments. Includes latency, bandwidth and network restrictions

Environment	Latency (μs)	Bandwidth (MB/s)	Network restrictions
NOW	500.0	1.0	bus contention 1 bus, 1 link
HPC	50.0	30.0	no bus contention infinite buses, 1 link

Table 19 allows us to classify the four NPB in terms of the following characteristics:

- Application static characteristics

  These values characterize the application granularity, which is expected to have a significant effect on its behavior under the multiprogrammed environment. They are:

  1. Number of communications

     Applications FFT, IS and MG use very few communication primitives, but this number is very high for the LU benchmarks. LU uses the same magnitude of communications primitives when using 5 or 9 tasks. In the other hand, MG uses less communication primitives when running with more tasks.

  2. Average bytes per message

     Again, this value is in the same order of magnitude for FFT and IS benchmarks and both applications use a lower message size when using more tasks. LU uses very short messages, and the size is maintained constant for the five and nine task executions. The MG benchmark uses medium size messages and this size is greater for the eight task application than for the four task.

- Simulation in NOW computers

  All benchmarks do not use all the processor time because of the network low bandwidth. In the case of FFT4 and IS4 this is more critical, and induces us to

think on sharing processor resources. The application gain for IS and FFT when using eight tasks is due to the reduction of the message size, and then the reduction of the bus contention and its influence in the critical path of the application.

- Simulation in HPC computers

  HPC environment is included to analyze the possibility of sharing processor and network resources. In this situation, the processor utilization is, on most of the codes, greater than 90%. These simulations show the poor scalability of the application due to the small problem size.

8.3.2 Processor scheduling evaluation

To ensure the correctness of the results, we have evaluated the different scheduling policies with the following condition: the measurements used have been taken from simulations with a close queue mechanism, where each application is simulated repeatedly until all application of the benchmark have been executed at least a certain number of times. In our experiments, the minimum number of executions for each application is 10, and the results presented correspond to the time interval where all applications are running.

To compare the different scheduling policies, we have used the following global system parameters: application slowdown and system throughput. Both of them can be computed using the application time T_{si} when the application is executed in a shared environment and application time T_{di} when the application is executed using a dedicated computer. *n* represent the number of applications included in the workload.

The equations to compute the system parameters, for the two different environments, are the following:

- Batch environment

  In dedicated batch environments, the execution of several applications is performed in sequential order. The global system throughput can be computed as the number of applications divided by the summation of the application times.

$$Th_d = \left(\frac{n}{\sum_{i=1}^{n} T_{di}} \right)$$

- Time sharing environment

  In time sharing environments, the execution of several applications is performed in parallel. The global system throughput can be computed as the summation of the individual throughput obtained in each application.

$$Th_s = \sum_{i=1}^{n} \frac{1}{T_{si}}$$

The slowdown obtained because of the non dedicated systems, can be computed, for each application, as:

$$S_i = \frac{T_{di}}{T_{si}} \qquad 0 \le S_i \le 1$$

If the slowdown, S_i, is near to 1, it means that the execution for this application is not delayed with respect to a dedicated one if it is executed in a non dedicated system. On the other hand, if the value is near to 0, this application suffers a high penalty when run in a time shared environment.

The slowdown analysis allows us to evaluate the fairness of the scheduling policies. A fair scheduling policy, in a distributed environment, is the one in which all individual slowdowns are greater or equal to *1/n*, where *n* is the number of application sharing resources. If some application gets more than *1/n* and other applications get near *1/n*, then processor scheduling is unfair and the first application takes advantage of the sharing resources, while the other ones are not penalized.

The left graph, in Figure 99, shows the throughput using the NOW environment, and the graph on the right shows it in the HPC environment. Both graphs, as those in Figure 100, can be analyzed as follows. Each of the different workloads (from B1 to B4) has one different set of columns in the graph. For a given workload, three different bars are presented. The first one represents the throughput of this workload when running in a batch (dedicated) system. The second bar present the throughput of each individual application included in the benchmark and the global system throughput obtained when FIFO is used in a shared environment. The last bar has the same information as the previous one, but it refers to the throughput when Round Robin is used.

