University of Southampton

PERFORMANCE VISUALIZATION OF PARALLEL PROGRAMS

by

Oscar Naím D'Paola

A thesis submitted for the degree of
Doctor of Philosophy

Faculty of Engineering
Department of Electronics and Computer Science

May 25, 1995

Supervisor: Prof. A.J.G. Hey

University of Southampton

ABSTRACT

Faculty of Engineering

Department of Electronics and Computer Science
Doctor of Philosophy

PERFORMANCE VISUALIZATION OF PARALLEL PROGRAMS

by Oscar Naím D'Paola

Supervisor: Prof. A.J.G. Hey

Performance is a critical issue in current massively parallel processors. However, delivery of adequate performance is not automatic and performance evaluation tools are required in order to help the programmer to *understand* the behaviour of a parallel program. In recent years, a wide variety of tools have been developed for this purpose including tools for monitoring and evaluating performance and visualization tools. However, these tools do not provide an abstract representation of performance. Massively parallel processors can generate a huge amount of performance data and sophisticated methods for representing and displaying this data (e.g. visual and aural) are required. Current performance views are not scalable in general and do not represent an abstraction of the performance data.

The Do-Loop-Surface display is proposed as an abstract representation of the performance of a particular do-loop in a program. It has been used to improve the performance of several algorithms on different hardware platforms (e.g. CM-5, Meiko CS-2, IBM SP2). The examples demonstrate that the Do-Loop-Surface display is an useful way to represent performance. It is implemented using AVS (Application Visualization System), a standard data visualization package. The use of scientific visualization tools such as AVS to display performance data, is becoming a very powerful alternative to support performance analysis of parallel programs. The Do-Loop-Surface (DLS) display is presented in this thesis as an example on how a data visualization tool can be used to define new abstract representations of performance, helping the user to analyze complex data potentially generated by a large number of processors.

Additionally to this main contribution, our experience developing the performance tool ANDES is presented in this thesis as well as our studies related to the invasiveness effects that performance instrumentation can generate.

Acknowledgements

I am very grateful to the following people, who in some way or another helped me to successfully finish this PhD Thesis: Alistair Dunlop, Nick Floros, Stephen Hellberg, John Merlin, Ivan Wolton, Richard Cloke, Emilio Hernández, Mario Dantas, Mark Rogers, Flavio Bergamaschi, Vladimir Getov, Roger Hockney, Ed Zaluska, Denis Nicole and the Concurrent Computation Group in general. I am also very grateful to Peter Ossadnik and Malcom Brown for their very useful comments and to GMD for providing access to a CM-5. Special thanks also for Rusty Lusk at Argonne National Laboratory for helping me with the final experiments of my Thesis and for providing access to an IBM SP2, as well as for the excellent treatment I had during my whole stay at ANL. I am also grateful to the British Council and Fundayacucho, my sponsors, for their economical support. Many thanks also to Chris Collier, one of the most wonderful secretaries I have ever met, and to all the secretaries of the department for their support and friendship.

Finally, I would like to say that this Thesis would not have been possible without the advice of Tony Hey, who always helped and directed me in the best way. Thanks to him, I had the opportunity to interact with other researchers all over the world and more important, I learned how to be a good researcher. Thanks Tony for your example!

To all of you folks, thanks a lot!

Dedicatoria:

Esta tesis de Doctorado esta dedicada a las dos mujeres mas importantes de mi vida: María Carrillo, mi novia y futura esposa, y Teresita D'Paola, mi mamá; y a Venezuela, mi querida patria.

Contents

List of Figures

List of Tables

1 Introduction

Massively Parallel Processor machines are a reality and they represent a revolution in terms of computational power, promising Teraflop/second performance. The need for this computational power has been widely justified by the *Grand Challenge* problems, which include problems from molecular dynamics to weather simulation and prediction. The current computational power of these massively parallel systems can outperform state of the art serial supercomputers, and they are far less expensive.

However, this advance in hardware technology has not been matched by software technology. It is clear that more powerful massively parallel systems will be built, but now the problem is to make software capable of exploiting this immense computational power. Parallel processing promises enormous power at a cost - *understanding*. Parallel computations involve highly complex and little understood behaviour and this lack of understanding prevents efficient use of this power [1].

The scenario that a parallel programmer has to face is much more complex than the equivalent one for sequential programmers. There are problems related to the parallelization of the algorithm itself (e.g. how to divide a process in tasks and how to map them on the physical processors) and problems related to the optimal use of the parallel resources (e.g. load balance, communications, use of cache, memory contentions). In order to produce a high performance application, the programmer requires a very good knowledge of the application and, in some cases (e.g. performance programming [2]), a good knowledge of the underlying parallel hardware as well as the software platform available (e.g. how the compiler actually works).

Thus, the degree of expertise required for a parallel programmer in order to produce a high performance application is higher than for a sequential programmer. Programming parallelism can be very painful and frustrating. In addition, debugging a parallel program and searching for performance bottlenecks is a difficult and time-consuming process.

For all of these reasons, massively parallel machines have acquired a reputation for being difficult to program, but this is largely an artifact of the lack of programming tools comparable to those that are available on serial machines [3]. Parallel performance monitoring tools are needed in order to understand the *behaviour* and inner workings of programs, and to help identify possible trouble spots such as communications bottlenecks. They may be used to evaluate and compare the performance of similar programs and variants of algorithms, and are also an aid to debugging [4]. Many projects from universities and research institutions have developed and implemented tools and environments to ease the use and programming of parallel systems. Dozens (hundreds?) of parallel programming tools are being developed and some of them are becoming commercially available [5]. However, most of them are either integrated in a programming environment or they are built only for one special monitor system.

Since *performance* is one of the most important requirements for the use of parallel computers, and since initial implementations of parallel programs typically yield

disappointing performance, *tuning* to improve performance is thus a significant part of the parallel programming process. A major factor contributing to the large amount of skilled effort typically required to achieve good parallel program performance is the shortage of good performance analysis and evaluation tools. In the absence of such tools, performance problems must be identified through a combination of guesswork, folklore, and application-specific instrumentation [6]. Tuning the performance of parallel programs is a crucial but difficult task which requires a good understanding of the program to be tuned. The aim of performance monitoring and visualization tools is to give this understanding to the programmer [7].

Performance visualization is the use of graphical display techniques to present an analysis of performance data for an improved understanding of complex performance phenomena [8]. Performance visualization systems for parallel programs have been helpful in the past and they are commonly used in order to improve parallel program performance. However, despite the advances that have been made in visualizing scientific data, techniques for visualizing performance of parallel programs remain *ad hoc* and performance visualization becomes more difficult as the parallel system becomes more complex [8].

Popular views like space-time/Feynman diagram (ParaGraph [9, 10]) or event timelines (Express Etool [11]) often provide some insight into program behaviour. Unfortunately, for these tools to be truly useful in the domain of large scale parallel machines they must be extended to include abstract high level views [12]. Massively parallel processors can generate a huge amount of performance data and sophisticated methods for representing and displaying this data are required. Visualization must be carefully used in order to get useful results [1].

In general, current performance views are not scalable and do not represent an abstraction of the performance data. If performance visualization is to become an integral tool in parallel performance evaluation, it must be based on a formal foundation that relates abstract performance behaviour to visual representations [8]. Two examples of the research effort in this field are PARASEER [8] and the work done by Carter [14, 15]. PARASEER is a parallel performance visualization project carried out by the University of Oregon, as an exploratory research study of next-generation parallel performance visualization technology. On the other hand, Carter proposes to represent computation intensive algorithms or programs as a (usually 3-dimensional) solid. In this way, the problem of tuning the code for a computer, or of parallelizing the code, can be visualized as cutting the solid into smaller pieces, where surface area corresponds to communication and volume to computation [15].

Another example is the development of a set of visualization tools for HPF (High Performance Fortran [16, 17]). HPF promises to be an attractive abstraction for Fortran programs on parallel and massively parallel computers. Since HPF does not contain explicit message passing, the opportunity arises to visualize program execution at a higher level than that of existing tools.

Parallel performance visualization offers a wide range of research opportunities, and one of these is the designing and development of new *scalable performance data representations*. These data representations should have the following features:

[1] Drawing useful pictures is difficult. As Miller states in [13], visualizations should *guide*, not *rationalize*. To *guide* means that it leads you to discover things that you did not already know and *rationalize* means that it lets you illustrate things that you already know.

- Scalability (i.e. number of processors, problem size).
- Higher level of abstraction than that of existing views.
- Usefulness. A performance view must be useful in the process of understanding parallel program behaviour.
- Incorporation and integration of state of the art technologies including: application domain expertise, human-computer interaction (HCI), visual perception, and graphic design.

The Do-Loop-Surface representation of the performance of a particular do-loop in a program is presented in this thesis as an alternative tool that provides an abstract representation of performance, using a commercial scientific data visualization tool (AVS, [18]) for this purpose [19, 20]. One advantage of this approach is that no tool development is required and that every feature of the data visualization tool can be used for further data analysis. AVS also has its own built-in analysis capabilities. Many analysis functions are available and the methods are scalable with respect to data set sizes and extensible in terms of applying additional analysis (or programming user-specific analysis), and support multiple domains of analysis (e.g. temporal, spatial, and frequency domains). The objective of multiple-domain analysis methods is to selectively focus on a particular aspect of the performance while hiding the contributions from other aspects [21].

In this thesis, several state of the art monitoring and visualization tools described in the literature are presented (after a short review of the state of the art in parallel architectures, message passing systems and emerging standards given in Chapter 2), including PABLO [22], ParaGraph [9], and SIMPLE [23] (Section 2.3). Additionally, ANDES [24], a recent development, is also described (Chapter 3). The degree to which these monitoring tools perturb the behaviour of the application being analyzed, i.e. their tool *invasiveness*, is described in Chapter 4. Chapter 5 describes in detail the Do-Loop-Surface abstraction and its main goals and advantages. Chapter 6 illustrates several tool evaluation tests and our experience at Argonne National Laboratory using DLS displays. Finally, Chapter 7 shows the conclusions of this research.

The main original contributions of this work are presented in Chapters 3, 4, 5, and 6. They include three main issues: *ANDES*, which was originally developed as an MSc Thesis at Universidad Simón Bolívar but enhanced at the University of Southampton as part of my Interim Thesis (presented in the 16th Technical Meeting of the World Occam and Transputer User Group [25], 1993); experiments with *invasiveness* (presented in IFIP'94 [26]); and the *DLS* abstract representation of parallel program performance, first presented at the High Performance Computing and Networking Conference held in Munich in 1994 [19], and to be published in *Concurrency: Practice and Experience* [20]. Other publications of research involving DLS are HPCN Europe'95 [27] and the Second Workshop on Environments and Tools for Parallel Scientific Computing 1994 [28].

"The increasing complexity of parallel computing systems has brought about a crisis in parallel performance evaluation and tuning. Although there have been important advances in performance tools in recent years, we believe that future parallel performance environments will move beyond these tools by integrating performance instrumentation with compilers for architecture-independent languages, by formalizing the relationship between performance views and the data they represent, and by automating some aspects of performance interpretation.

Indeed, the grand challenge problem in computational software for scalable, parallel systems is the development of tools that can routinely (and, ideally, automatically) produce high performance applications" [8].

2 Overview and State of the Art

The purpose of this Chapter is to briefly describe the state of the art in parallel architectures, languages, and performance tools. We describe the most interesting parallel machines used in our experiments as well as parallel languages and message passing systems. The last section of this Chapter, reviews current performance analysis tools including PABLO, ParaGraph and SIMPLE.

2.1 Review of Parallel Machines

The following three sections describe three state of the art parallel computers: Thinking Machines CM-5, Meiko CS-2, and IBM SP2.

2.1.1 CM-5

The Connection Machine CM-5 is a MIMD supercomputer that combines ease of use and high performance for programmers working on large, complex, data-intensive applications [29]. The CM-5 supports both data parallel programming and message-passing programming. The CM's data parallel compilers (CM Fortran, C*, *Lisp) present the user with a global address space and a single thread of control. The CMMD communications library, callable from Fortran 77, C, C++, CM Fortran and C*, provides fast communication between independent tasks.

The CM-5 may have configurations with a large number of processors (e.g. 512). Each of the CM-5 parallel processing nodes has its own memory. Nodes can fetch from the same address in their memories to execute the same instruction, or from individually chosen addresses to execute independent instructions. These processing nodes are supervised by a control processor. The system administrator can divided the nodes into groups, known as partitions. There is a separate control processor, known as a partition manager, for each partition. Each user process executes on a single partition, but can exchange data with processors on other partitions. The CM-5 processing node is a standard RISC microprocessor. This microprocessor may optionally be augmented with a special high-performance hardware arithmetic accelerator that uses a wide data path, deep pipelines, and large register files to improve peak computational performance. The node memory subsystem consists of a memory controller and either 8 Mbytes, 16 Mbytes, or 32 Mbytes of DRAM memory. The special arithmetic hardware consists of four vector units (VUs), each of them with a memory bank of 8 or 32 Mbytes. Each VU can deliver 40 MFlop/s peak 64-bit floating point performance.

Every control processor and parallel processing node in the CM-5 is connected to two communication networks, the Data Network and the Control Network. The control network is used for operations that involve all the nodes at once, such as synchronization operations and broadcasting. The data network is used for bulk data transfers where each item has a single source and destination. External networks, such as Ethernet and FDDI, can also be connected to a CM-5 via the control processors. The topology

of the data network is a *fat tree*, so called because some branches are *fatter* (of higher bandwidth) than others. One advantage of this topology is that traffic between two partitions does not interfere with traffic internal to a third partition.

In terms of software, the CM-5 operating system, CMOST, is an enhanced version of the UNIX operating system. The CM-5 also provides tools for parallel programming, debugging and tuning, such as Prism and the CMAX Fortran 77 to CM Fortran translator. Prism is a programming environment that integrates debugging, profiling, and other useful tools in a convenient windowed environment. CMAX provides a convenient migration path for serial programs onto the CM-5, since it helps in the translation of serial Fortran 77 code to Connection Machine Fortran (CM Fortran).

2.1.2 Meiko CS-2

The CS-2 is a scalable multicomputer which combines vector and concurrent processing capabilities [30]. CS-2 is a distributed global memory architecture and each processing element is a scalar or super-scalar SPARC processor running Solaris, the industry standard operating system. Each CS-2 vector PE consists of a SPARC scalar unit, a communications processor and 2 Fujitsu VP vector units sharing a three ported memory system. Although the SPARC itself offers a 40 MFlop/s processing power, supercomputing performance is obtained by exploiting the vector processors which combined can deliver 200 MFlop/s. Exercising a small set of such nodes in a parallel configuration can deliver GFlop/s performance. A few thousand processors would be needed to scale performance to TeraFlop/s range. Processors are connected by a multi-stage switching network and the communication itself is handled by specialized communication processors to achieve low communication latencies and high bandwidth.

The CS-2 application development environment includes compilers for Fortran 77, ANSI C, Fortran 90 and High Performance Fortran together with a wide variety of tools for instrumenting, analyzing, debugging and parallelizing programs. This toolset runs either on the system or on networked SPARC workstations. In terms of message passing, Meiko supports the standard interfaces PVM and PARMACS on the CS-2, together with their own CS-Tools. Intel NX/2 compatibility libraries provide portability from iPSC and Paragon systems.

CS-2 systems support a wide range of tools designed to assist in porting and parallelizing applications codes. Tools include both compiler tools for parallelizing applications and utilization tools for measuring performance. Hardware support for collecting performance data is provided by the CS-2 control network. The CS-Tools multi-process debugger *pdb* provides a *dbx* style interface to debugging multi-process programs. *pdb* allows the user to set break and watch points, trace, inspect and modify variables. Another tool available for the CS-2 is the TotalView debugger [31]. Totalview is part of a suite of software development tools for debugging, analyzing, and tuning the performance of programs, including multiprocess programs. Some of the main features of Totalview are: provides an X Windows graphical interface, TotalView allows to debug remote and distributed programs over the network, and it is possible to write source code fragments within TotalView and insert them temporarily into the program we are debugging. Additionally, many standard debugging features are provided (e.g. breakpoints) plus a number of useful functions for examining and manipulating data. Visualization of parallel performance data is provided under AVS.

2.1.3 IBM SP2

The IBM SP2 is the second generation of IBM's scalable POWERparallel series, for serial and parallel application execution [32]. Each node has substantial disk space and memory (from 64MB to 2GB of memory and from 1 to 8 GB internal disk). This allows local physical storage memory commensurate with the processing power of the node, thus allowing large problems and problem segments to run on individual nodes.

The SP2 nodes are RISC-based processors (from 4 up to 128) running full AIX/6000 software, providing a growth in peak performance from 1 to 34 GigaFlop/s. The multistage packet switching network supports high performance communication between processor nodes and the point-to-point communication time between any two nodes is independent of the relative node positions within the SP2 complex of nodes.

The internode network architecture provides relatively high bandwidth as well as good behaviour with respect to irregular communication patterns, thus permitting the efficient use of advanced algorithms that adapt to the structure of the problem. Additionally, the system includes a large, high performance I/O architecture that provides a significant number of connections to the external I/O network, thus allowing a very high aggregate bandwidth to secondary storage and other external devices and network connections.

The SP2 will support popular parallel application interfaces such as Express, PVM, MPI, and Linda. Additionally, the MPL message passing library is provided by IBM for efficient low level communication. FORGE 90 is available to assist in the conversion of existing serial code and a full version of Fortran 90 is also provided. High Performance Fortran is supported as well as the DB2 database package.

2.2 Review of Message Passing Systems and Emerging Standards

The following three sections of this Chapter describe: PVM, the current *de facto* standard for message passing, MPI, the emerging standard, and High Performance Fortran.

2.2.1 PVM

PVM, or Parallel Virtual Machine [33], is a software package that permits a heterogeneous collection of serial, parallel, and vector computers on a network to appear as one large computing resource. User programs written in C or Fortran are provided with access to PVM through the use of calls to PVM library routines for functions such as process initiation, message transmission and reception, synchronization and global operations.

PVM supports heterogeneity at three levels: application (subtasks may use the architecture best suited for their solution), machine (computer with different data formats, different architectures, and different operating systems), and network (different network types such as FDDI and ethernet). One of the major advantages of PVM is its portability. PVM runs on a wide range of parallel architectures and systems (more than 25 at the time of writing) including Intel iPSC/860, Paragon, CM-5, Meiko CS-2, IBM SP2, and clusters of workstations. Version 3 is also portable to non-Unix machines and multi-processor computers.

In terms of communication, PVM provides routines for packing and sending messages between tasks. The model assumes that any task can send a message to any other PVM task, and that there is no limit to the size or number of such messages. In practice, we have found that packing and unpacking data can be very costly and can seriously affects performance. The PVM communication model provides asynchronous blocking send and receive, and non-blocking receive functions.

An important advantage of PVM is that it is available in the public domain. Additionally, there are tools for visualization and debugging of PVM programs such as Xab [34, 35], and XPVM [36]. Xab, or X-window Analysis and deBugging, is a tool for run time monitoring of PVM programs. Using Xab, PVM programs can be instrumented and monitored. Xab uses PVM to monitor PVM programs, making Xab also very portable. Xab consists of three main components: a user library, which provides instrumented versions of the PVM calls; a monitoring program, which runs as a PVM process and gathers monitor events in the form of PVM messages; and an X-windows front end, that displays information graphically about PVM processes and messages.

XPVM is a graphical console and monitor for PVM. It provides a graphical interface to the PVM console commands and information, along with several animated views to monitor the execution of PVM programs. These views provide information about the interactions among tasks in a parallel PVM program, to assist in debugging and performance tuning. To analyze a program using XPVM, a user need only compile their program using the PVM library (version 3.3 or later) which has been instrumented to capture tracing information at run-time. Then, any task spawned from XPVM will return trace event information, for analysis in real time, or for postmortem playback from saved trace files.

2.2.2 MPI

The international Message Passing Interface (MPI) initiative [37, 38], was founded by Oak Ridge National Laboratory, the Center for Parallel Computing at Rice University and the University of Southampton. The MPI Forum is a collection of major vendors and users from around the world. The main goal of this effort has been to define a message passing interface which could be efficiently implemented on a wide range of parallel and distributed systems. MPI is intended to be the standard message passing interface for parallel applications and library programming. The basic content of MPI is point-to-point communication between pairs of processes and collective communication within groups of processes. MPI also provides more advanced message passing features which allow the user to manipulate process groups, topological structures, and support the development and utilization of parallel libraries.

MPI data structures allow the user to send and receive messages with complicated storage patterns without the need to copy data in to and out of message buffers, and allow an implementation to optimise communications with such storage patterns. Issues such as parallel input/output and remote read/write are not currently included in MPI. The MPI Forum intends to cover further topics in a second phase begining in 1995.

Communications within MPI are performed within a *communication context* which insulates messages in different parts of the program from one another. The defining property of a context is that a message send in one context can only be received in that same context. The communication context is the primary mechanism for isolation

of messages in different libraries. The point-to-point message passing routines are the core of the MPI standard. These routines are *send* and *receive*. There are 4 types of these routines: standard, synchronous, ready-receive, and buffered. MPI also provides non-blocking or immediate return send and receive primitives, as well as collective communication routines such as *barrier*, *broadcast*, *gather*, *scatter*, *reduce*, and *total exchange*. Since many applications have a geometrical background, MPI also allows the definition of a geometrical arrangement of processes as well as graph topologies.

At this moment, several public domain versions of MPI are available and some are commercially supported implementations. We expect MPI to become the most used message passing interface in the near feature.

2.2.3 HPF

High Performance Fortran, HPF [39], is a new data parallel language for writing efficient portable parallel programs. HPF consists of a set of extensions to Fortran 90 which facilitates data-parallel computations on multi-processor architectures. HPF provides directives for specifying how program data should be distributed over the network of processors in order to produce the most efficient result. HPF goals include high performance on MIMD and SIMD computers with non-uniform memory access costs and code tuning for various architectures.

Some of the most interesting features of HPF are:

- Data alignment and distribution to increase locality of reference.
- Assertion that the statements in a particular section of code do not exhibit any sequential dependencies.
- Declaration of rectilinear processor arrangements.
- FORALL statements, designed primarily for data parallel programming. Briefly, if you have, e.g.: $FORALL(i = 1 : n)a(i) = 0.5 * (a(i-1) + a(i+1))$, the RHS (right hand side) is fully evaluated for *all* index values $i = 1 : n$, and then the assignment is performed to corresponding elements of the LHS (left hand side). Therefore, if the RHS involves any assignment variables, it always uses the *old* values of the variables before they are updated by the assignment. Therefore the semantics of FORALL are data parallel by definition.
- Pure procedures (i.e. procedures with no side effects).
- Extended intrinsic functions and standard library (e.g. combining-scatter functions, sorting functions, bit-manipulation functions).
- Extrinsic procedures (interface to procedures written in other paradigms such as message passing, and interface with other languages).

The HPF directives are structured comments that suggest implementation strategies or assert facts about a program to the compiler. They may affect the efficiency of the computation performed, but do not change the semantics of the computation. The form of the HPF directives has been chosen so that a future Fortran standard may choose to include these features in the language by deleting the initial comment header. HPF has no I/O extensions at all. Of course, an HPF implementation of Fortran's I/O statements could perform the I/O in parallel in suitable circumstances.

2.3 Review of Performance Analysis Tools

2.3.1 PABLO

PABLO is a performance analysis environment developed at the University of Illinois, designed to provide performance data capture, analysis, and presentation (both sound and graphics) across a wide variety of parallel systems [22]. PABLO is a toolkit for the construction of performance analysis environments. It consists of two primary components: portable software instrumentation and portable performance data analysis, with a trace data meta-format coupling the instrumentation with data analysis.

- Portable software instrumentation: this is designed to support interactive specification of source code instrumentation points. The software instrumentation can be used to gather performance information from either the system or the application code. The initial architectural targets are: Thinking Machines CM-2, CM-5 and the Intel Paragon XP/S.
- Portable performance data analysis: this consists of a set of data transformation modules that can be graphically interconnected to form an acyclic, directed data analysis graph. Performance data flows through the graph nodes and is transformed to yield the desired performance metrics.
- Trace data meta-format: the performance data format has no embedded semantics (i.e. there are no predefined event types or data sizes).
- Performance data analysis software: this is an object-oriented software written in C++ designed to be easily ported to new machine architectures.

The PABLO design philosophy attempts to address the three issues of:

- *Portability.*
- *Scalability.*
- *Extensibility.*

Portability and ease of use are critical to the acceptance of new performance tools. Scalability is a key characteristic of the new generation of massively parallel systems by adding processors one can incrementally increase performance without replacing existing hardware or changing the underlying software. A performance environment must be extensible, allowing its users to interact with the system and add new features to it. If a tool's functionality is too limited, it will not be used. If a tool is too general and does not support common cases in obvious ways, it will also not be used. The environment must also be able to recognize different kind of users (e.g. novice, intermediate and expert).

The PABLO environment presently includes:

- A Motif-based interface for the specification of source code instrumentation points (both trace and count data).
- A C parser that can generate instrumented application source code.

- A performance data trace capture library for single processor Unix systems and for the Intel iPSC/2 and iPSC/860 hypercubes.
- A flexible self-documenting data meta-format and associated tools that can be used to describe and process diverse types of data.
- A graphical performance analysis environment. This environment is based on the graphical configuration of directed data analysis graphs, that can be used to analyze and display dynamic performance data.
- A set of graphical display widgets for the X window system. These include bargraphs, dials, scatterplots, kiviat diagrams, and contour plots.

In addition to dynamic X window graphics for the display of performance data, PABLO supports the use of sonic data presentation via replay of sampled sounds, via a Sun SparcStation audio port, and via use of the Musical Instrument Digital Interface (MIDI).

2.3.1.1 Instrumentation Software

The three components of the PABLO instrumentation software are a graphical interface for interactively specifying source code instrumentation points, C and Fortran parsers that generate source code with embedded calls to a trace capture library, and a trace capture library that records the performance data. This approach is a compromise to maximize portability.

The parsers accept source code to be instrumented and produce the parse tree information needed by the graphical instrumentation interface. The graphical interface, based on X and Motif, will then interpret the parse tree data information and allow the user to graphically specify source code instrumentation points.

Because PABLO's only modification to source code is the insertion of calls to the trace capture library, it is possible to move an instrumented program to another parallel system. If the trace capture library has been ported to the new parallel system, the same application data can be captured there, permitting cross-architecture performance comparisons. The performance data trace capture library is currently available for single processor Unix systems and for the Intel iPSC/2 and iPSC/860 hypercubes, Intel Paragon, IBM RS/6000, SGI, HP 700 series, IBM RS/6000, DECstation, and Thinking Machines CM-5.

In order to balance the needs of detailed data and minimal perturbation, the PABLO instrumentation software supports three classes of instrumentation events: trace, count and time interval. The instances of each event class are recorded in a trace file using a self-documenting data format (SDDF) that includes internal definitions of data types, sizes, and names. For events of all three classes, the PABLO trace capture library supports optional, user-written extension functions that can process event data before it is written to the trace file, acting as a filter mechanism.

Another important issue that is intended to minimize perturbation during data recording is PABLO's implementation of an adaptive instrumentation control, which allows a user-specified maximum trace level with each event. In addition to instrumentation control via user-specified trace levels, the PABLO instrumentation software will dynamically adjust event trace levels within the user-specified range. The PABLO trace capture library also monitors the aggregate event rate using a similar algorithm.

In this way, the PABLO instrumentation software allows a balance between event data volume against application perturbation, maximizing the amount of useful trace data.

2.3.1.2 The Self-Describing Trace Data Format (SDDF)

The PABLO portable trace data format links the PABLO instrumentation software, which captures dynamic performance data, and the PABLO data analysis environment, which provides the tools to reduce and analyze the data.

The Self-Describing Data Format (SDDF) is a trace description language that specifies both the structure of data records and data record instances. SDDF is best viewed as a data meta-format. When creating a trace file, one is free to decide what is appropriate for that situation and describe event records accordingly. In the same way, when analyzing the data one uses the embedded descriptions to interpret the data.

SDDF allows ASCII and binary representations in order to provide compactness and portability when necessary. Simple tools can quickly convert from one representation to the other.

The ASCII and binary versions of the SDDF meta-format describe four classes of data records:

- Stream attribute: information about the entire trace file.
- Record descriptor: template for a data record.
- Record data: record instance.
- Command: action to be taken.

2.3.1.3 Data Analysis Environment

The PABLO Data Analysis Environment provides a framework for reducing, analyzing, and presenting performance data. The keys to the PABLO data analysis environment's extensibility and portability are:

- The reliance on a toolkit of data transformation modules capable of processing the self-describing data format.
- The recognition that software embedded knowledge of parallel system architectures makes portability impossible.

This environment supports the graphical interconnection of performance data transformation modules to form a directed acyclic data analysis graph. Performance data then flows through the graph nodes and is transformed to yield the desired performance metrics. Details like file input/output, storage allocation, and module execution scheduling are isolated in the environment infrastructure. This allows extension of the environment by adding new data transformation modules (see figure 2.1).

By graphically connecting these modules to form a directed acyclic graph and interactively selecting which trace data records should be processed by each data analysis module, the user selects the desired data transformations and presentations. All data passed among modules in the configured data analysis graph is encapsulated in self-describing data format records. When the analysis environment begins execution, the environment infrastructure automatically extracts the necessary fields from data

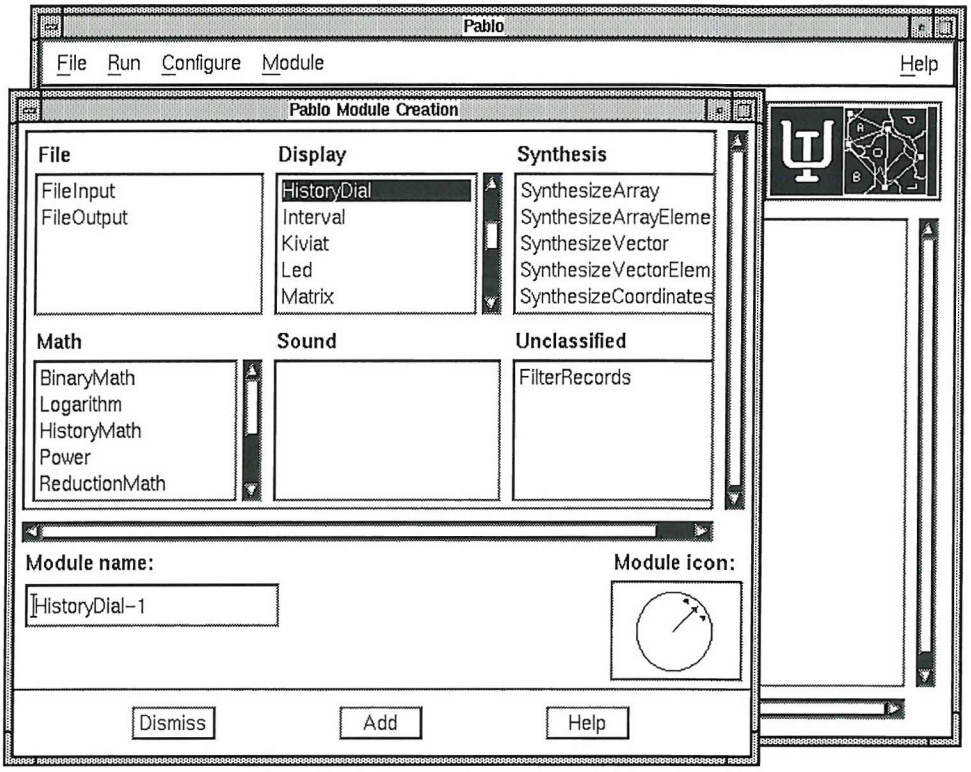

Figure 2.1: PABLO Module Creation.

records and passes those values to the modules for processing. The semantics of the data analysis are embedded in the user's graphical configuration, not the data analysis software.

The graphical programming model allows users to develop new data analyses by constructing data analysis graphs whose semantics reflect the desired transformation.

The PABLO data analysis environment's flexibility rests not only on the self-describing data format (SDDF) and the environment infrastructure, but also on the *palette* of data analysis and presentation modules. The basic set of data analysis modules process performance data without software embedded knowledge of higher level data semantics. All data analysis modules in the basic set are *polymorphic* - they process all mathematically valid combinations of data inputs. The basic module set includes simple mathematical transforms such as counts, sums, products, differences, ratios, maxima, minima, averages, absolute value, powers, logarithms, and trigonometric functions. This data analysis environment will serve three kind of users: novice, intermediate, and expert.

The PABLO environment supports two types of data presentation modules: graphical and sonic. The graphical displays are based on the X window system. Currently supported graphical displays include:

- Bar graphs.
- Bubble charts.
- Strip charts.
- Contour plots.
- Dials (with and without history).
- Interval plots.
- Kiviat diagrams.
- LEDs (discrete bar graphs).
- X-Y line/scatter plots.
- Matrix displays.
- Pie charts.
- Polar plots.
- 3-dimensional scatter plot.

2.3.1.4 Data Sonification

Just as visual elements (e.g. colour, form, and line) are combined to present and analyze data visually, the elements of sound (e.g. duration, pitch, volume, timbre, and spatial location) can be combined to present data aurally. Sound can emphasize data characteristics not easily seen, much in the same way a movie sound track conveys information complementary to the imagery.

Much of sonification's embryonic state is attributable to the lack of software standards and common hardware interfaces. A reasonable compromise is instruments that conform to the Musical Instrument Digital Interface (MIDI). The PABLO sound toolkit is an attempt to ensure portability across multiple sound devices by separating the sound hardware interface from the sonification (i.e. the algorithm that creates sound from data). Each sound device is managed by its own network server, which accepts sonic messages that cause sounds to be played. An application program opens a connection to a (possible remote) sound server, receives a list of supported sonic messages (like play note, vary pitch or select synthesized instrument), and uses parameterized variants of those messages thereafter to interact with the sound server.

The current implementation of PABLO supports both the sampled sound capability of the Sun SparcStation audio port and a variety of MIDI systems. Creating sonifications is simplified by four major abstractions: sound control files, transformation functions, sonic widgets and widget control files. Sonifications can be saved, edited, and recreated. Several examples of mappings of performance data from distributed memory parallel systems to sound have been developed. These examples include sonifying message transmissions by:

- Mapping the node number of a message transmitter to pitch.
  On hearing this example one can detect not only the frequency of transmissions but also the number of communicating processors and, with knowledge of the mapping function, their physical separation in a multicomputer network.

- Mapping the node number of a message transmitter to pitch in the left speaker and, when the message is received, mapping the node number of the message receiver to pitch in the right speaker.
  This reveals details about temporal communication activity that could not be easily deduced from concurrent visualization, namely that messages often occur in tightly clustered groups; staccato bursts of sound makes this immediately obvious.

- Mapping the node numbers of message transmitter to pitch, but a note begins when a message is sent and continues until the message is received.
  This shows not only the expected delay between message transmission and receipt but also the distribution of latencies.

A set of *earcons* or audio warnings have been also developed for PABLO. This presently includes sampled voice warnings, enumerations, and alarms. A good example is when an aggregate processor utilization was mapped to a sonic alarm that sounds only when utilization fell below a specified threshold. This is particularly useful when processing large volumes of data since users are unable to maintain attention focus on graphic displays for long periods of time. A sonic alarm can quickly alert about important behaviours.

2.3.1.5 A case study using PABLO

In order to gain some experience using PABLO, the LU Matrix Decomposition Algorithm of Section 4.4.3 was analyzed[1]. For this purpose, a directed data analysis

[1]Results for 4 processors on a T800 Transputer Parsys Supernode. The size of the matrix was n=100.

graph was built. The basic idea of this experiment was to determine the bottlenecks of the algorithm using the current displays provided by PABLO and to evaluate the tool itself.

The design and creation of the data analysis graph was easy. The most important consideration is to have specific questions to ask about the performance data. For this example, the factors to be analyzed are work load and communication traffic. Work load is represented by the percentage of idle time per processor using two displays: Bargraph and Kiviat. Communication traffic is represented by the number of messages that each node has produced: the length of those messages and the communication patterns. Five displays are used here: Led, Matrix, Contour, Bargraph and Kiviat.

1. The Led display represents its input parameter as a set of levels stacked either horizontally or vertically. Individual levels are filled according to the magnitude of the datum and the "minimum" and "maximum" data values stored within the functional unit. The minimum and maximum values may be set by the user during the configuration process.

2. Matrix accepts a two-dimensional array of input values and represents the data as individual squares in a grid pattern accessible by row/column indexing. The squares within a matrix are filled with a solid colour: the colour varies with the magnitude of the data according to the colourmap defined in the functional unit.

3. Contour represents the data as contour lines drawn within a grid in the plotting array. Contours are drawn according to the magnitude of the data: larger values correspond to contours having more lines. The grid line defining the region in which each contour appears may or may not be drawn. The intersection points of the grid lines define identify particular (row,column) locations; location (0,0) is the top-left intersection, and rows grow to the right and columns grow downward.

4. Bargraph represents its input parameter as a single bar whose size varies with the magnitude of the incoming datum. In addition to the bar, Bargraph maintains a "sticky hold" value, a line drawn in colour to mark the maximum value the bar has obtained during a given period.

5. The Kiviat display accepts as input a vector of data values and maps each vector element to a location on a corresponding axis within valid "minimum" and "maximum" ranges according to the values set by the user at configuration time. Kiviat axes are spoke like and radiate from the origin of a circle. A default option for Kiviat is to connect the plotted points with a line and fill (or shade) the resulting closed area with a colour.

All these displays can be seen in figure 2.2. The displays show the first stage of the algorithm in which every processor is working on its particular section of the matrix. It is easy to see in this figure how well the processors are working at this moment. The percentage of idle time is not high (around 10%), and it is equally balanced among the processors (Bargraph and Kiviat). It is possible to obtain this value by *clicking* the mouse on the desired position of the display.

The ring communication pattern can be recognized in the Contour and Matrix displays. The same number of messages have been sent by each processor. These messages have the same length (Led display). The second stage of the algorithm can

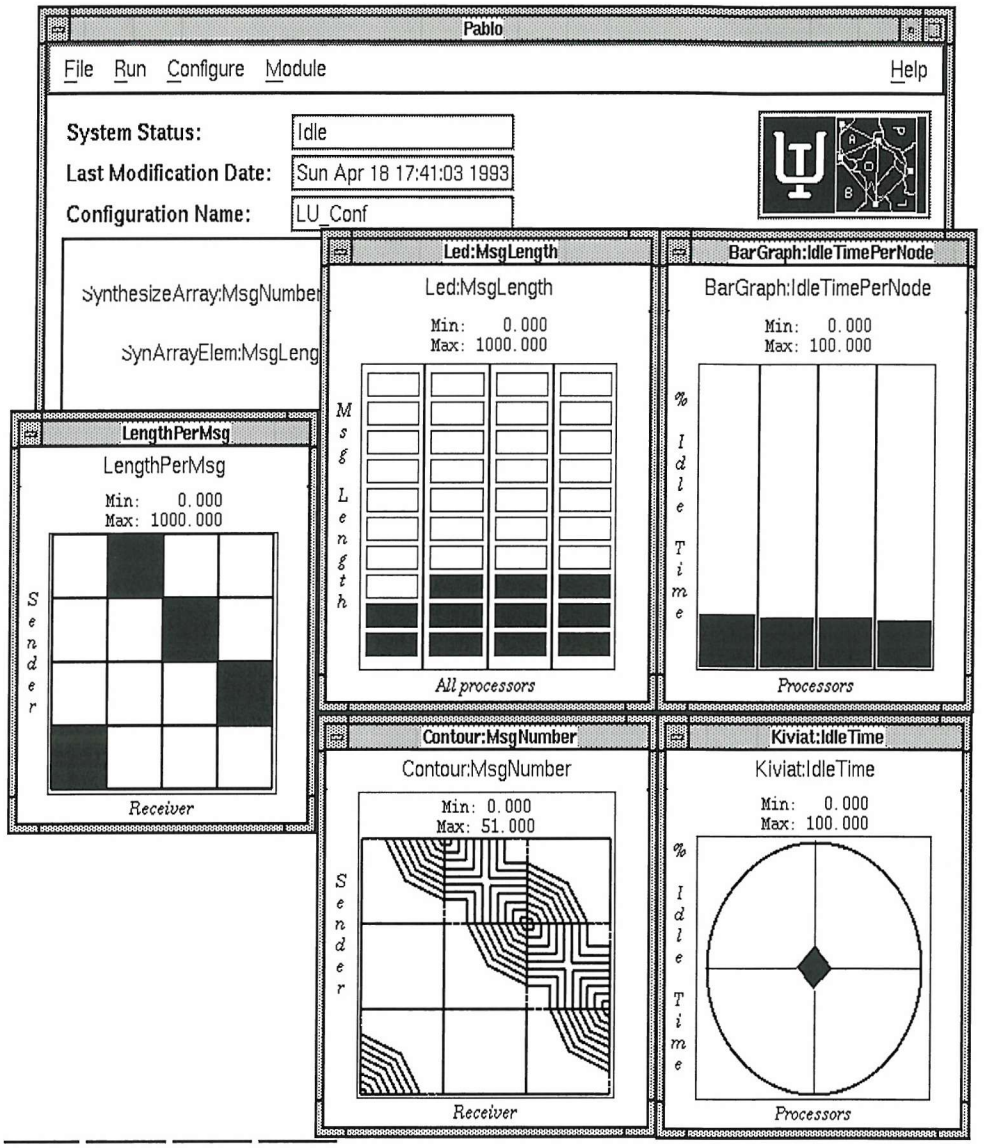

Figure 2.2: PABLO displays for the LU Matrix Decomposition Algorithm - Stage I.

be seen in figure 2.3. At this moment, the amount of work to be done in parallel is not enough to keep all the processors busy and the dominant factor is communications. It is also possible to recognize that the algorithm is finishing and that the communication pattern is changing (because of the gathering of partial results).