The slowdown parameter can be analyzed in the graphs of Figure 101 and Figure 102. In these figures, the information is presented as follows: each of the different workloads (from B1 to B4) has one different column in the graph. For a given workload, six bars are presented, two for each application. The three leftmost bars are the individual application slowdown when using FIFO scheduling, and the three rightmost bars are due to Round Robin. The maximum value for the slowdown is 1.0, and a fair policy (in our case) will be that one with all bars are higher than 0.33.

8.3.3 Results analysis

First of all, resource sharing is also a good solution for distributed memory machines. The system throughput in all workloads and environments is closer or higher to the batch value. Why is the throughput for NOW computers, in Figure 99 and Figure 100, lower than for HPC computers? Because our proposal is to share the whole computer, including processors and communication network. In NOW computers, the bottleneck for the applications is the network. Then, resource sharing motivates a better processor utilization but the network is still the bottleneck.

Figure 99. Throughput (in jobs per second) with four task workload. For each workload, the batch, FIFO and Round Robin throughput is presented

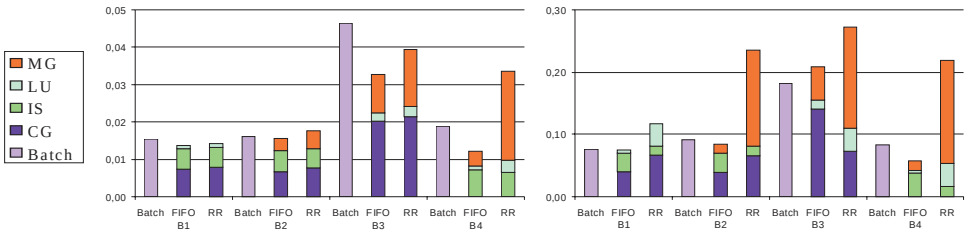
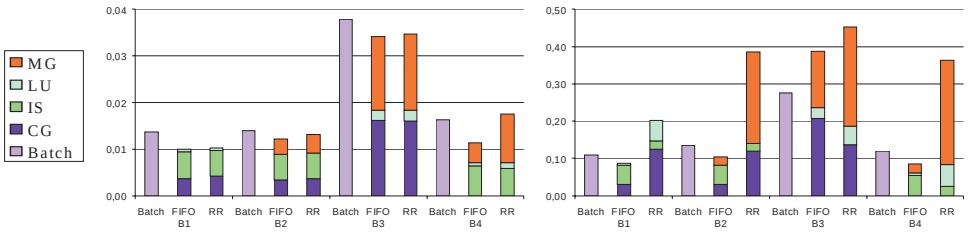


Figure 100. Throughput (in jobs per second) with four task workload. For each workload, the batch, FIFO and Round Robin throughput is presented



The slowdown analysis from Figure 101 and Figure 102, shows that for NOW environments there is no fairness at all, neither with FIFO nor Round Robin. The reason is that the network is a bottleneck. Scheduling policies try to share processor resources among applications but the bus conflict modeled in NOW environments uses a *First In First Out* policy. In this situation, the application with less communication takes advantage over the other ones (FFT and IS in Figure 101, Figure 102). The possible solutions are to use a non bus conflict network but the same communication parameters or to use HPC environments. Table 21 contains the slowdown when a non bus conflict network is modeled in a NOW environment. With 4 task applications and FIFO policy, there is still unfairness due to the difference between applications (each application has a different message per second ratio). With the Round Robin policy, best results are obtained, but the better ones are with 8 task applications and Round Robin. In this last case, a very impressive system sharing is obtained. Table 22 contains the individual and global system throughput in this situation.

Figure 101. Slowdown with four task workload. For each workload, the slowdown for each application is presented using the FIFO and Round Robin policies