2.3.1.6 Comments and Conclusions

PABLO is a very powerful and useful tool for monitoring the performance and behaviour of parallel applications. It was helpful in understanding some features described in the previous section about the LU Matrix Decomposition Algorithm. Unfortunately, the PABLO sound facility could not be used due to problems with the installation of the software. However, sound is a new way to represent performance data that could help to complement visual displays. More research is needed to understand how to incorporate and use sound.

There were some problems with the present release of PABLO (e.g. execution speed when the graph is complex, possibility of going backward or forward, in time, as well as some bugs), but these problems will certainly be fixed in future releases. PABLO has many advantages. It is portable, scalable and extensible and these issues are important. In the near future it is likely that many contributions will be made in order to increase the current PABLO displays and modules in general.

2.3.2 SIMPLE

SIMPLE is a modular tool environment for performance evaluation, modeling and visualization of monitored event traces [23, 40, 41]. The acronym SIMPLE stands for Source related and Integrated Multiprocessor and computer Performance evaluation, modeLing and visualization Environment. SIMPLE is the first performance evaluation tool environment which is independent of the monitor device(s) used and the system monitored. The crucial step toward this independence was introducing the data access interface TDL/POET which can decode measured data of arbitrary structure, format and representation.

2.3.2.1 Behavioural abstraction

SIMPLE's approach to view parallel and distributed systems is called behavioural abstraction [40]. It is based upon viewing a system's activity as consisting of a stream of points of interest, the so-called events, representing significant points of the system's behaviour. An event together with its attributes completely describes what occurred, when and where in the system. A stream of events sorted by increasing acquisition time is called an event trace.

2.3.2.2 POET - Problem Oriented Event Trace Interface

POET provides a simple and monitor independent function interface which allows the user to access measured data stored in event traces in a problem-oriented manner. In order to be able to access and decode the different measured data, the POET functions use a so-called access *key file*, which contains a complete description of formats and properties of the measured data.

The measured data are collected using event driven monitoring. This means that whenever the monitor device recognizes an event, it stores a data record describing the

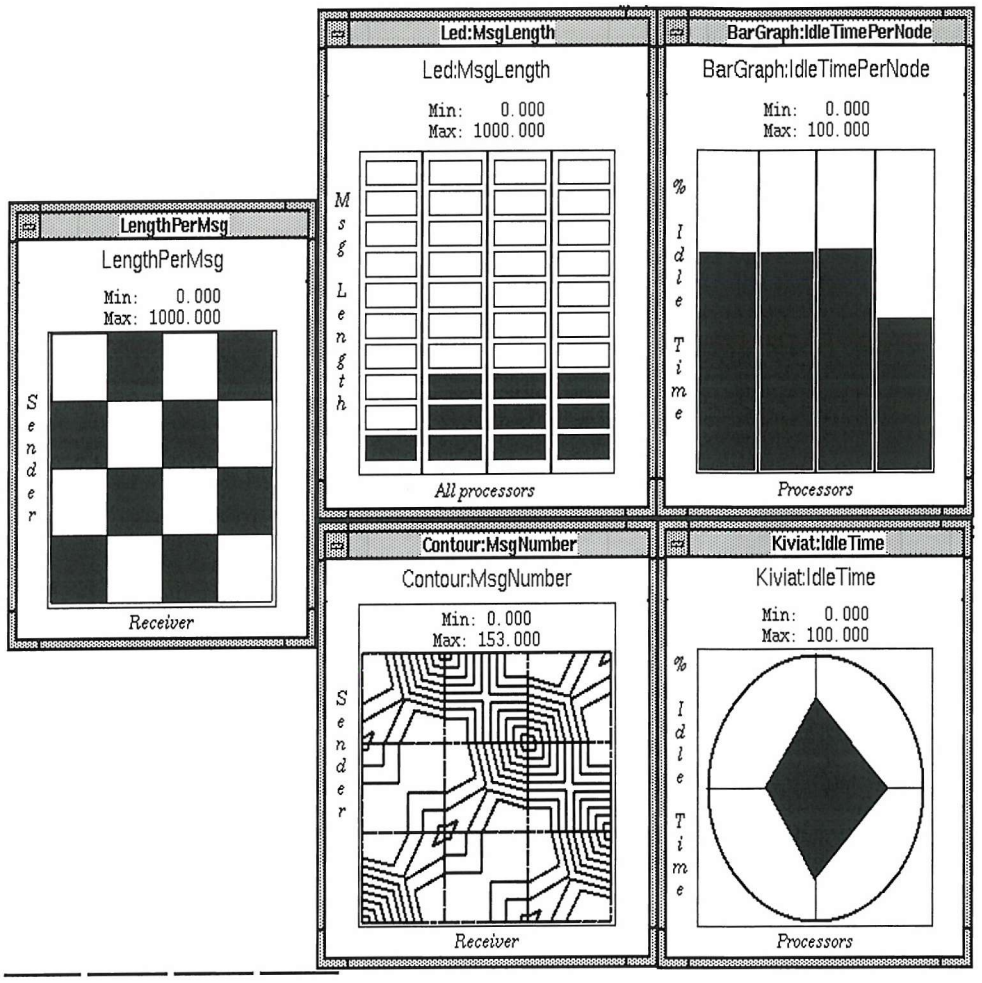

Figure 2.3: PABLO displays for the LU Matrix Decomposition Algorithm - Stage II.

occurred event. Such a record is called an E-record and contains information about what happened, when and where. An E-record consists of several record fields, each containing a single value describing one aspect of the event. A sequence of E-records is called an event trace (file). This event trace file can be segmented, if some E-records have been lost, or unsegmented, which is a trace segment describing a completely observed time interval of the dynamic behaviour of the monitored system.

Using the POET library, it is possible to write programs that can easily access performance data to retrieve any desired kind of information. The example in figure 2.4 shows the number of occurrences of each event.

```
  trace_start:  4
         open:  0
         load:  0
         send:  0
        probe:  0
         recv: ############ 399
recv_blocking: ############ 405
  recv_waking:  2
      message:  0
         sync:  0
    compstats: ################################################## 1612
    commstats: ################################################## 1604
        close:  5
  trace_level:  0
   trace_mark:  0
trace_message:  0
   trace_stop:  0
  trace_flush:  0
   trace_exit:  5
  block_begin:  0
    block_end:  0
  trace_marks:  6
   clock_sync: ############ 401
```

Figure 2.4: Example generated using the POET library.

2.3.2.3 TDL - Trace Description Language

The TDL language was developed in order to make the construction of the *access key* more user-friendly and is especially well suited for a problem-oriented description of event traces. The access key is generated by a TDL compiler called TDLC, which checks the user written TDL description (an example of a TDL description can be seen in figure 2.5).

The development of TDL had two main goals:

- To make a language available which clearly reflects the fundamental structure of an event trace.
- To allow users to be able to read and understand a given TDL description, even if they are not familiar with all language details.

```
TRACE DESCRIPTION:

  TRACE IS UNSEGMENTED;

EVENT RECORD:

  TOKEN:
    NAME IS PROCESSOR;
    LENGTH IS 1 BYTE;
    VALUES ARE [1..3];
    INTERPRETATION
      1 = 'PROC 1',
      2 = 'PROC 2',
      3 = 'PROC 3';

  TOKEN:
    NAME IS EVENT;
    LENGTH IS 1 BYTE;
    VALUES ARE ['B', 'C', 'S', 'E'];
    INTERPRETATION
      'B' = 'BEGIN',
      'C' = 'COMPUTE',
      'S' = 'SYNC',
      'E' = 'END';

  TIME:
    NAME IS ADQUISITION;
    FORMAT IS (UNSIGNED*4, ms);
    MODE IS POINT;
```

Figure 2.5: Example of TDL description

Using TDL/POET for all the SIMPLE tools allows independence from the object system, especially of its operating system and the programming languages used. In order to adapt SIMPLE to another kind of measurement, one only has to write a TDL description of the event trace to be analyzed. As TDL/POET provides a uniform interface, the evaluation of the data is independent of its recording.

2.3.2.4 SIMPLE Tools

- Checktrace
  The program *checktrace* performs some simple tests on the event trace given that can be applied to all event traces (e.g. it is checked whether the E-records are correctly sorted according to increasing timestamp).

- Varus
  Varus performs user-defined checks called validation rules or assertions on an event trace. Varus opens an input file and checks the specified assertions on syntactic and semantic correctness. If no errors can be found, the second phase is started, during which the assertions are checked against the event trace.

```
ASSERT
  NUMBER ( EVENT == 'RelaA' ) == NUMBER ( EVENT == 'RelaE' );

IF PROCESSOR == 'Slave1' ASSERT
  DISTANCE  ( EVENT == 'SyncA' ) TO ( EVENT == 'SyncE' ) <= 5 [ ms ];
```

Figure 2.6: Example VARUS file

- List
  List lists a (binary) event trace file in a readable form. For that purpose it needs two parameters: the name of the event trace file and the name of the corresponding key file (figure 2.7).

- Trcstat
  It performs simple statistical computations on an event trace. It can count the frequency of token field values, compute the distance between the different occurrences of an event or the duration of activities defined by a start and end event. The results can be seen by tables (figure 2.8) or by graphics (figure 2.9) using the UNIX facility *xgraph*.

- S data analysis package
  The tools *List* and *Trcstat* are quite useful, but normally more complex computations have to be done. The user wants to analyze the measured data interactively, with graphics support and in a high-level environment. For this purpose, the commercial data analysis package S from AT&T was integrated in SIMPLE. S provides a high level programming language for data manipulation and graphics. S was extended in order to access event traces via the TDL/POET interface.

- Gantt
  To visualize the dynamic behaviour of concurrent activities, the function *gantt*

```
******* NEW GLOBAL SEGMENT ************************************

[ record 0 ]

  PROCESSOR            :  Master
  EVENT                :  MGZ-E
  ACQUISITION          :  0  [ ns ]

[ record 1 ]

  PROCESSOR            :  Slave1
  EVENT                :  MGZ-E
  ACQUISITION          :  0  [ ns ]

[ record 2 ]

  PROCESSOR            :  Slave2
  EVENT                :  RelaE
  ACQUISITION          :  0  [ ns ]
```

Figure 2.7: Trace protocol generated by LIST

```
#01I  FREQUENCY EVENT PROCESSOR
#02I  DURATION 'SyncA' 'SyncE' PROCESSOR UPPER=5[ms]

#02S              639 [ us ] on 'Master'
#02S              591 [ us ] on 'Slave1'
#02S              519 [ us ] on 'Slave2'

...

#02T          Master      Slave1      Slave2      Slave3
#02T    no:   0)49(0      0)46(3      0)46(3      0)31(18     0)172(24
#02T   min:     412         477         474         487         412  [ us ]
#02T   max:   2.502       3.412       3.372       4.513       4.513  [ us ]
#02T   sum:  46.974      30.702      32.881      61.279     171.836  [ us ]
#02T  mean:     959         667         715       1.977         999  [ us ]
#02T   25%:     531         501         505       1.197         501  [ us ]
#02T   med:     904         534         513       1.433         562  [ us ]
#02T   75%:   1.289         639         646       2.726       1.265  [ us ]
#02T   var: 293.524     241.020     248.028   1.724.769     738.753  [ us ]
```

Figure 2.8: Output produced by TRCSTAT

was implemented in S. Gantt can be used to draw Gantt charts of arbitrary event traces. Gantt charts are usually known as diagrams that represent the temporal evolution of different program states of an examined object system versus a common time axis (see figure 2.10).

- Smart - Slow Motion Animated Review of Traces
  Smart visualizes an event trace file. This is useful for investigating the dynamic

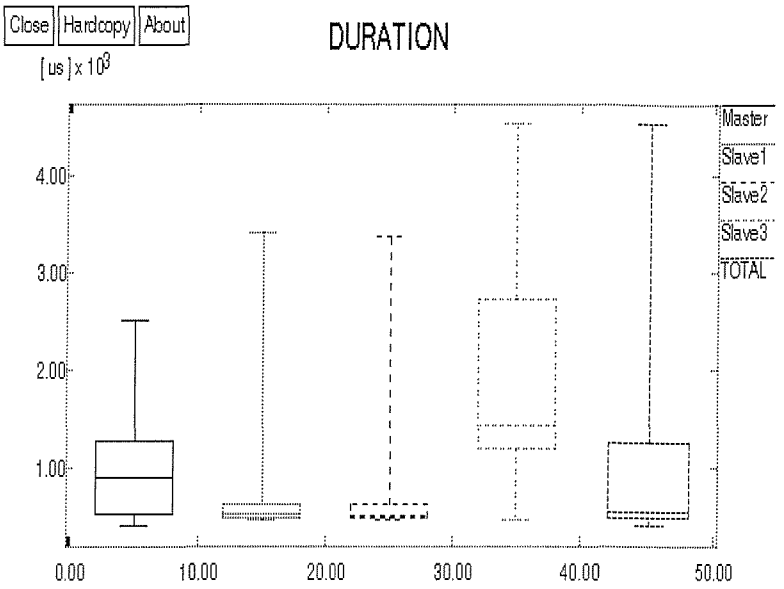

Figure 2.9: TRCSTAT graphic generated by XGRAPH

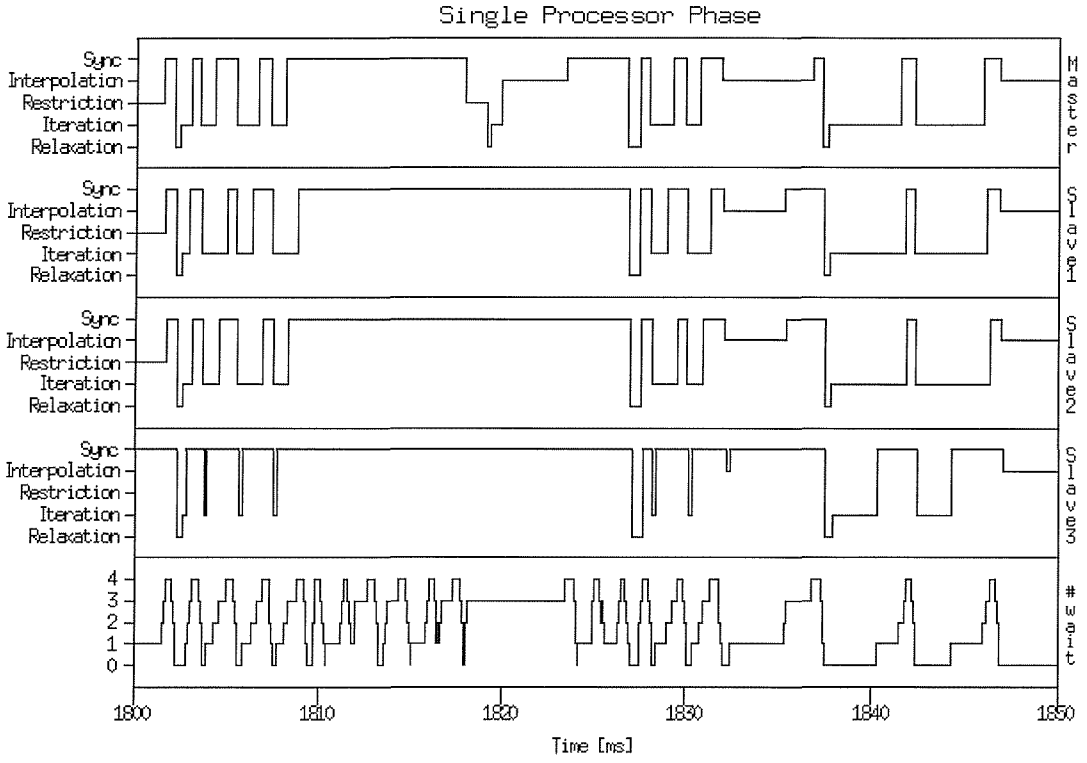

Figure 2.10: Gantt diagram generated with S

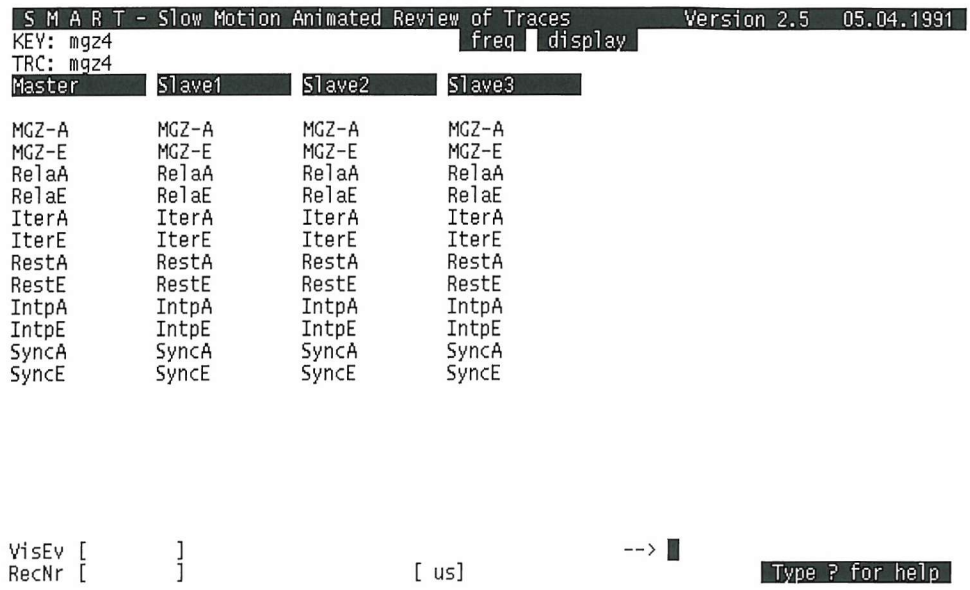

Figure 2.11: Screen snapshot of SMART

behavior of an algorithm or of a computer system. The visualization presents the algorithm in action, exposing properties of the program that might otherwise be difficult to understand or might even remain unnoticed. The monitored events will be shown in the correct temporal order, but it is possible to distinguish between two modes: first the interactive mode, in which the user can force the step to the next event record, and second the timely mode, in which the event trace is visualized as a movie in slow motion. Smart can be used on any ASCII-terminal (see figure 2.11).

- Fact - Find activities in event traces
  Fact can be used to find activities in an event trace. An activity is a sequence of events. The user can define activities by specifying this sequence as a regular expression of events. In the activities definition file used by fact the user can specify the activities to be searched for. It has to be created manually with a text editor.

- Other Tools
  There are other tools like AICOS (Automatic Instrumentation of C Object Software), VISIMON (visualization tool), CRDES (descriptor file editor), FILTER and some others which are not described in this report.

2.3.2.5 A case study using SIMPLE

An example trace file generated by PICL from a *fft* benchmark (also called *butterfly*), was chosen to be analyzed using SIMPLE. In figure 2.12 (see below) it is possible to compare the Feynman diagram of ParaGraph [4] and the Gantt diagram generated by one of the SIMPLE tools. Looking at the Gantt figure we cannot see the *butterfly* pattern, but another different pattern. Notice that at each time, all send and receive events occur almost at the same time. Only 4 processors are shown in the Gantt figure because it is not very scalable (i.e. the picture is not clear when there are either too

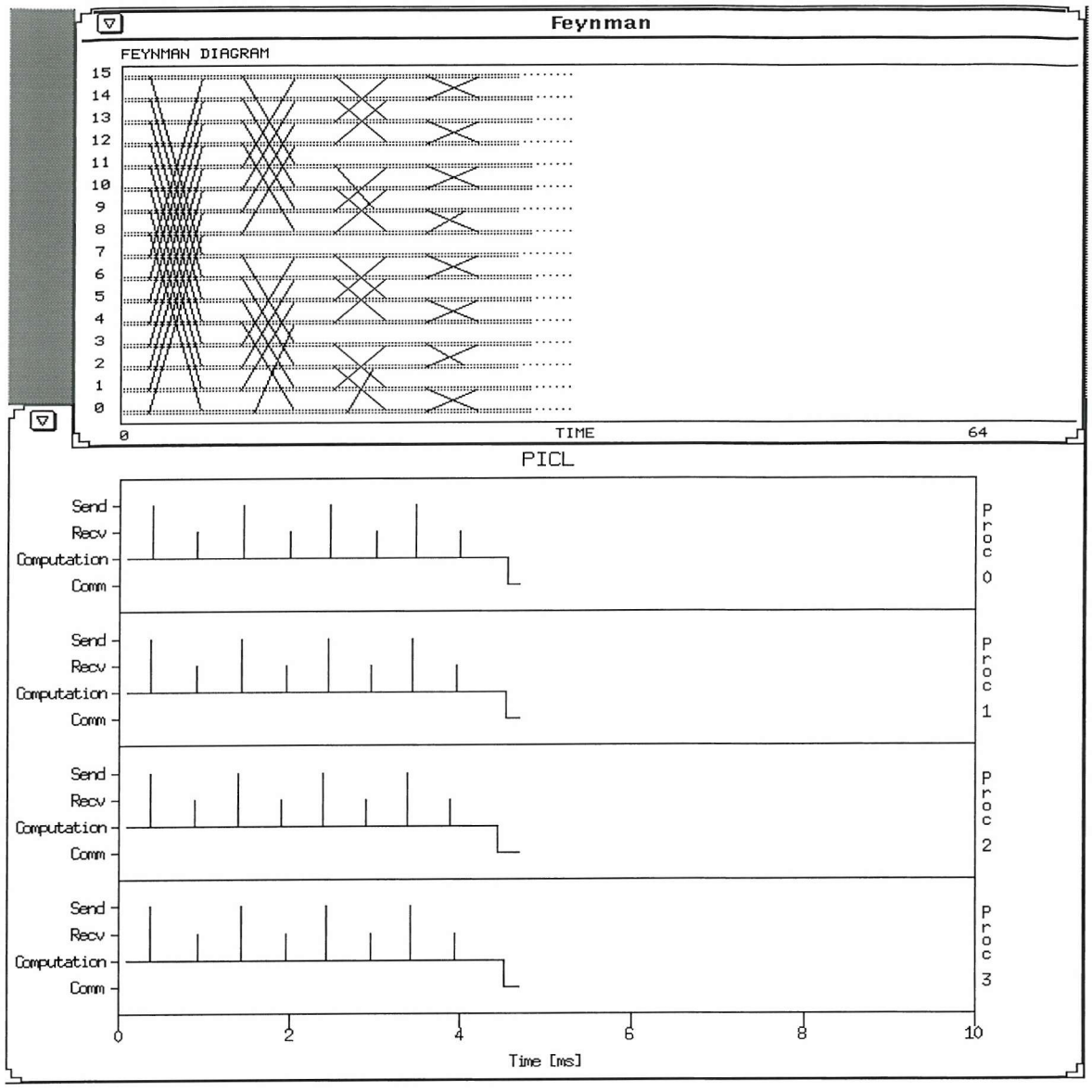

Figure 2.12: Feynman diagram vs. Gantt

many events or a large number of processors). The Gantt figure cannot show the relationship between send and receive. It is possible to see that a particular processor is sending a message, but the receiver is unknown.

If one wants to measure the duration between two events, e.g. send and receive on the same processor, the procedure is as follows:

- Create a descriptor file specifying the desired measure (i.e. *DURATION 'send' 'recv' PROCESSOR*).
- Use the *trcstat* tool to read the trace file and generate the corresponding results (*trcstatout* file).
- Finally, the result can be seen graphically (using *xgraph*, figure 2.15), or filtering the information of the *trcstatout* file (figure 2.13).

```
%grep '#02S' picl.trcstatout | grep 'Proc 0'
#02S              536 [ us ] on 'Proc 0'
#02S              529 [ us ] on 'Proc 0'
#02S              526 [ us ] on 'Proc 0'
#02S              522 [ us ] on 'Proc 0'
```

Figure 2.13: Filtered information - Processor 0

It is also possible to write a small program in order to make the query *how long is the interval of time between events send and recv?*, using the POET-library and get the following result (figure 2.14):

```
% duration picl.key picl.trc send recv 'Proc 0'
activity  1:              536 [1 us]
activity  2:              529 [1 us]
activity  3:              526 [1 us]
activity  4:              522 [1 us]
=========================================
        min:              522 [1 us]
        max:              536 [1 us]
       mean:              528 [1 us]
```

Figure 2.14: Filtered information using POET - Processor 0

As this example shows, it is not difficult to customize "queries" and get the corresponding information from the trace file.

It is also possible to define *activities*, as an interpretation of a specific sequence of events. For example, a sequence like *send*, followed by anything until a *recv* is found, it can be defined as an activity (see figure 2.16). To make a query using activities, the tool *fact* (find activities) is provided by SIMPLE. The result of this query for a particular processor (0 to be consistent), is showed in figure 2.17.

Using SMART it is possible to see the behaviour of the program, changing from event to event in slow motion. However, it is difficult to really *visualize* what is happening.

2.3.2.6 Comments and Conclusions

- In [5], Mohr affirms that standardization of the physical event trace format is not the right approach. No standard format can be flexible enough to represent all possible event trace formats unless format information is included in the trace. Moreover, there is a great variety of existing (hardware) monitors which cannot produce a standardized format and many conversion programs would have to be implemented. In this sense, the TDL/POET interface shows that a generalized access method for arbitrary event traces works well without requiring standardized physical formats. Tourancheau et al [42] affirm that no assumption about the trace record format should be made and a self-defining (PABLO, [22]) or external file description format (SIMPLE, [23]) should be used.

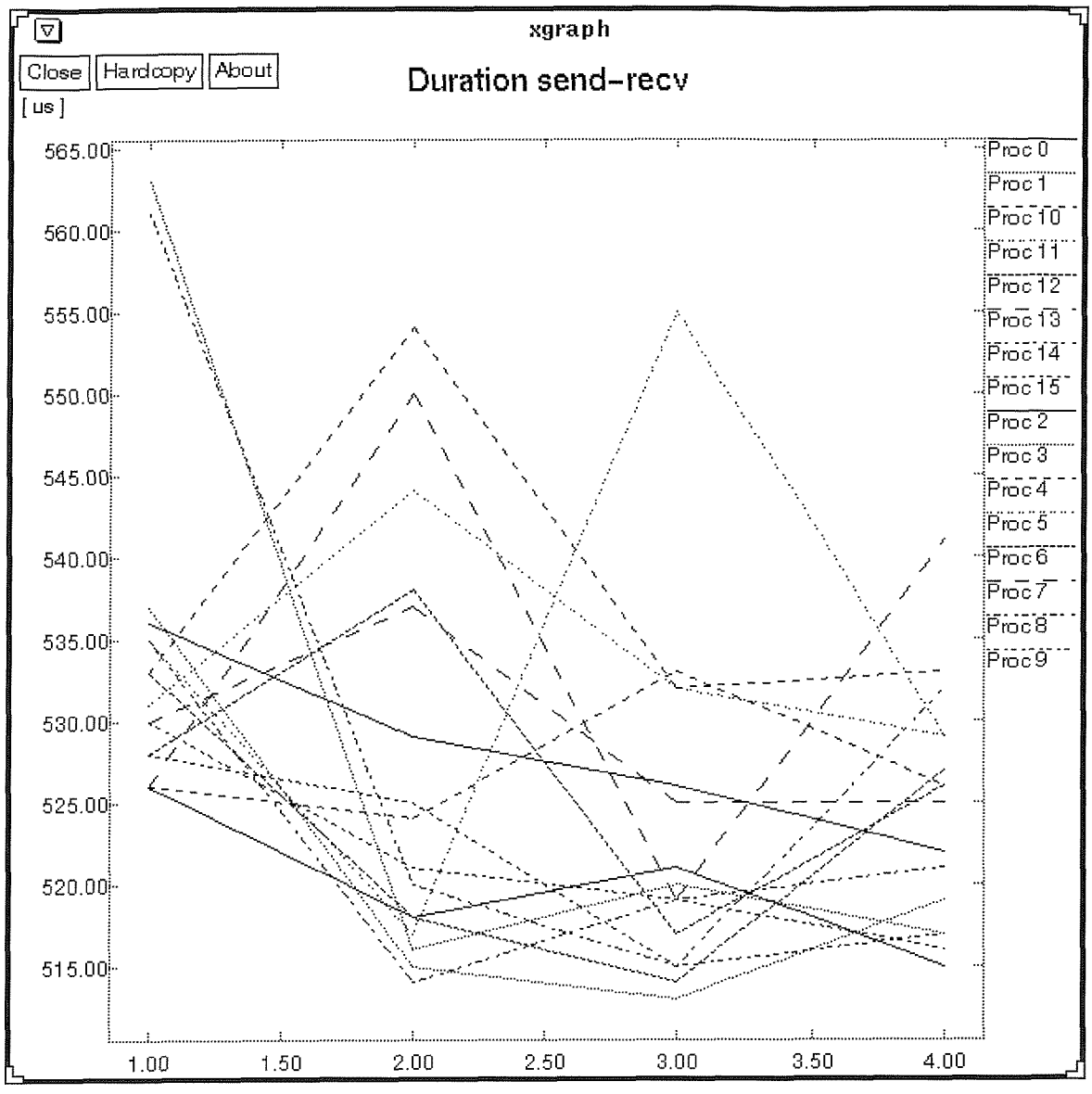

Figure 2.15: Duration send-recv

```
ACTIVITY step IS

  (send ->> recv)

END
```

Figure 2.16: Activity's definition file

- The Gantt display cannot show the relationship between sender and receiver as the Feynman display in ParaGraph. The Gantt display is also not scalable if the number of events shown is large.
- Using the POET library, it is possible to create any kind of query just by writing a small program. Using tools like *fact* it is also possible to define *activities* that relate a sequence of different events. These facilities have been shown to be very useful.
- The SIMPLE visualization facilities are not very complex and they are very easy to use. However, it is difficult to really *visualize* patterns of communications between processors like for example in ParaGraph. In SMART it is possible to see the transition between events but there is a lack of general perspective and is not possible to see what happened (i.e. to see all events) during a particular interval of time. To do this, other tools like *gantt* can be used.
- SIMPLE is a good example of what we may have in the future: a *collection* of different performance analysis tools in order to analyze a particular trace file, which have been generated using either hardware, software or hybrid mechanisms but which is also independent of the object system.

2.3.3 ParaGraph

ParaGraph is a graphical display system for visualizing the behaviour of parallel programs on message passing multiprocessor machines. It is a post-execution tool that was written for use with PICL (Portable Instrumented Communication Library). Both ParaGraph and PICL were developed at Oak Ridge National Laboratory. PICL optionally produces an execution trace file during an actual run of a parallel program on a message-passing machine, and the resulting trace data can then be replayed graphically with ParaGraph. ParaGraph provides several distinct visual perspectives from which to view processor utilization, communication traffic and other performance data. The multiplicity of ways offered by ParaGraph to view the information is an advantage it has over many other tools.

ParaGraph and PICL are public domain software. ParaGraph and PICL have been modified at the University of Southampton in order to extend and enhance some of their features, allowing ParaGraph to read trace files generated by software other than PICL (PARMACS [43] and Express [11]), and implementing a version of PICL to run on transputers.

For more details about the original versions of ParaGraph and PICL, see [9, 10, 44]. Details about the modified versions of ParaGraph and PICL can be found in [45, 4].

```
step[  1]:                    536 [ us ]
   66: send
   67: compstats
   91: compstats
  116: recv

step[  2]:                    529 [ us ]
  174: send
  175: compstats
  190: compstats
  220: recv

step[  3]:                    526 [ us ]
  270: send
  271: compstats
  286: compstats
  316: recv

step[  4]:                    522 [ us ]
  366: send
  367: compstats
  385: compstats
  412: recv


step   no:                      4
step  min:                    522 [ us ]
step  max:                    536 [ us ]
step  sum:                  2.113 [ us ]
step mean:                    528 [ us ]
step  med:                    528 [ us ]
step  var:                     35 [ (us)^2 ]
```

Figure 2.17: Results using the *fact* tool

This section will be based on the modified version of ParaGraph.

2.3.3.1 General description

The principal design objectives of the original version of ParaGraph were ease of understanding, ease of use and portability. Ease of use implies that the displays should be as self-evident and consistent as possible, providing a variety of them and offering different perspectives. ParaGraph is menu driven with most user input provided by mouse operation, although some options require a small amount of keyboard input. There are two aspects to portability. Firstly, the graphical system itself should be portable. ParaGraph is based on the X Window System and runs on a wide variety of graphical workstations. It uses no X toolkit and requires only Xlib.

Although ParaGraph is most effective in colour, it also works on monochrome and grayscale monitors. The second aspect of portability is that the package should be capable of displaying the execution behaviour of programs running on different types of parallel machines. This requirement has been partially satisfied by using PICL, and this has been extended with the development of trace filter programs for PARMACS and Express.

2.3.3.2 General displays

ParaGraph has a graphical, menu-oriented user interface that accepts user input via mouse clicks and keystrokes. Menu selections determine the execution behaviour of ParaGraph both statically (e.g. initial selection of parameter values) and dynamically (e.g. pause/resume, single-step mode). ParaGraph preprocesses the input trace file to determine relevant parameters (e.g. time scale, number of processors) automatically before the graphical simulation begins. These values, however, may be overridden by the user if desired.

ParaGraph offers the following display options[2]:

- *Feynman*
  This is one of the most useful displays for getting an overall impression of the state of execution of the program. The thread of execution of each processor is represented by a horizontal line, which changes colour to indicate whether the processor is active, idle, or waiting for a send or receive to complete. The message transfers between any pair of processors are represented by diagonal lines from source to destination. Those points where the communication lines meet the processor lines mark the send and receive times. The Feynman display is particularly useful showing whether processors are idle because they are blocked waiting for a message to be sent or received.

- *Animation*
  It is useful to see how communications take place between processors. Nodes are depicted as circles, communications between nodes are represented by lines and the status of each node is displayed by a colour which may be either busy, idle, sending or receiving. When there is a send from one processor to another, a line is drawn and then deleted when it is received. At the end of the simulation all

[2]The actual maximum number of processors supported by this version is 32. Most of the displays of the current version of ParaGraph described in [10], allow up to 512 processors although a few are limited to 128 and one is limited to 16.

the lines of communication that were used are displayed, indicating the logical communication links between processors.

- *Message Lengths*
  Messages are represented by a coloured square in a grid whose rows and columns correspond to sending and receiving processors. The length of each message is indicated by the colour of the square. This colour, may be modified by the user via keyboard input in the appropriate window of the legend. At the end of the simulation, the cumulative message volume for the complete run is shown.

- *Kiviat*
  This is a geometric representation of individual processor utilization and overall load balance. Each processor is depicted as a spoke of a wheel. The fractional utilization of each processor portrays a point on the spoke, with the hub of the wheel representing zero (completely idle) and the edge of the spoke representing one (completely busy). The vertices of the polygon whose size and shape indicate the load balancing and utilization of the system, are determined by taking together all those points on the spoke.

- *Gantt*
  This display shows when each processor is busy, idle or waiting to receive a message, using different colours for each state. Each processor is depicted as a line which changes its colour depending on the state. It is similar to the Feynman display, but does not show the communication lines between processors.

- *Utilization*
  This plot displays the number of processors that are busy as a function of time. It is useful to determine the load balance of the system. The same information can be found in the Gantt and Feynman displays, but it is usefully summarized here.

- *Communication Load*
  It shows either the volume or the number of messages of the system as a function of time. It is like the Utilization display but for communications.

- *Queue Size*
  This display depicts the length in bytes of the message queue for a particular node as a function of time, as well as the number of messages in a particular queue.

- *Message Queues*
  Depicts the length of the messages that are currently traveling on the system for each processor. The display uses different colours to show different lengths and at the end of the simulation it produces a useful summary. It works also with the number of messages as well as with the length.

- *Communication Statistics*
  This display shows the communication statistics for a single node as a function of time. It displays not only the source but also the destination. It has three types: node, message type and message length. In the first case, it shows the messages going from and to a particular node. The last two cases show the same situation but for message type and message length.

- *Clock*
  This indicates current time in the simulation.
- *Trace Record*
  This shows the current trace record being processed and optionally by expanding the window, past trace records.
- *Number of nodes*
  This display shows the actual number of nodes of the simulation.
- *Order*
  This enables the user to specify the ordering of processes in the displays or to select a particular subset of them. Automatic reordering ensures that a given process is not multiply displayed.
- *Scroll*
  This option allows control of the scroll of a particular display over the time. There are fives alternatives: smooth, jump 1/4, jump 1/2, jump 3/4 and jump 1.
- *Scale*
  This controls the scale factor of the displays. There are four options: 64, 128, 256 and 512.
- *Reset*
  This resets the current display.
- *Pause/Resume*
  This option allows the user to stop the simulation and study a particular snapshot of it, resuming the execution when desired.
- *Step*
  This enables the user to study the execution of the simulation step by step.
- *Set Options*
  This enables the user to modify the start and end times, the time step, the printer for screen dumps, and the trace image dump file names.
- *Save Layout*
  This option allows to user to save the actual layout of the ParaGraph's displays in order to re-draw them in another session.
- *Menu*
  This is ParaGraph's main menu. Display windows may be selected or de-selected by toggling their corresponding buttons, and replay of the trace may be started, stopped or single-stepped. A number of display options may be adjusted, and screen-dumps may be taken.

2.3.3.3 A case study using ParaGraph

The LU matrix decomposition algorithm used in this section was proposed by Geist and Romine in 1988 and implemented by Ortega [46]. This algorithm, known as CSPR (Column Storage Row Pivoting), builds the LU decomposition using row pivoting on

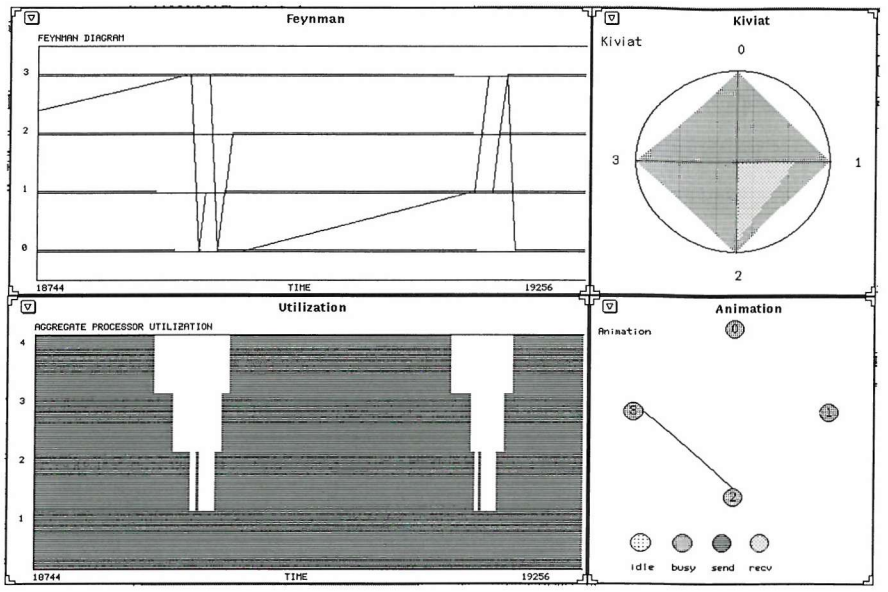

Figure 2.18: LU Algorithm - Part I: All processors are busy (n=100).

a matrix distributed by columns in a wrap-around fashion. Ortega's version of the algorithm was implemented on a ring of transputers with a copy of the program on each processor. This implementation has two subroutines and one main routine. The first subroutine, *urand*, is a random number generator; the second, *backslv*, solves a triangular system of equations. The main routine, *main*, controls the LU decomposition process.

ParaGraph has been shown to be a useful tool to help users to understand the behaviour of their application. In figures 2.18, 2.19 and 2.20 the three different stages of the LU decomposition algorithm can be seen.

This implementation of the algorithm made by Ortega, first computes the matrix to be solved in order to generate a solution where all entries of the right hand side vector (usually called b) are 1.0. For this reason, this first part is not considered for tracing and therefore the trace facility is off during this stage. The first part of the algorithm computes the actual *pivot* and makes the corresponding updates on the matrix (part I, figure 2.18). This part shows the best behaviour and all processors are busy almost all the time, except when they are communicating data in order to decide the value of the pivot and to make updates.

In part II (figure 2.19) the situation is different. The algorithm is finishing and the amount of work to be done per processor is not enough to keep them busy. The interval of time between communications is therefore becoming shorter.

In part III (figure 2.20), the algorithm is collecting all the partial results. This part is dominated by communications and only one or two processors are working at the same time (in this implementation, communications do not overlap with computation). ParaGraph displays; Feynman, Utilization, Kiviat and Animation, were very helpful in understanding the different stages of this LU decomposition algorithm.