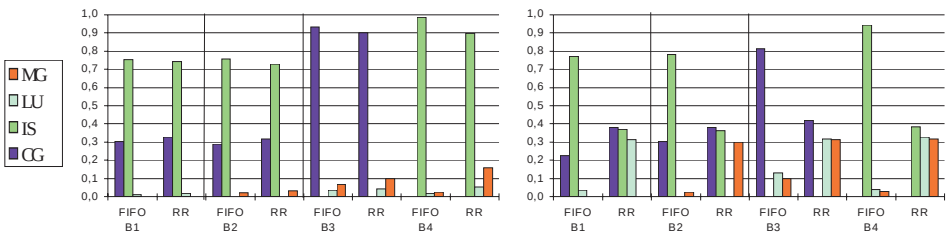
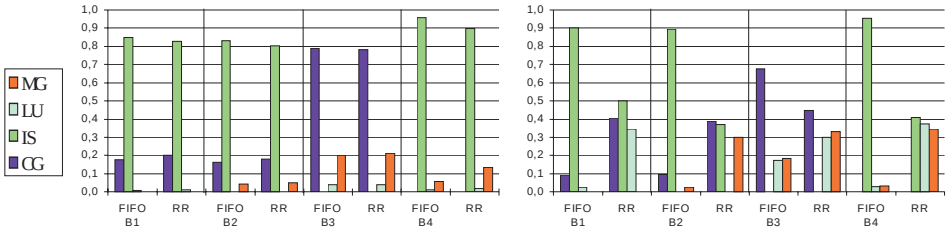


Figure 102. Slowdown with eight task workload. For each workload, the slowdown for each application is presented using the FIFO and Round Robin policies



Why do some applications, FFT and IS, take advantage in the execution in shared environments? The reason is located in the application behavior in dedicated systems. Just refer to Table 19, FFT and IS are the applications that use less communication primitives. A process releases the processor when blocking for a non ready message, and it will resume after the arrival of the message when the scheduling algorithm assigns the processor to it. Process with long CPU bursts and infrequent communications are favored in a shared environment. Round Robin is more fair than FIFO as there is a system imposed limit *(quantum*(number of process-1))* on the time from the actual arrival of the message, until the process is rescheduled. This fairness problem, which is originated at the node level, propagates and accumulates through the dependence chain of the application. That is the reason why a policy such as Round Robin, which is introduced on single processors for fairness purposes, may lead to unfairness in a parallel environment.

TABLE 21. Slowdown with four and eight task workload in NOW environments but without bus conflict network

Workload	Appl.	4 tasks		8 tasks	
		FIFO	RR	FIFO	RR
B1	FFT	0.325	0.527	0.265	0.704
	IS	0.738	0.862	0.892	0.508
	LU	0.014	0.150	0.131	0.459
B2	FFT	0.317	0.523	0.268	0.703
	IS	0.724	0.860	0.913	0.501
	MG	0.029	0.170	0.120	0.508
B3	FFT	0.901	0.886	0.730	0.732
	LU	0.040	0.382	0.459	0.420
	MG	0.098	0.273	0.360	0.520
B4	IS	0.895	0.911	0.912	0.457
	LU	0.049	0.239	0.146	0.435
	MG	0.157	0.160	0.113	0.486

TABLE 22. **Throughput for eight task workload in NOW environment but without bus conflict network**

	Batch	FIFO	Round Robin
B1	0.013614	0.081306	0.134448
B2	0.013900	0.130561	0.342844
B3	0.037799	0.284145	0.357830
B4	0.016304	0.108203	0.284697

8.3.4 Conclusion

We have shown the use of the Dimemas tool to better understand not only the behavior of individual applications (Section 8.3.1) but also to evaluate their behavior in multiprogrammed environments (Section 8.3.2).

Dimemas is a very useful tool. It runs on several platforms, and the processor time used for each different simulation is proportional to the number of communication of the workload and to the number of context switch. As the simulator does not have to rebuild the computation for the applications being simulated, the simulation time is, commonly, lower than the real execution time. Another goal of the simulator is that it is executed on sequential machines, freeing the parallel machines and the network for parallel application development. In fact, a real multiprocessor is needed if we really need very fine trace files, otherwise message passing libraries and workstations may be used to analyze parallel programs.

Because it is a simulation tool, it is very easy to implement different modules to analyze other issues: processor scheduling, file system caching, ATM communications,... It is also very useful to compare centralized control algorithms versus non centralized ones, just because centralized algorithms have been studied for shared memory machines. Centralized algorithms are not allowed for distributed memory machines and low level scheduling but, in some sense, may offer an optimum value.

We have shown that the effect of sharing resources can lead to high system throughput, but very important fairness problems arise depending on the application characteristics (granularity) and scheduling policies. The simulator interaction with Paraver help us in the understanding the effect of different scheduling algorithms has on the applications.

Some effort has to be directed to study cache pollution, context switch duration,... Some priority scheduling should be analyzed, and also the interaction of sequential programs in different nodes.

8.4 Scheduling analysis conclusions

In this chapter, we have presented a set of three different case studies to analyze different scheduling policies for distributed memory machine systems.

All over the chapter we have presented three different workloads, starting from synthetic workload, for a better understanding of the whole environment, to NAS programs utilization. In this later case, we have proposed a grouping mechanism to evaluate the influence of each program compared to reaming programs of the group.

We have been also evaluating different scheduling policies, from simple one as FCFS and Round-Robin, to some proposed by ourselves, BOOST. Although, from the logical point of view, it seems that our proposal should perform better, results demonstrates that Unix-like policies (also included in some analysis) performs better when using small time slices.

Different performance indices has been also presented in this chapter. These indices allow us to evaluate and compare the behavior of the system when running different applications and using different scheduling policies. These performance indices provides numerical information, that in some cases has been completed (to better understand the system behavior) using Paraver.

Using the different examples presented in this chapter we can conclude that processor sharing provides better performance and better resource utilization, compared to batch execution. The processor scheduling policy used is not required to be very complex, as basic policies have demonstrated good results, as FCFS and Round Robin. Priority base application obtains small benefits compared to previous ones, and those advantages are not maintained in the different examples.

CHAPTER 9

Conclusions

This work has presented several innovative tools, some of them in different areas: Dimemas and Paraver, for performance prediction and analysis, and Promenvir, for Robustness Management and Optimization.

Ten year duration for the work is relative large, but all the tools, concepts and methods are still valid and provide valuable information. This positive point of view, is demonstrated with the variety set of tools and methods used today for the same purposes as Dimemas, Paraver and Promenvir. Related work is today very prolific, but it was minimum when we have started working in this area.

Dimemas, with its proposed instrumentation method, the simulation with a very simple model for the parallel architecture, and the model for point to point communication and global operations, has been demonstrated a very useful tool for performance analysis and prediction. The validated results demonstrate the quality of the tool, and show that its is a useful tool for development of parallel application.

Paraver, with its interaction with Dimemas, and its design, is today a worldwide used tool. The internal architecture of the tool was designed in 1994, but the concept is still valid, and extendible. The current versions incorporates OpenMP analysis, and the tracing libraries include information related to source code. The author contribution to Paraver includes participation in the design process, and requirements specification. This is important, as its simple design is still valid for current researches, and incorporates functionalities (as the differentiation between logical and physical communications), that are necessary for application analysis on real platforms.

The concept of Promenvir is still under the tool named ST-ORM, distributed by EASi Engineering. The clear architecture enables the extension of the tool with new functionalities and the possibility to easily plug in any new method on robustness management. The contribution in Promenvir is the architecture design as well as the implementation of the internal system to distribute the workload on a Grid environment, in the epoch where Grid was an emerging concept. In fact, Promenvir is one of the first Grid killer application.

Another contribution is the methodology presented for application analysis. This methodology includes application monitoring (tracing), simulation, and visualization, as well as the analysis on the influence of the different parameters on the application execution.

We have also presented some analyses, using all the tools, on how low/medium level scheduling affects the execution of message passing applications. We have also proposed some scheduling algorithms that behave properly under multiprogrammed workloads. Current research in NANOS[74] and POP[75] project focuses on the analysis of process scheduling with prototype implementations of this type of scheduling policies.

Another contribution is the possibility to integrate all the tools, to perform a complete analysis of the system, and the easy extension of all the tools to include new possible functionalities.

9.1 Future work

The author current interest is focused in ST-ORM, and the tool will evolve to include intelligent methods to help in the analysis and to provide more insight of the models used in simulation.

Regarding Dimemas, on this ten years it has been extended to include a Distributed File System, and it is currently being extended to support analysis of application running on Grid Environments. This work focuses in finding the proper modelization for the MPI communication primitives.