2.3.3.4 Comments and Conclusions

- ParaGraph has been shown to be a useful tool to help the user to understand the

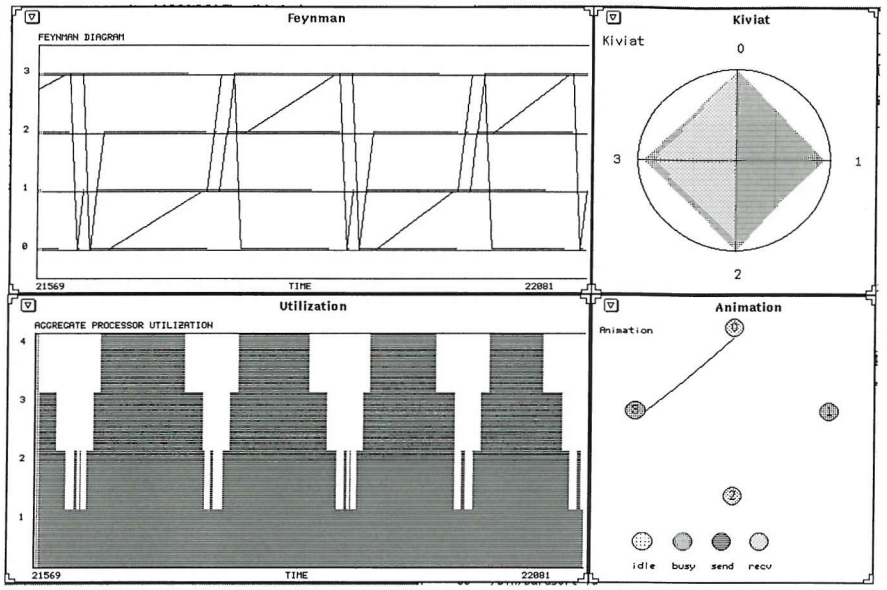

Figure 2.19: LU Algorithm - Part II: Communications start to increase (n=100).

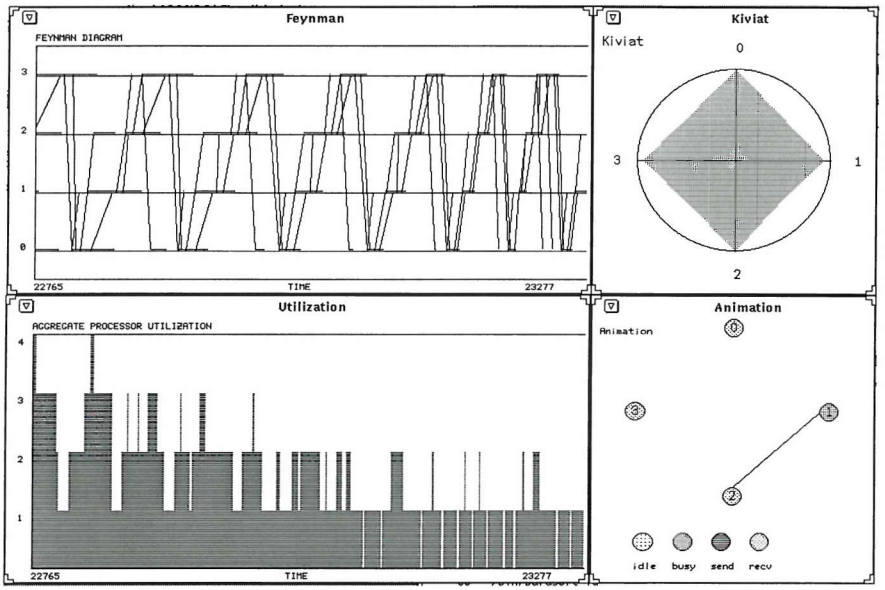

Figure 2.20: LU Algorithm - Part III: Gathering results, communications are the dominant factor (n=100).

behaviour of an application. The following displays were used in the analysis of the LU decomposition algorithm: Feynman, Utilization, Kiviat and Animation.

- It is not straightforward to find out the particular interval of time to be analyzed. This situation is even worse when the application has a long execution time. It would be worth to have some kind of *find start* operation like Express [11].
- There is no direct relation between ParaGraph's displays and the source code.
- This version of ParaGraph does not provide statistics in order to compare, for example, the percentage of idle time per processor. It is important to see the dynamic behaviour of an application, but it is also important to obtain *summary* information about that behaviour, allowing the user to focus his/her attention on a particular aspect of interest.

2.3.4 Summary and further references

Some other tools can be found in the literature. They cover performance analysis, performance measurement, visualization and tuning of parallel programs among other features.

The tools and references in alphabetical order are:

- Chitra, Visual Analysis of Parallel and Distributed Programs in the Time, Event and Frequency Domains, Abrams et al [47].
- Faust, An Integrated Environment for Parallel Programming, Guarna et al [48].
- Instant Replay, LeBlanc and Mellor-Crummey [49].
- IPS-2, The Second Generation of a Parallel Program Measurement System, Miller et al [50].
- JEWEL, A Distributed Measurement System, Lange et al [51].
- Maritxu, Generic Visualisation of Highly Parallel Processing, Zabala and Taylor [1].
- The Massively Parallel Monitoring System, Tourancheau et al [42].
- Monit, A performance Monitoring Tool for Parallel and Pseudo- Parallel Programs, Kerola and Schwetman [52].
- Mtool, An integrated System for Performance Debugging Shared Memory Multiprocessor Applications, Goldberg and Hennessy [53].
- Parasight Programming Environment, Aral and Gertner [54].
- PAWS, A performance Evaluation Tool for Parallel Computing Systems, Pease et al [55].
- PIE, A programming and instrumentation environment for parallel processing, Segall and Rudolph [56].
- Poker Parallel Programming Environment, Notkin et al [57].

- Prism Programming Environment, Sistare et al [3].
- PTOPP, A Practical Toolset for the Optimization of Parallel Programs, Earl McClaughry [58].
- Quartz, A Tool for Tuning Parallel Program Performance, Anderson and Lazowska [6].
- Start/PAT, A Parallel Programming Toolkit, Appelbe et al [59].
- TIPS, A Transputer-based Interactive Parallelizing System, Wagner et al [60].
- TMP, A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems, Haban and Wybranietz [61].
- TOPSYS, A run time tool for visualizing message-passing parallel programs, Bemmerl et al [62].
- TRACEVIEW, A trace visualization tool, Malony et al [63].
- TRAPPER, TRAffonic Parallel Programming EnviRonment, Schäfers et al [64].
- Upshot, An X-based graphic program for displaying the information captured in a logfile during the execution of a parallel program. It emphasizes the static display of detailed information. Upshot has proven useful in tuning parallel programs running on a moderate number of processors [65].
- VMMP, A practical tool for the development of portable and efficient programs for multiprocessors, Gabber [66].

Further references for performance metrics:

- *A case study using ParaGraph*, Roger Hockney [67].
- *Performance Parameters and Benchmarking of Supercomputers*, Roger Hockney [68].
- *Parameterization of Computer Performance*, Roger Hockney [69].
- $f_{1/2}$ *: A parameter to characterize memory and communication bottlenecks*, Roger Hockney and I. Curington [70].
- *A framework for benchmark performance analysis*, Roger Hockney [71].
- *Parallel Computers 2: Architectures, Programming and Algorithms*, Roger Hockney and C. Jesshope [72].
- *Reevaluating Amdahl's law*, J. Gustafson [73].
- *Measuring parallel processor performance*, Allan Karp and H. Flatt [74].
- *Modeling the serial and parallel fractions of a parallel algorithm*, E. Carmona and M. Rice [75].

- *Benchmarking parallel programs in a multiprogramming environment - the PARBENCH system* W. Nagel and M. Linn [76].
- *Toward a better parallel performance metric*, X. Sun and J. Gustafson [77].
- *Toward a taxonomy of performance metrics*, J. Worlton [78].
- *The integration of application and system based metrics in a parallel program performance tool*, J. Hollingsworth et al [79].
- *Parallel Program Performance Metrics: A comparison and validation*, J. Hollingsworth and B. Miller [80].
- *Measuring parallelism in computation intensive scientific/engineering applications*, M. Kumar [81].
- *Selective Monitoring Using Performance Metric Predicates*, C. Fineman and P. Hontalas [82].
- *Measuring the business of a transputer*, G. Jones [83].
- *Bridging the gap between Amdahl's Law and Sandia Laboratory's results*, X. Zhou [84].
- *Interpreting parallel processor performance measurements*, H. Jordan [85].

Further general references for performance analysis:

- *Experiences with Monitoring and Visualising the Performance of Parallel Programs*, K. İmre [7].
- *Performance Evaluation of Parallel Programs in Parallel and Distributed Systems*, Bernd Mohr [40].
- *What to Draw? When to Draw? An essay on Parallel Program Visualization*, Barton Miller [13].
- *Distributed Performance Monitoring: Methods, Tools and Applications*, R. Hofmann et al [86].
- *A Visualization System for Parallelizing Programs*, C. Dow et al [87].
- *The Performance Realities of Massively Parallel Processors: A Case Study*, O. Lubeck et al [88].
- *Performance Visualization for Parallel Programs*, E. Lusk [89].
- *The RP3 program Visualization Environment*, D. Kimelman [90].
- *Visualization of Program Performance on Concurrent Computers*, D. Rover et al [91].
- *Simulation and Visualization Tools for Link-based Parallel Architectures*, E. Luque et al [92].

- *Performance Tools*, K. Nichols [93].
- *Performance-Measurements Tools in a Multiprocessor Environment*, H. Burkhart and B. Miller [94].
- *A Petri net approach for performance oriented parallel program design*, A. Ferscha [95].
- *Understanding parallel program behavior through Petri net models*, G. Balbo et al [96].
- *A methodology for performance analysis of parallel computations with looping constructs*, A. Kapelnikov et al [97].
- *A methodology for performance evaluation of parallel applications on multiprocessors*, D. Menasce and L. Barroso [98].
- *Timing parallel programs that use message passing*, N. Karonis [99].
- *Parallel performance of applications on Supercomputers*, C. Grassl [100].
- *Portable execution traces for parallel program debugging and performance visualization*, Alva Couch and David Krumme [101].
- *Monitoring Parallel Executions in Real Time*, Alva Couch and David Krumme [102].
- *Visual-Aural Representations of Performance for a Scalable Application Program*, Joan Francioni and Diane Rover [103].
- *Debugging Parallel Programs using sound*, Joan Francioni et al [104].
- *The sounds of parallel programs*, Joan Francioni et al [105].
- *A Methodology for Visualizing Performance of Loosely Synchronous Programs*, S. Sarukkai et al [12].
- *A Dataflow toolkit for visualization*, D. Dyer [106].
- *Multiprocessor Performance*, E. Gelenbe [107].
- *Introduction to Parallel Computing*, T. Lewis and H. El-Rewini [108].
- *Performance analysis of parallel processing systems*, R. Nelson et al [109].
- *Visualizing performance debugging*, T. Lehr et al [110].
- *Performance measurement for parallel and distributed programs: A structured and automatic approach*, C. Yang and B. Miller [111].
- *Performance Instrumentation and Visualization*, M. Simmons and R. Koskela [112].
- *Instrumentation of Future Parallel Computing Systems*, M. Simmons, R. Koskela, and I. Bucher [113].

- *Parallel Programming: A Performance Perspective*, S. Thakkar [114].
- *An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors*, R. Fowler et al [115].
- *Instrumentation and Performance Monitoring of Distributed Systems*, R. McLaren and W. Rogers [116].
- *Multiprogramming and the Performance of Parallel Programs*, M. Benten and H. Jordan [117].
- *Performance Measurement Instrumentation for Multiprocessors Computers*, R. Carpenter [118].
- *Hybrid Performance Measurement Instrumentation for Loosely-Coupled MIMD Architectures*, J. Roberts et al [119].
- *Monitoring of Distributed Real-Time Systems: The Versatile Timing Analyzer*, W. Kastner and U. Schmidt [120].
- *Understanding the behaviour of parallel systems*, Peter Capon [121].
- *Workshop on Performance Measurement and Visualization of Parallel Systems*, Rainer Klar [122].
- *Monitoring Parallel Programs Running in Transputer Networks*, Umberto Villano [123].
- *A performance analysis exemplar - Parallel Ray Tracing*, D. Jensen and D. Reed [124].
- *Analyzing parallel program executions using multiple views*, T. LeBlanc et al [125].
- *An Overview of Common Benchmarks*, Reinhold Weicker [126].
- *Special Issue on Petri Net Modeling of Parallel Computers*, Giovanni Chiola [127].
- *Tools for Multiple-CPU Environments*, Warren Harrison [128].
- *Model-driven Validation of Parallel Programs Based on Event Traces*, P. Dauphin et al [129].

2.3.5 Comparisons and Conclusions

In order to compare the set of tools described in previous sections and make some conclusions, the following issues will be considered:

- *Run-time vs. Post-mortem*
  Most of the tools that have been mentioned in this document are postmortem tools, that means tools which work with trace data generated by a previous run of the application being studied (PABLO, SIMPLE, and ParaGraph are postmortem tools).

  There is one clear advantage of using a run-time tool: only one execution of the application is needed (at least in theory) to obtain and analyze the current

performance of the application (for some systems the analysis of trace files implies another execution of the program). However, some disadvantages were found: it is not always possible to repeat an event of interest; when the elapsed time of an application is long enough and the user has no idea about what to analyze (i.e. which breakpoints to use to stop the application when it is needed), it could be difficult to follow a run-time display and, finally, there is a lack of perspective due to the fact that the views are only *partial* or *local* views.

In a postmortem environment it is often possible to repeat events of interest (e.g. ParaGraph), and to change the perspective of the current view of the system in time (e.g. changing the scale of the display in ParaGraph). In this way, the analysis can be done in a more flexible way, examining the data as many times as needed. The amount of information provided by a postmortem tool is also more specific, because a postmortem tool has the possibility to allow a certain amount of invasiveness while recording the performance data that a run-time system cannot accept.

In general, a postmortem tool provides more flexibility than a run-time tool and most of the existing performance analysis tools for parallel programs are postmortem, including recent and sophisticated developments (e.g. PABLO). However, for large volumes of data run-time systems (e.g. Paradyn [130]) are a more convenient choice.

- *Performance Data Representation: Visual and Aural*
  Visualization is the most popular way to represent performance data. All of the tools reviewed here have visual representations of the data, offering a wide range of displays (e.g. Kiviat, Feynman, Gantt, Dial, etc). However, other forms of data representation have started to appear. PABLO uses sound as an alternative way to represent certain kinds of performance data. Other authors (e.g. Francioni et al [103, 104, 105]) have also been working in this field.

  Most researchers are convinced that visual displays are necessary to help the user in understanding the behaviour of a parallel application, but they are not convinced that sound would help. However, it is wise to test alternative ways to represent data (i.e. use of sound) especially to overcome current problems of performance data representation on massively parallel machines with hundreds or thousands of processors.

- *Portability*
  Portability is a key issue for the success of any performance analysis tool. However, some of the tools presented here have some kind of platform dependence (e.g. trace format, hardware architecture). ParaGraph can only read the trace data generated by PICL and more recently by PARMACS and Express (Southampton's trace file converters). Although it is possible to obtain data from applications that have been run on a variety of architectures, ParaGraph is restricted to the information included in the PICL trace standard. In contrast, SIMPLE and PABLO are not dependent on the trace format. SIMPLE tools use the TDL/POET library to access the trace data and PABLO has a Self Defining Data Format (SDDF), where all the characteristics of the data are represented. In this way, SIMPLE and PABLO can analyze *any* trace data generated by *any* machine.

Other levels of portability are generally fulfilled by the use of the X standard and Motif for the graphical system. However, the only tools that offer *true* portability (i.e. independence of the performance trace data) are SIMPLE and PABLO.

- *Extensibility*
  Extensibility means that it is possible and straight-forward to add new features to the performance analysis tool and that users are able to incorporate their own ideas into it. In this sense, only PABLO and SIMPLE are really extensible (although the most recent version of ParaGraph has some extensibility features [10]). SIMPLE is a set of tools which use the TDL/POET interface to access the performance trace data and PABLO is designed as a toolkit for the construction of performance analysis environments. In both cases it is easy to add new features, but PABLO has a more robust design for this particular purpose.

- *Scalability*
  Scalability is another key characteristic of the new generation of massively parallel systems; by adding processors one can incrementally increase performance without replacing existing hardware or changing the underlying software. However, new ways to represent performance data are required in order to produce useful and understandable displays for thousand of processors.

  The latest version of ParaGraph has 512 as the maximum number of processors to be represented. However, it is not clear that the displays are still useful for this number of processors. In some cases, there is not an explicit maximum number of processors, but the displays cannot work beyond a certain point (e.g. Smart tool in SIMPLE, Etool in Express [11], PATOP in TOPSYS [131, 62] and TimeMap in PA-TOOLS [132]).

  Some ideas like, for example, using *clusters* or *classes* of processors instead of individual processors, might be useful, but more research is needed in this area. Another aspect of scalability, that has not been tested but certainly very important, is that the performance analysis tool itself must be scalable to work with even larger trace files. If a tool works very well for small traces but only very slowly for larger ones, it is unlikely that the user will continue using it.

- *Ease of use*
  Most of these tools are in general easy to use. However, some are easier than others. ParaGraph is very easy to use, but unfortunately it does not provide graphical help (this could make things easier for the user). PABLO is easy to use once the user has understood its approach. SIMPLE also requires some training.

  In some cases a tool is easy to use but not *straight-forward* to use. Ease of use must be balanced against the power of the tool. We can expect a complex but powerful tool to be more complicated to use than a simple and not very powerful one.

- *User levels*
  A powerful tool must be extensible, but should also provide a set of basic features for the novice user. Thus it is desirable for a tool to have different user levels (e.g. novice, intermediate and expert). An expert user will wish to modify and add new features to the system depending on the application being analyzed. PABLO and SIMPLE fulfill this requirement.

- *Use of colours to indicate range of numerical values*
  There are many ways to represent numerical values in a visual form but one of the most popular is using some kind of colour scale. Typically, a light colour represents a small numerical value and a dark colour a large numerical value.

  This way of representing data is helpful when the amount of data is considerable. Another use of colours is to establish a difference between events or stages (e.g. green means that the processor is busy and red that it is idle). The use of colours is provided by all the tools. It would be helpful to define a standard use of colours in order to avoid confusion between systems.

- *Programming Model*
  Several tools are restricted to message passing systems. However, languages like HPF (High Performance Fortran) do not include communications statements explicitly. HPF is based on data parallelism and it could become a standard for high performance computing. For these reasons, it is important to separate the event collection process and the analysis of the performance data. A tool must be general enough to avoid any dependency on the communication model.

  When a tool is dependent on the information and semantics of a specific trace data format (e.g. ParaGraph), it is unlikely to be flexible enough to support different models. PABLO and SIMPLE have a performance data format without embedded semantics, and can be extended to support different models.

- *Possibility to interact with other tools*
  Since a perfect tool does not exist, it is sometimes useful to use a combination of tools. However, the lack of a standard for the representation of performance data makes this option difficult and several *converters* (e.g. PICL $\Rightarrow$ SDDF (PABLO), PARMACS $\Rightarrow$ PICL (ParaGraph)) have been written.

  However, if the problem was only due to a particular *format* it would be relatively easy to solve. The major problem arrives when there is a lack of information and no translation is possible. Again, PABLO and SIMPLE are most likely to be able to interact with other tools due to their portable data format (SDDF and TDL).

- *Availability of ad hoc queries by the user*
  The performance analysis of a parallel program is an iterative process. The user usually wishes to ask questions about the performance, using the feedback to produce even more interesting queries.

  The key factor here is to allow the user to build queries about the performance data in as flexible a way as possible. Although it is useful to provide some basic facilities (e.g. it is always useful to know the percentage of idle time per processor), it is also worth allowing users to use their imagination in asking any kind of question about the data. We cannot forget that the user is the *expert* of the application being analyzed.

  SIMPLE allows users to *program* their own queries using the TDL/POET interface. In contrast, PABLO provides a performance data analysis environment where the user could build a *data analysis graph* which will produce the required answer.

- *Performance of the Performance Tool*
  It is desirable for a performance analysis tool to have a reasonable performance. If this is not the case, the user would spend more time answering even simple questions about the data. In extreme cases, the user would decide to avoid the use of the tool.

- *Mechanisms to handle the execution and display of data*
  Experience using several tools suggest the following features to handle the execution and display of data:

  - Possibility to go forward and/or backward in time.
  - Access to a specific point in time or where a specific event occurs.
  - Re-execution of events.
  - Control of the speed of the execution by the user.

- *Performance Analysis Methodology*
  In order to successfully complete the performance analysis process, a methodology is required. However, none of the tools presented in the previous sections suggest a particular methodology. The use of a methodology would save time and also give ideas to the user about what to do. Such a methodology would have to be flexible, understandable, portable and extensible.

- *Relation between performance data and source code*
  Once a possible bottleneck have been found, it is often difficult to relate the problem to the source code of the application in order to know *where* the problem is located. It would be desirable from the user point of view, to *click* the mouse over a specific display where the problem is being represented and then obtain the source file and the line where the program is currently executing. However, this has not been achieved by any of the tools described here. In general, the relation between performance data and source code, depends on the user's ability and his/her knowledge about the problem. Some features, like user defined events, could be used in order to facilitate this effort.

3 Review of ANDES

ANDES is a performance monitor designed for MIMD distributed memory machines that inserts additional code in the program to be analyzed [24, 25, 133, 134]. ANDES was developed at Universidad Simón Bolívar (Caracas, Venezuela) as a MSc Thesis with Prof. Alejandro Teruel as supervisor. Further improvements have been done at the University of Southampton.

ANDES determines the following metrics: speedup, efficiency, experimentally determined serial fraction, percentage of idle time per processor, load and communication balancing, synchronization time and percentage of cpu-communication overlap. Within ANDES, a performance analysis methodology is proposed.

A prototype of ANDES was designed to analyze programs written in parallel INMOS ANSI C running on bidirectional ring of T800 transputers.

3.1 Description of ANDES

ANDES is a software monitor which analyzes the performance of a parallel program by inserting additional code in order to get information related to the events that define the performance of the program. This information is distributed among all the processors in the network and is local to each one. Once the application ends, ANDES keeps the results generated and gives the user the possibility to analyze them *postmortem*.

The development of ANDES took approximately one year, considering that there already existed an ANSI C parser to build the abstract tree related to the program being analyzed [135]. The ANDES prototype was written in parallel INMOS ANSI C [136] and has around 1300 lines of code.

ANDES has three main modules:

- *Configurer*: This allows the user to introduce particular hardware characteristics like topology, communication channels, etc. Using this information, ANDES can decide how to transmit the results of the analysis at the end of the process. For this prototype, the topology was restricted to a bidirectional ring and the configurer reduced to a configuration file.

- *Preprocessor*: Once the data related to the system configuration has been specified, the preprocessor inserts code in the program being analyzed in order to get information about the events that define performance. This information is accumulated on each processor until the application ends and the results are then collected.

  Each event occurrence is registered and accumulated in a "dynamic call tree" of the application. The main routine of the application is the root of the tree and when one routine calls another one, a new node is created and added to the tree. This new node links the "father" with his "son".

Each node of the tree contains a register with the information associated with that node. There is one tree per processor and at the end of the application, these trees are saved in order to be postprocessed.

- *Visualizer*: This provides graphical facilities to visualize the results obtained. For this version of ANDES, the visualizer was not included.

ANDES has the following restrictions:

- All the parallel programs to be analyzed by ANDES must terminate successfully, including all its constituent processes.
- ANDES is not designed for real time applications.
- Due to the fact that ANDES is a software monitor, it introduces some degree of *invasiveness*[1] [137].
- The user must specify the particular characteristics of the topology where ANDES will run. For this prototype, the topology is a bidirectional ring.

3.1.1 Performance Metrics

The main goals of a parallel performance analyzer are:

- To determine the performance of the program: This allows the programmer to know whether his application is executing efficiently. In this way, the programmer may decide whether it is necessary to carry out a deeper analysis.
- To make projections and define scalability: Once the performance of the application is known, it is important to ask the following question: how would the application behave if the number of processors increases?
- To find bottlenecks: If the performance is not as expected, it is necessary to find the modules of the program where the possible problem is located. Only the modules of the application that are consuming a substantial amount of time need to be considered.
- To obtain enough information in order to know and improve the degree of parallelism of the program: Some node architectures, like transputers, allow concurrent computation and communication. This freedom allows the maximum degree of parallelism on one particular node to be obtained. However, it is important to determine if the total amount of time of cpu-communication overlap is significant with regard to the total execution time and whether the cpu-communication overlap has been fully exploited. In this way, it is possible to improve the execution time of the program by optimizing the cpu-communication overlap and maintaining each processor as busy as possible.

With these objectives in mind, this section describes some of the most useful metrics to analyze parallel programs: speedup, efficiency, experimentally determined serial fraction [74] and percentage of cpu-communication overlap. The remaining

[1] Overhead produced by the measuring mechanism.

metrics: percentage of idle time per processor, load and communication balancing and synchronization time are described in detail elsewhere [24].

Speedup is the elapsed time of the best sequential algorithm divided by the elapsed time required by the parallel version on p processors. Efficiency is related to price-performance. It is defined by the ratio between the speedup and the number of processors. A small efficiency indicates waste of resources. In general, increasing the number of processors should reduce the elapsed time of the application, but by what factor? This is measured by the speedup. Speedup close to linear is good, but how close to linear is good enough? The serial fraction answers this question.

This section describes the two most interesting metrics measured by ANDES: the experimentally determined serial fraction and the percentage of cpu-communication overlap.

3.1.1.1 Experimentally determined serial fraction

The theoretical serial fraction of a program is the part of the program that must be executed sequentially. To calculate this serial fraction in an analytical way is a very complex problem, because there is not an easy way to determine for each program which fraction of it is executed serially and which fraction is not.

For this reason, Karp and Flatt [74] introduce the definition of experimentally determined serial fraction (serial fraction from now on), which is an empirical estimation of the theoretical serial fraction. The serial fraction is defined by the following equation:

$$f(n,p) = \frac{\frac{1}{A(n,p)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

where $A(n,p)$ is the speedup for a problem size of n on p processors.

One of the advantages of the serial fraction is its capability to make projections of the performance of the program using a larger number of processors and thereby determine the scalability of the algorithm.

The serial fraction is a good diagnostic tool to detect anomalies in performance. While speedup and efficiency change when the number of processors increases, the serial fraction should remain constant in an ideal system (problem size remains the same). Small variations are much easier to detect from something that should be constant than from something that varies.

3.1.1.2 Percentage of cpu-communication overlap

Computation and communications do not always occur in a disjoint way. When an algorithm is doing only communication, the processor is practically idle. This time may be used to do some computations independent of the data being transmitted or received. Computing the percentage of cpu-communication overlap, allow us to determine whether the program is using the maximum parallelism possible on one node. If this is the case, the processor would be busy most of the time doing useful things; if not, there may be a problem. This indicator also helps to define the degree of parallelism of a program more precisely.

3.1.2 Performance analysis methodology

In order to successfully complete the performance analysis process using ANDES as a tool, a methodology is required. The main objective of the following methodology is to analyze the performance of a specific program and to determine if this performance can be improved. The main steps of the methodology are:

- Step 1: Evaluate the application's performance.
  - Metrics:
    * Speedup
    * Efficiency
  - Description: The first thing to be determined when analyzing the performance of a parallel application, is whether this performance is good enough. In the case of SIMPAR[2], the results obtained using four transputers were compared with MPSX, a commercial Simplex solver, running on an IBM3090 where the linear programming problems are solved. In this way a comparison can be made between the performance of a parallel application and an excellent sequential version (e.g. if the results obtained using 16 transputers are equal in time with the corresponding ones on the IBM3090, then an equivalent solution, but 100 times cheaper, has been found!)[3].

    When analyzing speedup and efficiency, it is important to notice the following "problem indicators":
    * Speedup is not a linear function of the number of processors.
    * Speedup grows linearly until a certain number of processors, but from that point begins to decrease.
    * The number of processors used in the analysis (e.g. 2 and 4 processors) is too small to estimate with enough precision whether the parallelization of the algorithm was successful. With too few comparison points erroneous conclusions can be reached.

    Although speedup and efficiency allow performance evaluation, these metrics are not sufficient to determine what caused this behaviour if the speedup is not as expected.

- Step 2: Make projections and estimate the extent to which performance can be improved.
  - Metrics used:
    * Serial fraction
    * Amdahl's law
  - Description:

[2]SIMPAR is a joint effort between Maraven S.A. (Venezuelan Oil Industry) and the Universidad Simón Bolívar (Caracas, Venezuela). The main goal of this project is to build commercial software to solve sparse linear programming problems on a transputer based platform using a PC as a front-end. In [134] there is a description of the methodology and studies made in order to evaluate SIMPAR.

[3]This comparison was requested by our sponsor MARAVEN since they were interested in obtaining shorter turn-around times at significantly lower costs

  * Compute the serial fraction for p=1,..,P (where P is the total number of processors). Extrapolate this information to project the performance for any number of processors.
  * Apply Amdahl's law, using the serial fraction, and compute the maximum speedup possible for p processors.

  It could be difficult to decide whether the speedup of an application is good enough (for example, is a speedup of 70 on 100 processors good enough or not?). For this reason, the serial fraction and Amdahl's law are used. Through these metrics it is possible to know which part of the program (the serial or the parallel fraction) grows faster when the number of processors is increased (again, the problem size remains constant).

- Step 3: Analysis per routine.
  - Metrics used:
    * Load and Communication balancing
    * Percentage of communication time
    * Percentage of non communication time (complement of the communication time)
    * Percentage of cpu-communication overlap
    * Percentage of synchronization time between processes
  - Description:
    * Locate which are the heaviest used routines in the application. These are the routines consuming the largest amount of time.
    * Determine specific bottlenecks.

    If the speedup is not as expected and the serial fraction confirms some kind of additional overhead (perhaps due to communication, synchronization or load balancing), then it is necessary to specify: what exactly are the problems, what are the possible causes of these problems, which routines generate these problems and in what proportion do these routines contribute to the total execution time of the application.

- Step 4: Apply steps 1,2 and 3 and evaluate results.
  - Description: Once the problems are identified and the improvements have been applied, the evaluation process starts again. This iterative evaluation process continues until the performance is satisfactory or until no further improvements can be gained.

3.1.3 A case study using ANDES

In order to evaluate ANDES with a real problem and validate the methodology proposed, Acosta and Fernández [138] implemented a parallel version of the *Gibbs Sampler* algorithm and analyzed it using ANDES.

The *Gibbs Sampler* algorithm determines a curve that represents the function associated with a given collection of initial data using statistical methods. This algorithm has a high degree of parallelism, because the load balancing is good and

also because there are no communications until the end of the program to gather the final results.

The results obtained by Acosta and Fernández using direct measurements, were the same as those obtained by ANDES. The speedup was located between 3.72 and 3.87 for 4 processors.

The lowest speedup, 3.72, was generated by one of the most representative problems. The serial fraction was 2% but the percentage of idle time per processor was 4% on average and the load balancing was 0.96 (the maximum possible is 1). These results indicate that the load balancing is very good. However, as there is no communication throughout the algorithm, it seems to be contradictory to have a serial fraction of 2%. One possible answer, which prompts an interesting discussion, is that the *Gibbs Sampler* has a random execution time depending on the initial data. Moreover, the sequence of instructions executed by the algorithm could change depending on the initial results obtained. This means that a problem that has been executed on 1 processor is not exactly the same problem as the one executed on 4 processors, with the latter requiring more computation. As a result, there is a slight effect on the serial fraction.

As regards the precision of the measurements made by ANDES, the error was 14% in the worst case (synchronization time between processes) and 1% in the best case (communication time). Observations on codes for other algorithms, including an LU parallel decomposition algorithm, are described in [24].

3.2 ANDES Enhancements

This section describes the latest improvements of ANDES including a version running with PARMACS [43] and the implementation of VisANDES, a prototype of a visualization tool for ANDES based on the Unix facility *gnuplot*.

3.2.1 ANDES on PARMACS

ANDES has been successfully extended in order to run with the PARMACS communication macros [43]. This extension allows ANDES to analyze programs running on any topology, making it more portable. However, only two PARMACS communication primitives are being supported for measuring: SENDR and RECVR, which are the synchronous version of send and receive. It is important to remember at this point, that ANDES does not support asynchronous communications in its measurements and therefore it is not possible to support these primitives. ANDES on PARMACS will not keep track of any PARMACS primitive other than SENDR and RECVR.

ANDES has been modified without a big effort, showing that it is suitable for this purpose. These modifications were made basically on the code which determines what routines will be *substituted* by *new definitions*. Also the gathering of the performance data at the end of the execution was modified, making it easier since there is no worry about the topology.

3.2.2 VisANDES

This prototype allows a graphical interface with ANDES based on *gnuplot*. gnuplot is a command-driven interactive function plotting program. If files are given, gnuplot loads each file with the load command, in the order specified and gnuplot exits after the last file is processed. Here are some of its features: plots any number of functions,

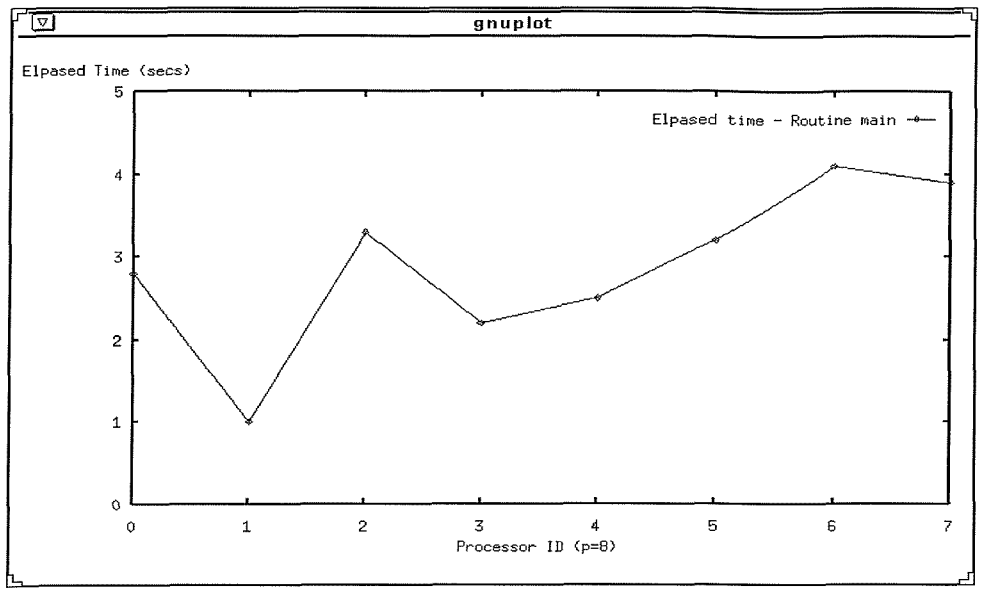

Figure 3.1: Elapsed time per processor (p=8)

built up of C operators, C library functions, support for plotting data files, and comparisons between actual data and theoretical curves. Currently, there are 5 displays automatically generated by VisANDES:

- Elapsed Time for a particular routine (figure 3.1).
- Processor Activity - busy time (figure 3.2).
- Comm., Waiting and Comm/Compt. Time vs. Elapsed Time (figure 3.3).
- Speedup vs. Number of processors (figure 3.4).
- Serial Fraction vs. Number of processors (figure 3.5).

A trace filter is also provided. Using this option, the user can generated a subset of the original trace file with only some particular fields previously specified. In this way, the user can generate his/her own pictures of the data or post-process it in a different way.

3.2.3 Tabular Output

The current version of ANDES produces three types of output files:

- *andes.out*: is the original readable trace file generated by ANDES (figure 3.6).
- *andes.trc*: is a new compact version of the trace file, which contains only data. This is in order to make the post-processing of the trace file easier for the user.
- *andes.tab*: is a new tabular output which shows data that have been previously filtered. This data is showed in a *percentage* format (figure 3.7).