Among the features we have been incorporated in the last years, we should mention a powerful 2D quantitative analysis module, mechanism to derive elaborated views by combining other basic views and a configuration file mechanism to capture knowhow on how to compute performance metrics. Ongoing work includes scalability issues, tracing packages for other type of systems (Java, commercial applications,...) and a non visual interface to Paraver. This late tool leverages the configuration file mechanism and is being used in conjunction with ST-ORM to develop models of applications and processor performance based on hardware counter samples.

Bibliography

[1] Fabrizio Petrini. *Communicaton Performance of Whormhole Interconnection Networks.* PhD thesis, Computer Science Dept. Pisa University, Genova-Udine. Italy, February 1997.

[2] Vicente Blano.Analisis, Prediccion y Visualizacion del Rendimiento de Metodos Iterativos en HPF y MPI. PhD thesis, Departamento de Electrónica e Computación. Universidad de Santiago de Compostela. España. Diciembre 2002.

[3] J.Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA. USA, 1992.

[4] A. Agarwal, A.K. Chandra, and M. Snir. *On communication latency in PRAM computations.* Technical Report RC 14973 (No. 66882), IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, 1989.

[5] A. Agarwal, A.K. Chandra, and M. Snir. *Communication complexity of PRAMS.* Theoretical Computer Science, pages 3-28, March 1990.

[6] P.B. Gibbons. *A more practical PRAM model*. In 1989 ACM Symposium on Parallel Algorithms and Architectures, pages 158-168, 1989.

[7] L.G. Valiant. *Handbook of Theoretical Computer Science*, volume A, chapter General purpose parallel architectures, pages 943/972. Elsevier and MIT Press, 1990.

[8] C.H. Papadimitriou and M. Yannakakis. *Towards an architecture independent analysis of parallel algorithms.* In 20th ACM Symposium on Theory of Computing, pages 510/513, 1988.

[9] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufman Publishers, San Mateo, CA, USA, 1992.

[10] Willian J. Dally. *Performance analysis of k-ary n-cube interconnection networks.* IEEE Transactions on Computers, 36(9):775-785, June 1990.

[11] Annant Agarwal. *Limits on interconnection network performance*. IEEE Transactions on Parallel and Distributed Systems, 2(4):398-412, October 1991.

[12] Amotz Bar-Noy and Shlomo Kipnis. *Multiple message broadcasting in the postal model*. Networks, 29(1):1-10, 1997.

[13] Luis Gargano and Adele A. Rescigno. *Fast collective communication by packets in the Postal model*. Networks, 31:67-79, 1998.

[14] Richard M. Karp, Abhijit Sahay, Eunice E. Santos, and Klaus E. Schauser. *Optimal broadcast ans summation in the LogP model*. In ACM Symposium on Parallel Algorithms and Architectures, pages 142-153, 1993.

[15] Abhijit Sahay. *Hiding communication costs in bandwidth-limited parallel FFT computation*. Technical Report UCB/CSD 92/722, Computer Science Division. University of California, Berkeley, 1992.

[16] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. *LOGP: Incorporating long messages into LogP model - one step closer towards a realistic model for parallel computation*. In Symposium on Parallel Algorithms and Architectures (SPAA), pages 95-105, Santa Barbara, CA, USA, July 1995.

[17] C. Andras Moritz and Mathew I. Frank. *LoGPC: Modelling network contention in message-passing applications*. Performance Evaluation Review. Special Issue, 26(1). June 1998.

[18] David Culler, Lok Tin Liu, Richard P. Martin, and Chad Yoshikawa. *Assesing fast network interfaces*. IEEE Micro, 16(1):35-43, February 1996.

[19] Andrea C. Dusseau, David E. Culler, Klaus Erik Schause, and Richard P. Martin. *Fast parallel sorting under LogP. Experience with the CM-5*. IEEE Transactions on Parallel and Distributed Systems, 7(8):791-805, August 1996.

[20] G. Iannello. *Efficient algorithms for the reduce-scatter operation in LogGP*. IEEE Transaction on Parallel and Distributed Systems, 8(9):970-982, September 1997.

[21] David Sundaram-Stukel and Mary K. Vernon. *Predictive analysis of a wayfront application using LogGP*. In 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP'99, Atlanta, GA, USA, May 1990.

[22] David B. Skillicorn, Jonathan M.D. Hill, and W.F. McColl. *Questions and answers about BSP*. Scientific Programming, 6(3):249-274, 1997.