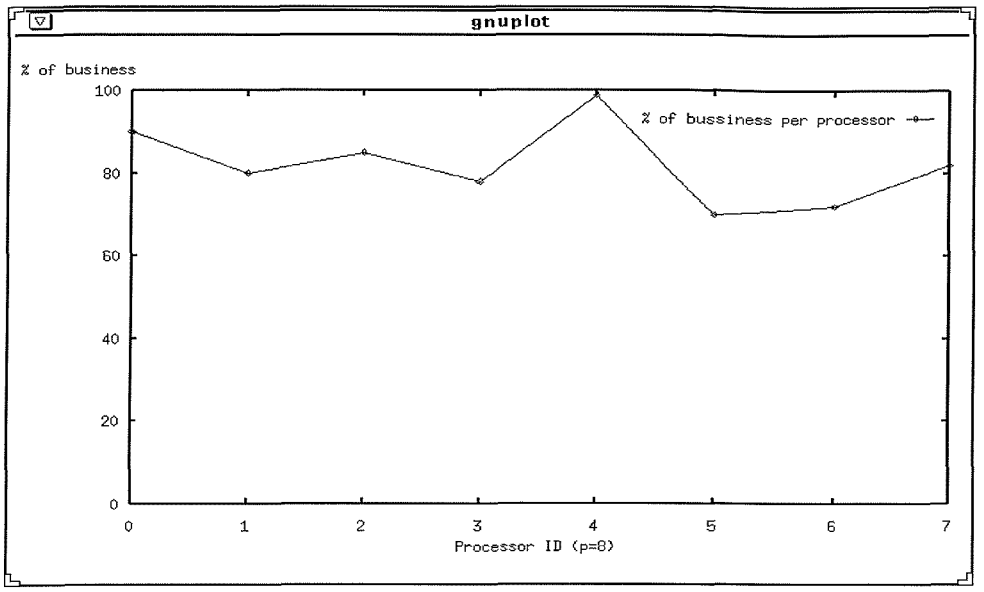

Figure 3.2: Percentage of business per processor

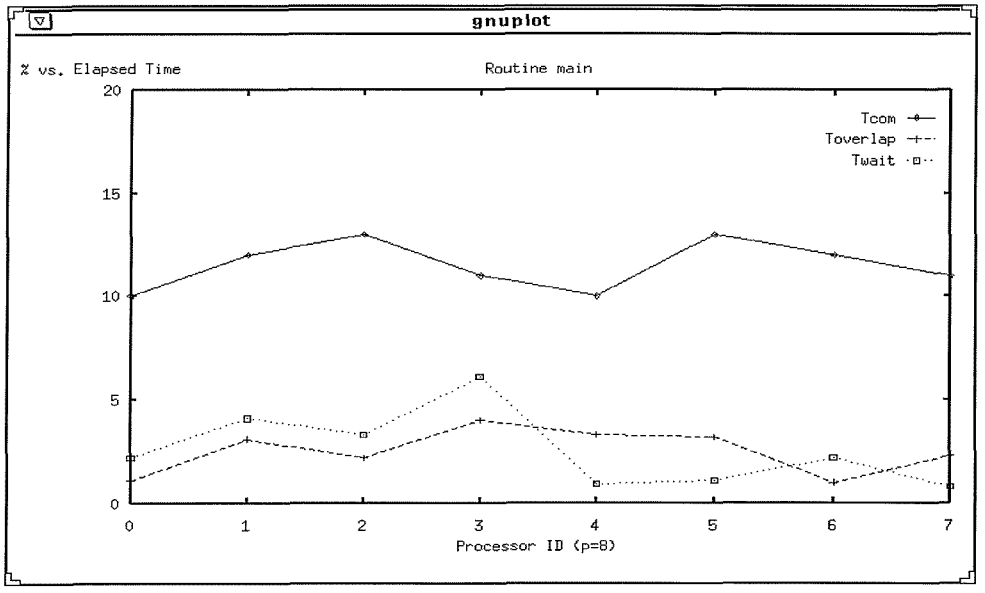

Figure 3.3: Percentage of Tcom, Twait and Toverlap vs. Elapsed time

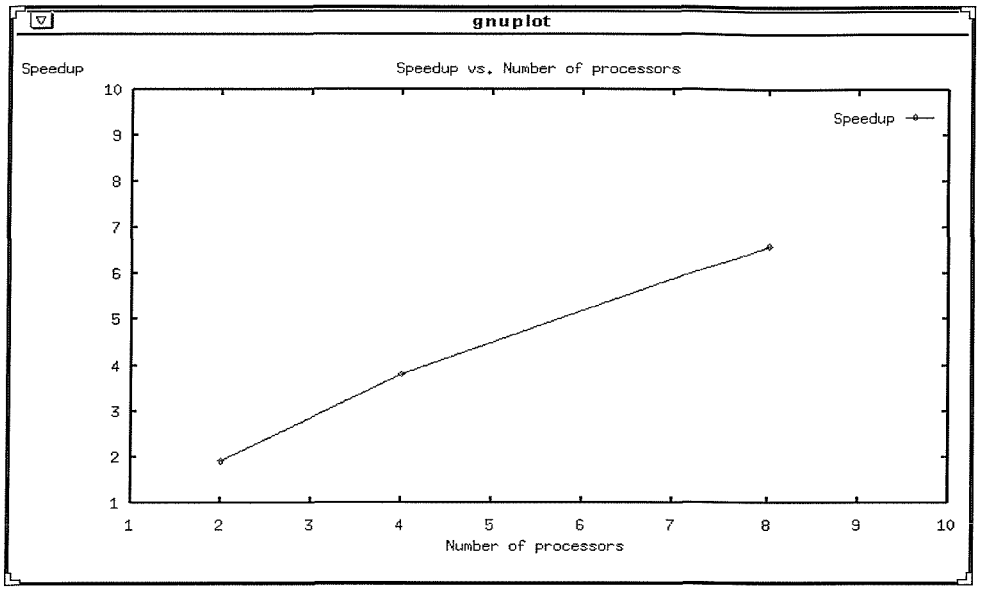

Figure 3.4: Speedup vs. Number of processors

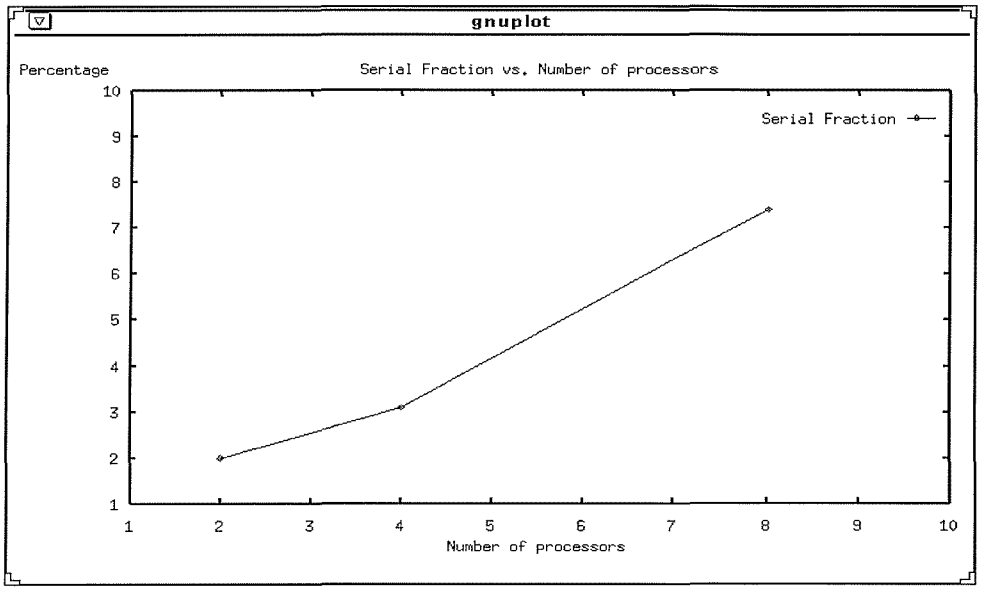

Figure 3.5: Serial Fraction vs. Number of processors

```
Data for processor 0:
--------------------
Position 0:
-----------
Function         : main
Processor        : 0
Index            : -1

==> Results <==
Telap     : 2.6109888000e+02
Tcom      : 1.1491852800e+02
Tncom     : 1.4618035200e+02
TotComPar : 0.0000000000e+00
Toverlap  : 0.0000000000e+00
InputWT   : 3.0885760000e+00
OutputWT  : 1.0936261780e+02
Total WT  : 1.1245119380e+02
==> Statistics <==
% elapsed time vs. total     : 1.0000000000e+02
% busy time                  : 57
% Communication time         : 4.4013412850e+01
% Comm/Computation Overlap   : N.A.
% Waiting time               : 9.7852970933e+01
% Input WT vs Total WT       : 2.7465924510e+00
% Output WT vs Total WT      : 9.7253407549e+01
% TotComPar vs. Tcom         : N.A.

Position 1:
-----------
Function         : urand
Processor        : 0
Index            : 0

==> Results <==
Telap     : 1.9200000000e-04
Tcom      : 0.0000000000e+00
Tncom     : 1.9200000000e-04
TotComPar : 0.0000000000e+00
Toverlap  : 0.0000000000e+00
InputWT   : 0.0000000000e+00
OutputWT  : 0.0000000000e+00
Total WT  : 0.0000000000e+00
==> Statistics <==
% elapsed time vs. total     : 7.3535359478e-05
% Communication time         : 0.0000000000e+00
% Comm/Computation Overlap   : N.A.
% Waiting time               : N.A.
% TotComPar vs. Tcom         : N.A.
```

Figure 3.6: Original tabular output generated by ANDES

Routine	Node	%Telap	%Tcomm	%Twait	%Tov	%Tov vs. Tcomm
main	00	100.00	96.70	0.00	N.A.	N.A.
Routine	Node	%Telap	%Tcomm	%Twait	%Tov	%Tov vs. Tcomm
main	01	100.00	0.03	0.00	N.A.	N.A.
Computation	01	70.27	0.00	0.00	N.A.	N.A.
Control	01	70.27	0.00	0.00	N.A.	N.A.
ReceiveMsg	01	2.39	96.37	0.00	N.A.	N.A.
SendMsg	01	2.41	97.04	0.00	N.A.	N.A.
SendMsgSeq	01	2.34	97.64	0.00	N.A.	N.A.
ReceiveMsgSeq	01	7.13	98.94	0.00	N.A.	N.A.
Routine	Node	%Telap	%Tcomm	%Twait	%Tov	%Tov vs. Tcomm
main	02	100.00	0.00	0.00	N.A.	N.A.
Computation	02	50.66	0.00	0.00	N.A.	N.A.
Control	02	50.66	0.00	0.00	N.A.	N.A.
ReceiveMsg	02	2.36	96.23	0.00	N.A.	N.A.
SendMsg	02	2.38	97.17	0.00	N.A.	N.A.
ReceiveMsgSeq	02	2.25	96.94	0.00	N.A.	N.A.
SendMsgSeq	02	2.33	97.73	0.00	N.A.	N.A.
Routine	Node	%Telap	%Tcomm	%Twait	%Tov	%Tov vs. Tcomm
main	03	100.00	0.00	0.00	N.A.	N.A.
Computation	03	33.01	0.00	0.00	N.A.	N.A.
Control	03	33.00	0.00	0.00	N.A.	N.A.
ReceiveMsg	03	2.34	96.54	0.00	N.A.	N.A.
SendMsg	03	2.31	97.21	0.00	N.A.	N.A.
ReceiveMsgSeq	03	4.48	98.49	0.00	N.A.	N.A.
SendMsgSeq	03	2.27	97.75	0.00	N.A.	N.A.
Routine	Node	%Telap	%Tcomm	%Twait	%Tov	%Tov vs. Tcomm
main	04	100.00	0.00	0.00	N.A.	N.A.
Computation	04	16.16	0.00	0.00	N.A.	N.A.
Control	04	16.15	0.00	0.00	N.A.	N.A.
ReceiveMsg	04	2.27	96.66	0.00	N.A.	N.A.
SendMsg	04	2.26	97.25	0.00	N.A.	N.A.
ReceiveMsgSeq	04	6.63	99.01	0.00	N.A.	N.A.
SendMsgSeq	04	2.10	97.54	0.00	N.A.	N.A.

Figure 3.7: New tabular output generated by ANDES. Note: N.A. stands for *Not Available.*

3.3 Comments and Conclusions

- ANDES is a software monitor that analyzes the performance behaviour of a parallel program by applying performance debugging.
- ANDES computes the following metrics:
  - Speedup
  - Efficiency
  - Serial fraction
  - Load and Communication balancing
  - Elapsed time
  - Communication and non communication time
  - Synchronization time between processes
  - Percentage of cpu-communication overlap
  - Percentage of idle time per processor
- The ANDES prototype is a useful tool to analyze the performance of parallel programs, as has been shown by the experience with the *Gibbs Sampler* algorithm [138].
- The percentage of cpu-communication overlap is a new metric proposed in this work that can be used to compute the degree of parallelism of one specific processor.
- The ANDES prototype is an "easy to use" tool. In SIMPAR, as an example, the direct measurement process took around three days including the analysis of 200 pages of numbers! However, the measurements obtained using ANDES were made in half an hour for the *Gibbs Sampler* algorithm. Even accounting for the difference in size between these two programs (SIMPAR is 10 times bigger than the *Gibbs Sampler*) the improvement is substantial. Unfortunately, we could not make any measurements of SIMPAR using ANDES due to the large size of SIMPAR's source code and the restrictions of the current ANDES prototype.
- The experience acquired in the development of this tool has contributed to a better comprehension of the behaviour of the processes executed in parallel. Moreover, it has been demonstrated that it is possible to build a useful tool in a reasonable period of time.

4 Experiments with Invasiveness

Performance is a critical issue in order to justify the use of parallel computers. However, it is usually a difficult task to write an application which successfully exploits the target parallel architecture. For this reason, many performance analysis tools have been developed (e.g. Pablo [22], ParaGraph [9], Express [11], PA-Tools [132], TOPSYS [139], ANDES [25]) in order to increase the understanding about the behaviour of parallel applications with the aim of improving their performance. These tools use different mechanisms (i.e. software, hardware or hybrid) to record events of interest related to performance.

However, these profiling mechanisms may perturb the behaviour of the application being monitored and can become a significant factor. This factor, often called *invasiveness* [137], *perturbation* [140] or *intrusiveness* [54], must be taken into account when measuring the performance of an application in order to provide more accurate information.

The purpose of this Chapter is to describe our experience determining the degree of invasiveness of PARMACS, a well known set of macros which provides a portable programming model, on four different hardware platforms (Intel iPSC/860, Parsys Supernode, Meiko CS-1 and a SUN Workstation Network) and for two different applications (the COMMS1 Genesis Benchmark and an LU Matrix Decomposition Algorithm), comparing these results with similar tools (PICL, a portable instrumented communication library [44], and ANDES, a software monitor [25]), as well as some other results recorded in the literature.

Having the knowledge of the degree of invasiveness of a particular profiling tool, it would be easier to use this tool in such a way that the invasiveness is kept low, producing more accurate results.

The following sections describe why it is important to study invasiveness, looking at its negative effects, and what is the state of the art in this field. More detailed information about invasiveness can be found elsewhere [26].

4.1 Effects due to invasiveness

In extreme cases, the insertion of costly software event collection code may mask or aggravate erroneous behaviour. This is known as the *probe effect*. The term *probe effect* has been used to describe the phenomenon that has been observed when erroneous behaviour of a program is extinguished or introduced by the insertion of a monitoring device [141]. Intuitively, the more invasive the collection mechanism, the more pronounced the effect.

The primary source of instrumentation perturbations is execution of additional instructions. However, ancillary perturbations can result from disabled compiler optimizations and additional operating system overhead. These perturbations manifest themselves in several ways: execution slowdown, changes in memory reference patterns,

event reordering, and even register interlock stalls. From a performance evaluation perspective, instrumentation perturbations must be balanced against the need for detailed performance data [140].

With the exception of passive hardware performance monitors, performance experiments rely on software instrumentation for performance data capture. Such instrumentation mandates a delicate balance between volume and accuracy. Excessive instrumentation perturbs the measured system; limited instrumentation reduces measurement detail [140].

4.2 State of the art: A Perturbation Analysis Model

Malony, Reed and Wijshoff [140] proposed a *perturbation analysis model* in order to recover the *true* trace of events as they would have been generated during an execution without instrumentation. Models to capture and remove timing perturbations due to instrumentation must be based on a particular instrumentation approach. Because tracing is the most general form of instrumentation, allowing both static and dynamic analysis, a time-based perturbation model for trace instrumentation was derived.

The results of the application of this model proposed by Malony, Reed and Wijshoff, has successfully demonstrated that global performance measures, such as total execution time, are computable to an acceptable accuracy via application of performance perturbation models. Moreover, the ability of such relatively simple models to approximate actual code execution times to within 15% from full trace instrumentations, with execution time perturbations exceeding by four orders of magnitude the unperturbed time, appears remarkable.

4.3 Measurement Tools

Other efforts have also been made in order to provide a feasible way of low-intrusive performance analysis measurement in distributed processing environments. An example is Parasight, a programming environment that is geared towards non-intrusive performance analysis and high level debugging. In Parasight, profilers execute as observer programs which run concurrently with the target program and monitor its behaviour [54].

Another example is JEWEL, a distributed measurement system [51]. JEWEL consists of a flexible toolkit for low-interference on-line performance measurement integrated with a powerful adaptable graphical presentation facility and a generic interactive experiment control system.

The last example to be mentioned is Mtool, an integrated tool for isolating performance bottlenecks in shared memory multiprocessor applications [53]. Mtool isolates sources of performance loss in two steps. In an initial pass, Mtool instruments a program by adding basic block counters. Using a knowledge of instruction latencies and the basic block counts obtained by running the instrumented program, Mtool builds an execution profile describing how much time the program spent in each basic block. This profile is used both to identify important regions of the program to instrument with performance probes and to insure that the overhead of such probes is kept to acceptable levels (on average less than 10%).

4.4 Measuring Invasiveness

The purpose of this section is to determine the invasiveness of PARMACS on the COMMS1 Genesis Benchmark for a set of four different hardware platforms: Parsys Supernode (transputer based), Intel iPSC/860, Meiko CS-1 (transputer based), and SUN Workstation Network. The reason why the COMMS1 Genesis Benchmark was selected is that although this benchmark is very simple, tracing measurements in PARMACS are made basically when a process is doing communication and the purpose of this benchmark is, precisely, to measure the basic communication properties of a computer network.

The Genesis Benchmark Suite has been developed to fulfill a need for benchmark programs that can be used to evaluate the performance of distributed-memory MIMD systems on scientific and engineering problems [142]. The COMMS1 Genesis Benchmark measures the basic communication properties of a computer network by performing the *pingpong* experiment between a neighbouring pair of nodes. A message of varying length is sent to a neighbouring node, and immediately returned after the data has become available to the receiving user program.

On the other hand, the PARMACS ANL/GMD Macros provide a portable programming model for distributed memory architectures. They are available for a variety of architectures (Cray Y-MP, Intel iPSC/2, iPSC/860, nCUBE 2, SUPRENUM, Meiko, Transputers) as well as for networks of workstations [43]. PARMACS provides portable point-to-point message passing between processes arranged in a host-node configuration. The macros were originally based on a message passing interface developed at Argonne National Laboratory for the C programming language. Four implementations of PARMACS were used for this experiment: Parsys Supernode, PARMACS based on top of VCR [143]; Meiko CS-1, PARMACS based on top of CS-Tools [144]; SUN Workstation Network, PARMACS based on top of PVM [33] and Intel iPSC/860, PARMACS based on top of NX-2 (Intel's communication library software).

4.4.1 Results

The general results for this experiment are presented in table 4.1. The experiment consisted of measuring the time to send a message of a particular length between a neighbouring pair of processors, with the tracing mechanism of PARMACS on and off, recording the difference between them.

The overhead produced by the tracing mechanism of PARMACS is very high for short messages. The worst case is 68.5% (Msg. length = 1 byte, Meiko - Transputer based) and the best value 0.4% (Msg. length = 40000 bytes, Intel iPSC/860). In general, the overhead appears to be inversely proportional to the message length. This is true for the Parsys, Meiko and Intel machines but not for the SUN Workstation Network. For this latter case, the overhead is not linear and depends on the dynamic network load (Sun-I and Sun-II represent two different executions of the experiment).

The best general behaviour is achieved by the Intel iPSC/860, followed by the Parsys and Meiko versions of PARMACS (see figure 4.1). For these three cases the invasiveness for a message length greater than 5000 bytes, is less than 3.3%.

Since tracing measurements in PARMACS are made when a process is doing communication, the results given above mean that we could expect a low invasiveness for applications with message lengths greater than 5000 bytes for these particular hardware

Table 4.1: Percentage of overhead for the tracing mechanism of PARMACS on the COMMS1 Genesis Benchmark.

Msg. Length (bytes)	*Parsys*	*Sun-I*	*Sun-II*	*Meiko*	*iPSC/860*
1	56.9	16.7	30.2	68.5	45.1
100	48.9	4.8	0.2	61.3	-
400	-	-	-	-	16.0
946	18.2	19.8	29.1	28.2	-
1000	-	-	-	-	11.5
4946	4.6	16.3	21.3	7.8	-
5000	-	-	-	-	3.3
14946	1.6	5.8	12.4	2.8	-
24946	0.9	18.0	12.3	1.7	-
40000	0.6	6.7	11.3	1.0	0.4

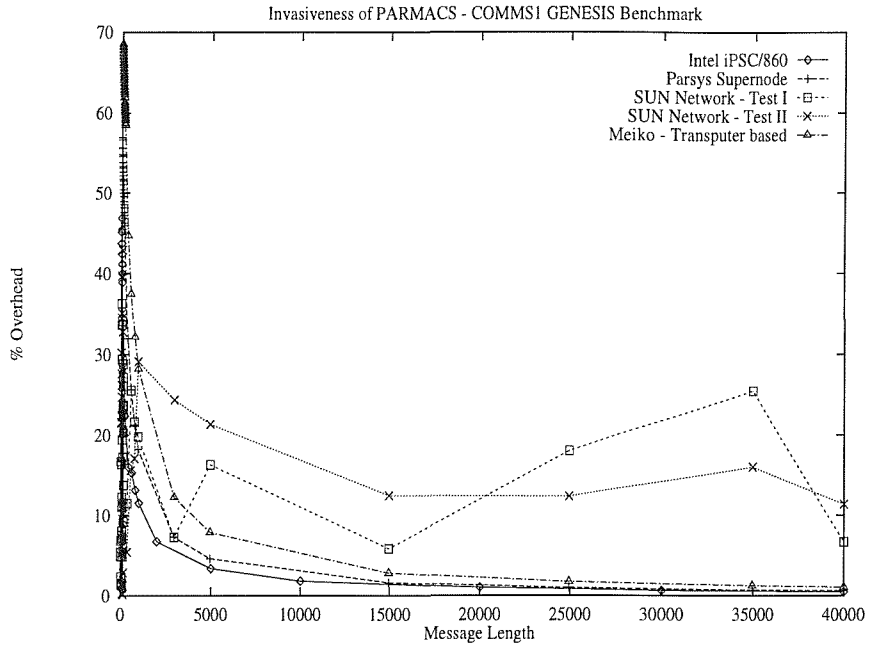

Figure 4.1: Invasiveness of PARMACS - COMMS1 Genesis Benchmark

platforms (except the SUN Workstation Network, where the results depend on the network load).

4.4.2 Comparison between PARMACS and PICL

It is useful to compare the results of the previous section with equivalent results for a similar tool. In this section, some results are presented about the invasiveness of PICL for single operations such as *send* and *receive*, measured in the same way as in section 4.4.1 (comparing the results with and without trace generation) for the Intel iPSC/860 [145]. Other results for the Intel iPSC/2 and nCUBE/3200 can also be found in [145]. PICL is a portable instrumented communication library designed to provide

portability, ease of programming and execution tracing in parallel programs [44]. An important feature of PICL is that messages are allowed to be sent between arbitrary pairs of processors, irrespective of the underlying hardware communication network.

Table 4.2 gives the time required to send one byte of data to an immediate neighbour and receive another byte back. Thus, this time represents the minimum cost associated with exchanging information. The table gives the time for this operation using the native commands csend and crecv (iPSC/860) and using the PICL routines *send0* and *recv0* in the following way: with tracing off (no-trace); t000, with tracing on but no trace records generated; t333, with tracing on and all possible trace records generated.

Table 4.2: Time spent exchanging 1 and 1000 bytes messages between neighbouring nodes on the iPSC/860 (time in microseconds).

iPSC/860-No Trace	*Trace Level*	*Time*	*% Overhead Trace/No Trace*	*Bytes sent*
113.5	t000	141.5	19.7	1
-	t333	222.9	49.0	1
814.4	t000	836.6	2.6	1000
-	t333	876.4	7.0	1000

For short messages the overhead is high in all cases, but for large messages the overhead decreases. The best value is 2.6% (Msg. Length = 1000 bytes, Trace level = t000, iPSC/860). Unfortunately, there are no more comparison points other than 1 and 1000 bytes for the message length.

For PICL, with full trace and with a message length of 1000 bytes, the overhead generated by the tracing mechanism was lower (7.0%) compared to the overhead for PARMACS on a similar benchmark (11.5%) on an Intel iPSC/860 (see table 4.1 and 4.2). If this behaviour is the same for larger message lengths, then we could expect the invasiveness to be lower for PICL than PARMACS on this particular machine. However, as it is shown in the next section, the application also has an affect on the degree of invasiveness generated by the profiling mechanism.

4.4.3 LU Matrix Decomposition Algorithm

The invasiveness of an event collection mechanism can also be affected by the particular application being monitored. For this reason a small real, and well known application, LU Matrix Decomposition Algorithm, was selected in order to test this idea.

The LU matrix decomposition algorithm used in this section, was proposed by Geist and Romine in 1988 and implemented by Ortega [46]. This algorithm, best known as CSRP (Column Storage Row Pivoting), builds the LU decomposition using row pivoting on a matrix distributed by columns in a wrap-around fashion. Ortega's version of the algorithm was implemented on a ring of transputers with a copy of the program on each processor.

This experiment was performed using a transputer version of PARMACS [143] and PICL [146], both implemented at the University of Southampton. Additionally, the invasiveness of ANDES (a software monitor), was also determined and compared against the invasiveness of the tracing mechanisms of PARMACS and PICL for this particular application.

ANDES is a performance monitor designed for MIMD distributed memory machines that inserts additional code in the program to be analyzed [24, 25]. Within ANDES, a performance analysis methodology is proposed. Although the underlying ideas of ANDES are general, the current version of ANDES (1.0) was designed to analyze programs written in parallel INMOS ANSI C [136] running on a bidirectional ring.