[23] Leslie G. Valiant. *Bulk synchronous parallel computers*. Technical Report TR-08-89, Aiken Computation Laboratory. Harvard University, 1989.

[24] Leslie G. Valiant. *A bridging model for parallel computation*. Comminications of the ACM, 33(8):103-111, August 1990.

[25] Xingfu Wu. *Performance Evaluation, Prediction and Visualization of Parallel Systems*. Kluwer Academic Publishers, 1999.

[26] E. Hagersten, A. Landinn, and S. Hairidi. *DDM - A cache only memory architecture*. IEEE Computer, pages 44-54, September 1992.

[27] Thomas Fahringer. *Estimating and optimizing performance for parallel programs*. IEEE Computers, 28(11):47-56, November 1995.

[28] M.J. Clement and M.J. Quinn. *Analytical Performance prediction on multicomputers.* In Supercomputing 93, pages 886-83, 1993.

[29] Richard P. Larowe Jr and Carla Schlatter Ellis. *Experimental comparison of memory management policies for NUMA multiprocessors.* ACM Transactions on Computer Systems, 9(4):319-363, November 1991.

[30] Joe Truman and John L. Hennessy. *Evaluating the memory overhead required for COMA architectures.* Computer Architecture News (Special Issue ISCA´21 Proceedings), 1994.

[31] T. Hey, A. Dunlop, and E. Hernández. *Realistic parallel performance estimation.* Parallel Computing, 23:5-21, 1997.

[32] K.S. Trivedi and P. Heidelberger. *Queuing network models for parallel processing with asynchronous tasks*. IEEE Transactions on Computers, C-32:15-31, January 1982.

[33] Alois Ferscha. *A petri net approach for performance oriented parallel program design.* Journal of Parallel and Distributed Computing, 153):188-206, July 1992.

[34] Alois Ferscha and James Johnson. *Performance Prediction of Dyamic Task Structures with N-MAP.* Kluwer Academic Press, 1995.

[35] Shirley Browne, Jack Dongarra, and Kevin London. *Review of Performance analysis tools for MPI parallel programs*. www.cs.utk.edit/~browne/perf-tools-review, December 1997.

[36] R.J. Allan, Y.F. Hu, and P. Lockey. *A survey of Parallel Numerical Analysis Software: edition 2. Parallel Application Software on High-Performance Computers.* Technical Report DL-TR-99-01, CLRC Daresbury Laboratory, 1999. www.dl.ac.uk/TCSC/Subjects/Parallel_Algorithms/lib_survey.

[37] J.C. Yan and S.R. Sarukkai. *Analyzing parallel program performance using normalized performance indices and trace transformation techniques.* Parallel Computing, 22(9):1215-1237, 1996. www.nas.nasa.gov/Groups/Tools/Projects/AIMS.

[38] Louis Lopez. *NTV - the NAS trace visualizer.* science.nas.nasa.gov/Groups/Tools/Projects/NTV.

[39] W. Williams, T. Hoel, and D. Pase. *Programming Environments for Massively Parallel Distributed Systems, chapter The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D*, pages 333-345. Birkhauser Verlag, 1994.

[40] Jens Volkert, Dieter Kranzlmuller, and Siegfried Grabner. *The tools of the monitoring and debugging environment.* In 2nd European School of Computer Science, ESPPE'96, pages 169-172, Alpe d'Huez, France, April 1996. www.gup.uni-linz.ac.at/research/debugging/mad.

[41] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. *MEDEA - a tool for workload characterization of parallel systems.* IEEE Parallel and Distributed Technology, 3(4):72-80, 1995. mafalda.unipv.it/Laboratory/research/Medea/index.html.

[42] Daniel Reed et al. Pablo: *Scalable performance tools*. www-pablo.cs.uiuc.edu.

[43] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jerey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. *The Paradyn parallel performance measurement tools*. IEEE Computer, 1995. www.cs.wisc.edu/ paradyn.

[44] Christopher Carothers, Brad Topol, Richard Fujimoto, John Stasko,and Vaidy Sunderam. *Visualizing parallel simulations that execute in network computing environments*. Future Generation Computer Systems,15(4):513-529, July 1999.

[45] M.T. Heath and J.A. Etheridge. *Visualizing the performance of parallel programs*. IEEE Software, 8(5):29-39, September 1991.

[46] Satish Balay, Kris Buschelman, William Gropp, Dinesh Kaushik, Matt Knepley, Lois Curfman McInnes, Barry Smith, and Hong Zhang. *PETSc users manual*. Technical Report ANL-95/11, Mathematics and Computer Science Division. Argonne National Laboratory, May 2002.

[47] O. Zaki. *Toward scalable performance visualization with jumpshot*. International Journal of High Performance Computing Applications, 13(3):277-288, Fall 1999.

[48] Brad Topol, John Stasko, and Vaidy Sunderam. *Integrating visualization support into distributed computing systems.* In 15th International Conference on Distributed Computing Systems, pages 19-26, Vancouver, B.C., May 1995. http://www.cc.gatech.edu/gvu/people/Phd/Brad.Topol/pgpvm.html.

[49] V.S. Sunderam. *PVM: A framework for parallel distributed computing*. Concurrency: Practice and Experience, 2(4):315-339, December 1990. www.csm.ornl.gov/pvm/pvm home.html.

[50] B.J.N. Wylie and A. Endo. *The Annai/PMA performance monitor and analyzer.* In 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'96), pages 51-58, San Jose, CA, USA, February 1996. www.cscs.ch/Ocial/TechReports/1995/CSCS-TR-95-05.ps.gz.

[51] P. Kacsuk, J.C. de Kergommeaux, E. Maillet, and J.M. Vincent. *Parallel Program Development for Cluster Computing, Methodology, Tools and Integrated Environments, chapter The Tape/PVM Monitor and the PROVE Visualisation Tool*, pages 291-303. Nova Science Publishers, Inc, 2001. www.lpds.sztaki.hu.

[52] ANL/MSU MPI implementation. *MPICH-A portable implementation of MPI*. www-unix.mcs.anl.gov/mpi/mpich/.

[53] W.E. Nagel, A. Arnold, M. Weber, H-C. Hoppe, and K. Solchenbach. *VAMPIR: Visualization and analysis of MPI resources.* Supercomputer 63, 12(1):69{80, January 1996. www.pallas.com/e/products/vampir/index.htm.

[54] IBM Corporation. *Using the Parallel Operating Environment. Operation and Use.* Volumen 1 Volumen 2.

[55] LAM / MPI Parallel Computing. Open Systems Laboratory. University of Indiana. *XMPI - A Run/Debug GUI for MPI*. www.lam-mpi.org/software/xmpi/.

[56] T. Kunz and D. Taylor. *Visualizing PVM executions*. In 3rd PVM Users' Group Meeting, Pittsburgh, USA, May 1995. www.netlib.org/utk/icl/xpvm/xpvm.html.

[57] Walter Kuhn and Hermart Buckhart. *Performance modelling for parallel-skeletons.* In ESPPE'96, 1996.

[58] M. Crovella. *Performance Prediction and Tuning of Parallel Programs*. PhD thesis, University of Rochester, 1994.

[59] W. Meira Jr., T. LeBlanc, and A. Poulos. *Waiting time analysis and performance visualization in Carnival*. In SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools, pages 1-10, 1996. www.cs.rochester.edu/u/leblanc/prediction.html.

[60] K. Kubota, K. Itakura, M. Sato, and T. Boku. *Practical simulation of large-scale parallel programas and its performance*. In Lecture Notes in Computers Science, number 1470 in LNCS, pages 244-254. Springer Verlag, 1998.

[61] A.J.C. van Gemund. *Performance Modeling of Parallel Systems*. PhD thesis, Department of Information Technology and Systems. Delft University of Technology, The Netherlands, April 1996. 156 pp.

[62] A.J.C. van Gemund. *Symbolic performance modeling of parallel systems*. IEEE Transactions on Parallel and Distributed Systems, 13(11), November 2002. rama.pds.twi.tudelft.nl/gemund/Pamela.

[63] D.J. Kerbyson, E. Papaefstatuiou, J.S. Harper, S.C. Perry, and G.R. Nudd. *Is predictive tracing too late for HPC users*. In R.J. Allan, M.F. Guest, D.S. Henty, D.Ñicole, and A.D. Simpson, editors, HPCI'98 Conference, High Performance Computing, pages 57-67. Plenum/Kluwer Publishing, 1999. www.dcs.warwick.ac.uk/hpsg/html/htdocs/public/pace.html.

[64] A.M. Alkindi, D.J. Kerbyson, and G.R. Nudd. *Dynamic instrumentation and performance prediction of application execution.* In High Performance Computing and Networking (HPCN2001), volume 2110 of Lecture Notes in Computer Science, pages 313-323, Amsterdam, June 2001. Springer-Verlag.

[65] B. Armstrong and R. Eigenmann. *Performance forecasting: Towards a methodology for characterizing large computational applications.* In International Conference on Parallel Processing, pages 518-525, August 1998. paramount.www.ecn.purdue.edu/ParaMount.

[66] Engineous Software. http://www.engineous.com.

[67] LMS International. http://www.lmsintl.com.

[68] Altair Engineering. http://www.altair.com

[69] Samtech group, http://www.samtech.fr

[70] VK Naik SK Setia and MS Squillante. *Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments.* Supercomputing 93 pp 824-833

[71] D Bailey JBarton TLasinksi and H Simon. *The nas parallel benchmarks* Techical report, NASA Ames Research Center Moett Field CA. July 1993.

[72] PARKBENCH Committee. *Public international benchmarks for parallel computers.* Technical Report CS 93-123, Computer Science Departament University of Tenesse, Knoxville, Tenesse. November 1993.

[73] A. Silberschatz, P. Galvin, G. Gagne. *Operating systems (6th edition)*, Ed. Limusa-Wiley, 2002.

[74] CEPBA-UPC. http://www.cepba.upc.es/nanos

[75] CEPBA-UPC. http://www.cepba.upc.es/pop

9.2 Author bibliography

[76] Sergi Girona, Toni Cortés, Jesús Labarta, Vincent Pillet, Andrés Pérez, Eva López. *Effect of short term scheduling on message passing multiporogrammed systems.* Deliverable OPS4A of the project Basic research APPARC, August 1994.

[77] Jesús Labarta, Sergi Girona, Toni Cortes, Judit Gimenez, Cristina Pujol and Luis Gregoris. *The Paros Operating System Microkernel*. CEPBA/UPC Report No. RR-94/05 June 1994.

[78] Sergi Girona, Toni Cortes, Jesús Labarta. *Evaluación de políticas de planificación de bajo nivel para multiprocesadores.* Technical Report UPC-DAC-1995-21, June 1995

[79] Jesús Labarta, Sergi Girona, Toni Cortés. *Analyzing scheduling policies using Dimemas* . 3rd Workshop on environment and tools for parallel scientific computation. Faverges de la Tour (Francia), August 1996. Publicado en Parallel Computing, vol. 23 (1997) pp 23-34. Ed. Elsevier.

[80] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortés, Luis Gregoris. *DiP : A Parallel Program Development Environment*. 2nd International EuroPar Conference (EuroPar'96), Lyon (Francia), August 1996

[81] Sergi Girona and Jesús Labarta. *Sensitivity of Performance Prediction of Message Passing Programs.* 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Monte Carlo Resort, Las Vegas, Nevada, USA, July 1999.

[82] Vincent Pillet, Jesús Labarta, Toni Cortés, Sergi Girona. *PARAVER: A Tool to Visualize and Analize Parallel Code*". The 18th Technical Meeting of WoTUG, Manchester (Inglaterra), April 1995.

[83] Sergi Girona, Jesús Labarta y Maite Ortega. *Monte Carlo Workload Scheduling.* Capítulo 9 del libro "Computational Stochastic Mechanics in a Meta-Computing Perspective" de la editorial CIMNE. ISBN - 84-89925-04-6

[84] Sergi Girona, Santi Bello, Jesús Labarta. *The Queue System within PHASE*. S HPCN'99, Amsterdam, April 1999

[85] *Arquitectural Design Document*, deliverable of the project Promenvir, Esprit 20189, November 1996

[86] *Project Specification*, deliverable of the project PHASE, Esprit 27238, November 1999