4.4.3.1 Results

The results of this section can be seen in figure 4.2. The matrix size is the same (n=100) for the whole experiment. As can be deduced from the graph in figure 4.2, when the number of processors increases the invasiveness of ANDES grows faster than for PICL and PARMACS (although the invasiveness of ANDES is high, ANDES is collecting more than just communication information [25]). In general, the invasiveness generated by the tracing mechanism of PICL is lower than the ones for ANDES and PARMACS. However, for PICL and PARMACS the invasiveness first increases and then decreases as the number of processors increases. This is due to the fact that the size of the matrix is not as large as required to keep every processor as busy as possible and the algorithm becomes communication bound. Therefore, using more processors does not improve the performance but generates more overhead.

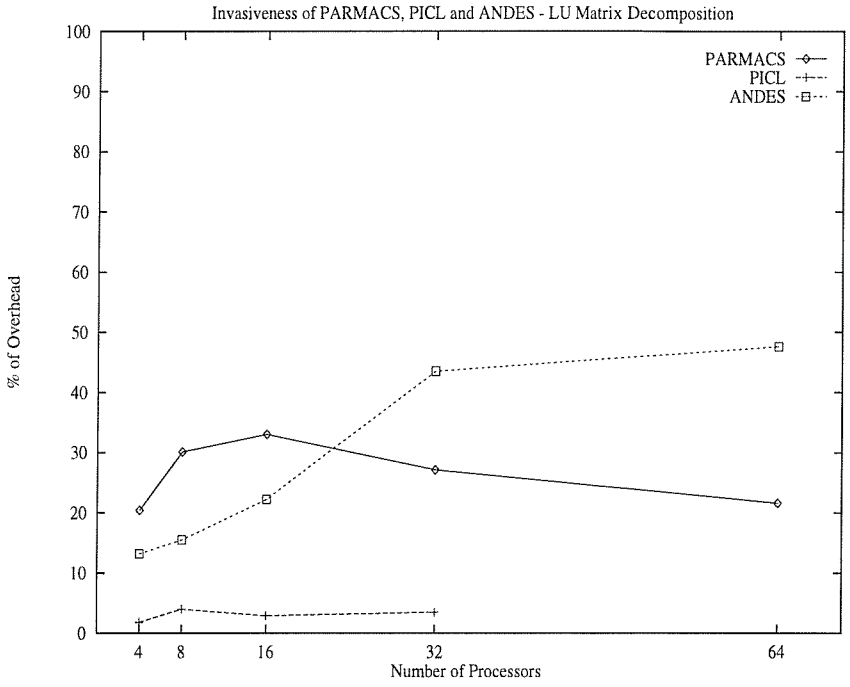

Figure 4.2: Invasiveness of PARMACS, PICL and ANDES - LU Matrix Decomposition

If these results are compared with the results of section 4.4.1, some interesting things can be found: the invasiveness of PARMACS for messages of 800 bytes (which is the size in bytes of one column of a matrix of n=100 in double precision) is around 20% for the Parsys Supernode, and this value is approximately what is obtained for the LU Matrix Decomposition Algorithm for 4 processors. However, as the number of processors increases, the invasiveness increases up to 30% but for the reason explained

above, the amount of work is not enough to keep all processors busy and there is no gain in performance but in overhead.

4.4.4 Other results

Mohr affirms in [147] that times reported in the literature for software monitoring and "normal" amount of event instructions (e.g. each entry and exit of each function) vary from 10% up to 500%. Examples are: 45% for IPS-2 [50], and 200% for TOPSYS [139]. In [148], Beier shows that due to the measurements made by the software monitor, the program execution slows down about 2%. However, a software implementation of these mechanisms of the monitoring system reduces the complexity of the hardware monitor by about 30%.

Mtool reduces the intrusiveness of software instrumentation by using an initial profile to estimate and control the overhead it is introducing [53]. Goldberg and Hennessy affirm that generally, Mtool's instrumentation increases program execution time by less than 10% while comparable existing tools (e.g. Quartz [6], IPS-2 [50]) increase it by 40-70%. Using hybrid monitoring and carefully placed event calls, these values could be reduced to 0.1 - 1% [147, 149, 150].

4.5 Comments and Conclusions

Invasiveness is an important factor to be considered when measuring the performance of a parallel application. Invasiveness represents the overhead produced by a measuring mechanism and it can affect the accuracy of the performance data generated. In this Chapter, the invasiveness of PARMACS has been determined on four different hardware platforms (Intel iPSC/860, Meiko CS-1, SUN Workstation Network and Parsys Supernode) and compared with results using similar tools. One of the main purposes of this experiment is to learn how to use profiling tools in such a way that the invasiveness is minimized, satisfying the user requirements in terms of data accuracy.

As an example of the previous statement, the results of the COMMS1 Genesis Benchmarks Suite show that we could expect a low invasiveness for applications with message lengths greater than 5000 bytes for those particular hardware platforms (except the SUN Workstation Network, where results depend on the network load).

Another way of minimizing the invasiveness, is to select the tool which has the smallest degree of invasiveness for the particular application and hardware platform, as it can be deduced from the experiment of section 4.4.2.

The application being monitored is another factor that affects the degree of invasiveness. Using an LU Matrix Decomposition Algorithm as a small, real application and comparing these results against the COMMS1 results, it is possible to see how the invasiveness is similar for an equivalent message length, but it is also possible to notice how the invasiveness increases up to 30% for the LU Matrix Decomposition Algorithm as the number of processors increases, since the amount of work is not enough to keep all processors busy and there is no gain in performance but in overhead.

There are some other issues that are important to notice. Using PARMACS on top of any other communication software like VCR, increases the overhead. However, there are no efficient implementations of PARMACS for every hardware platform (at least not at the time these experiments were made). In any case, further measurements should be done for more efficient implementations of PARMACS. Another issue is that only the

overhead in terms of execution slow down is measured as an effect due to invasiveness. In general, these results show a wide variety of values for the invasiveness depending on the event collection mechanism, the hardware platform, and the application.

One very interesting issue is to determine what values are low in terms of invasiveness. In general, values below 10% are considered acceptable if the reordering of events do not affect the final result of the application and this statement is based on empirical results (Miller [50], Goldberg [53], Mohr [147]). Greater invasiveness implies longer execution times and less accuracy.

Mohr has suggested that values for the invasiveness of the order of 0.1 to 1% could be achieved using hybrid monitoring and carefully placed event calls. However, it is clear that timing perturbations cannot be completely removed by efficient instrumentation: perturbation analysis must be applied for reliable performance prediction and resolution of tracing-generated timing errors.

Finally, further research is needed into these invasiveness effects, as well as a better understanding of perturbation models. Accurate information is crucial for an accurate performance analysis.

5 Parallel Performance Visualization

Visualization is now recognized as a fundamental element in scientific computing. As high performance computing allows us to solve ever larger problems, and as measurement technology yields ever more data to be studied, so our ability to analyze and understand this data relies more and more on visual processing [151].

Visualization has been important in the past and is becoming even more important at present. One of the advantages of visualization is that it is a way of seeing the unseen [151]. *"Imagination or visualization, and in particular the use of diagrams, has a crucial part to play in scientific investigation"* (Rene Descartes, 1637).

In this Chapter, the Do-Loop-Surface representation of the performance of a particular do-loop in a program is presented as an alternative to provide an abstract representation of performance (at least of the performance of a particular do-loop or kernel in a program), using a scientific data visualization tool (AVS) for this purpose [20, 19, 28, 152, 27].

5.1 Do-Loop-Surface: General Description

A Do-Loop-Surface (DLS), is an abstract representation of the performance of a particular do-loop in a program. This surface is created by measuring the elapsed time per iteration for every processor and representing this data as a 3D display. The three axes represent *processors*, *iterations* and *elapsed time per iteration* respectively and we will call *iteration* the execution of each cycle of the particular do-loop being analyzed. By representing a do-loop using a DLS, the user can view a complete execution of this section of the program in a single picture and looking at this display from the right perspective may uncover unknown bottlenecks [1]. A new and evocative vocabulary that explains the particular features of the display is also incorporated (e.g. *mountains, valleys, hilly, flat*). This surface is displayed using AVS [18] (Application Visualization System), which allows rotation of the figure in several ways as well as zooming when necessary and some other interesting transformations.

The main goals of the DLS representation are:

- To provide an abstract representation of program performance.
- To provide useful information for the user, allowing detection of unknown bottlenecks using 3D displays.
- To provide scalable views for programs running on a large number of processors.

By analyzing a DLS, the user can identify performance features like load balance and communication delays. The work presented in this thesis has shown that DLS

[1] A *visualization* makes some details of the view manifest, while obscuring others. A view defines *what* information is presented; a visualization describes *how* the information is displayed [125].

displays are also useful in identifying cache/memory effects [153], comparing different communication strategies, and even identifying hardware irregularities (figure 5.1 illustrates a DLS display that enabled us to discover performance differences in the floating point units of the SPARC-2 chip set on a CM-5. This figure shows that some nodes execute faster than others. Additionally, the figure shows how the user can get information from the display by "clicking" the mouse on the desired position).

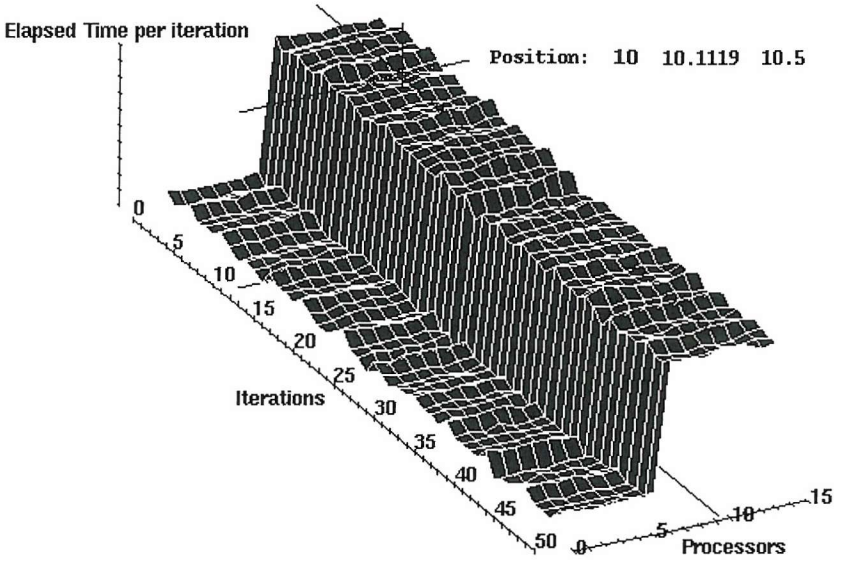

Figure 5.1: DLS display showing differences in execution time of floating point units on a CM-5.

A do-loop represents one of the most important sections of the program, since it is where the main computation generally takes place. Additionally, a programmer may understand more about do-loop structures than communication directives. A DLS is, therefore, an appropriate structure to visualize. In terms of scalability, a DLS is represented by a single picture that can be zoomed if necessary and a general view of the surface will determine which regions require a more detailed analysis. The DLS displays have been successfully scaled up to 128 processors and we expect they will scale well for a larger number of processors.

5.1.1 Trace Generation

The procedure to be followed in order to generate the necessary trace data for a DLS is straightforward. A small trace library has been implemented on top of PARMACS [43], PVM [33], and MPI [37], (both for FORTRAN and C) on the following architectures:

- PARMACS version: CM-5, PARSYS Supernode, and MEIKO CS-1.
- PVM version: CM-5, Meiko CS-2, and Sun workstations network.
- MPI version: Sun workstations network, IBM SP2.

We have chosen PARMACS, PVM, and MPI to assure portability and, therefore, access a wide range of architectures. The user has only to add a few instructions to the program as in the example in figures 5.2 and 5.3.

```
      PROGRAM host

      ...General Declarations

C     DLS_TRACE
      include 'dls_fparmacs.inc'
C     DLS_TRACE

C     Declarations and init for macros
      ENVHOST
      INITHOST

      TORUS(nprocs,1,1,'node','xx_proc_group')
      REMOTE_CREATE('xx_proc_group',procid)

C DLS_TRACE
      CALL DLSHINIT(20,1,LOOK_UP_NPROCS,MYPROC,procid)
C DLS_TRACE

      ...Body of the program

C DLS_TRACE
      CALL DLSCT(1)
C DLS_TRACE

      ENDHOST
      STOP
      END
```

Figure 5.2: Generating trace information for a DLS. HOST program, PARMACS version.

The trace library provides the following functions:

- DLSHINIT: Initialization of global variables and general setup for the host program. It has 6 parameters:
  - Trace size in Kbytes, integer.
  - Frequency of trace recording, integer. If the frequency is 1, then DLSGETT will record traces for every iteration of the do-loop. If the frequency is 100, then it will record traces every 100 iterations. This option is very useful in order to control the amount of trace information produced.
  - Number of node processors, integer.
  - Processor ID, integer.
  - Host ID, integer.
- DLSINIT: Initialization of global variables and general setup for the node program. It has the same parameters as DLSHINIT, except the last one.
- DLSGETT: Generates do-loop trace information. It must be called at the beginning and at the end of the do-loop being analyzed. It has two parameters: the

```
      PROGRAM node

      ...General Declarations

C     DLS_TRACE
      include 'dls_fparmacs.inc'
C     DLS_TRACE

C     Declarations and init for macros
      ENVNODE
      INITNODE

C DLS_TRACE
      CALL DLSINIT(20,1,LOOK_UP_NPROCS,MYPROC)
C DLS_TRACE

      ...Body of the program

C     Do-Loop to be measured
      DO k=1,niter

C DLS_TRACE
         CALL DLSGETT(1,k)
C DLS_TRACE

      ...Do-Loop body

C DLS_TRACE
         CALL DLSGETT(1,k)
C DLS_TRACE

      ENDDO

      ...Rest of program

C DLS_TRACE
      CALL DLSST()
C DLS_TRACE

      ENDNODE
      STOP
      END
```

Figure 5.3: Generating trace information for a DLS. NODE program, PARMACS version.

tag of the do-loop (several traces for different do-loops can be generated in the same run of the program), and the iteration variable. Both values are integers.

- DLSST: Sends the trace information to the host program. It must be called from the node program. No parameters.
- DLSCT: Collects the trace generated by each particular node. It must be called from the host program. It has one parameter which specifies whether or not the trace will be saved.

The only communication functions required by the library are *send* (from the nodes to the host) and *receive* (the host receives traces from the nodes). The first one is provided by *SENDR* (PARMACS), *pvm_send* (PVM) and *MPI_Send* (MPI). The second one is provided by *RECVR* (PARMACS), *pvm_recv* (PVM) and *MPI_Recv* (MPI).

The trace generated has the following format:

```
1 1 0 0.018624
| | | |
| | | --- Time stamp
| | ----- Processor number
| ------- Iteration
--------- Do-loop tag
```

The do-loop tag allows the user to collect data from more than one do-loop during the same run. Further processing can separate and reorder (by processor number) this data (program *dls_reorder*). In order to reduce trace size, the trace data may be *filtered* by using the program *dls_filter*. The *dls_filter* program can be used to remove redundant information (i.e. information that does not show us any important change in the behaviour of the algorithm) from a DLS, allowing larger data sets to be displayed on a single screen. The user specifies the filtering percentage, delta, and then if two consecutive values on the trace file differs by less than that delta value, then one of the values is eliminated from the file and only one class representative is kept (see figure 5.4). A summary of the trace data flow is presented in figure 5.5.

The overhead, or *invasiveness* [26], generated by this tracing mechanism, is low and the difference in time between running a program with and without such tracing is often negligible. In fact, the overhead generated by DLSGETT (which is the only routine that really makes an effect in the execution time of the do-loop being measured) is, approximately, 0.0001 seconds (on the CM-5, for each do-loop iteration). However, there are other systems (e.g. ANDES [25]), where the invasiveness may become a significant factor. For more details, please refer to Chapter 4.

5.2 DLS and AVS - Integration with Scientific Data Visualization

AVS, Application Visualization System [18], is a data visualization environment designed to analyze scientific and engineering data in areas like fluid dynamics, computer-aided engineering, and molecular modeling. AVS users can construct their own visualization applications, by combining software components into executable flow networks.

```
File name: gaus2.15p
Number of processors: 15
Number of iterations: 678
Filtering Reduction Value (percentage): 5.0

Filtering iterations...
Saving new data file...

Iterations filtered:  92.33 %
Min. difference (neighbours iter.):   0.00 % (proc=2, iter=282)
Max. difference (neighbours iter.):  22.98 % (proc=8, iter=156)
Ave. difference (neighbours iter.):   1.34 %
New number of iterations: 52
```

Figure 5.4: Output from the dls_filter program. In this example, the number of iterations has been reduced from 678 to 52 or 92%.

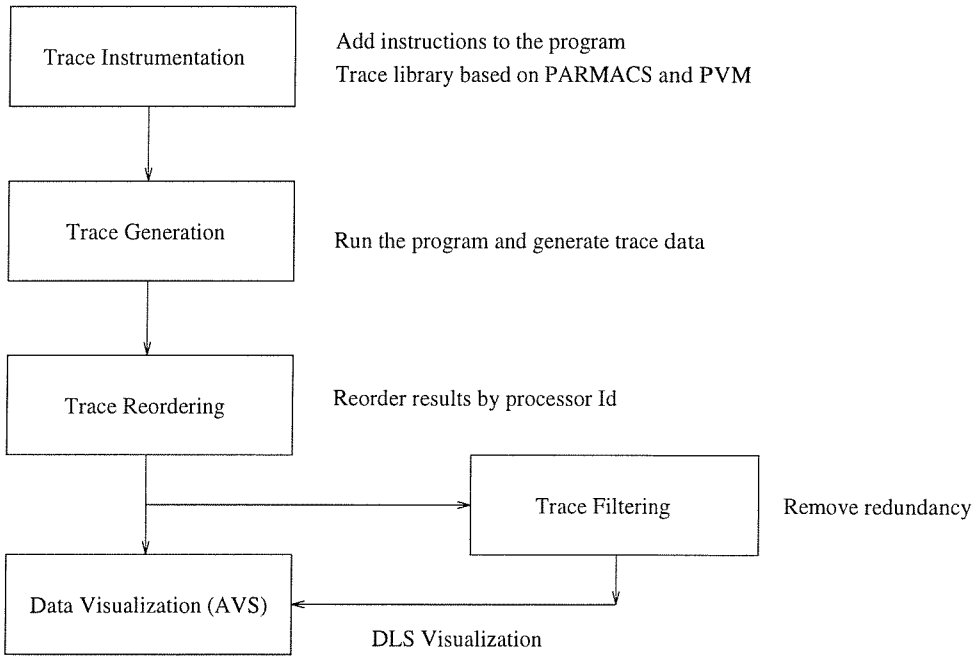

Figure 5.5: Trace data flow to produce a Do-Loop-Surface.

The components, called modules, implement specific functions in the visualization cycle (filtering, mapping, and rendering). The flow networks are built from a menu of modules by using a direct-manipulation, visual programming interface called AVS Network Editor. With this editor, the users can create their own visualizations by selecting a group of modules and drawing connections between them (see figure 5.6).

The user views, organizes, and further processes the output of a network through one of the AVS subsystems. These subsystems are:

- The Geometric Viewer: displays 3D geometric objects.
- The Image Viewer: displays 2D images.
- The Graph Viewer: creates XY and contour graphs of data.

AVS also includes a set of modules for construction of networks (figure 5.7), although it is possible for the user to create his own modules by using the AVS Module Generator. This module generator can be used to automatically create module code in C or FORTRAN and it has a visual interface. Modules are software building blocks with well-defined interfaces, written either in FORTRAN or C. A good example of these customized modules can be seen in figures 5.8 and 5.9. The figures illustrate the *dls_transform* module which is a customized module that controls the iterations and processors currently being displayed. In this way only relevant information of the DLS can be analyzed. Modules take typed data as inputs and produce typed data as outputs. The basic data types in the system are oriented toward scientific data manipulation and graphic display. These types are:

- 1D, 2D, and 3D grids of numbers with scalar values or vectors of byte, integer, or floating-point values at each grid point.
- unstructured cell data.
- geometric data.
- images.
- molecular data.

AVS implements two basic strategies for translating numerical data into colour pictures. In the pixel-based method, data points become pixels, more or less directly. In the geometry-based method, the numerical data is converted to descriptions of 3D geometric objects. There are, in turn, turned into colour images by the machine's low-level graphics software and rendering hardware. For more detailed information about AVS, see the AVS reference manual [18].

AVS has been used in this research in order to support performance analysis of parallel programs. One advantage of using AVS to represent Do-Loop-Surfaces is that no tool development is required and that every feature of the data visualization tool can be used to find unknown bottlenecks (e.g. rotating a figure may lead the user to discover something that he has not seen before, using a specific module that computes statistics about the data points displayed, etc.). AVS also has its own built-in analysis capabilities. Many analysis functions are available and the methods are scalable with respect to data set sizes and extensible in terms of applying additional analysis (or

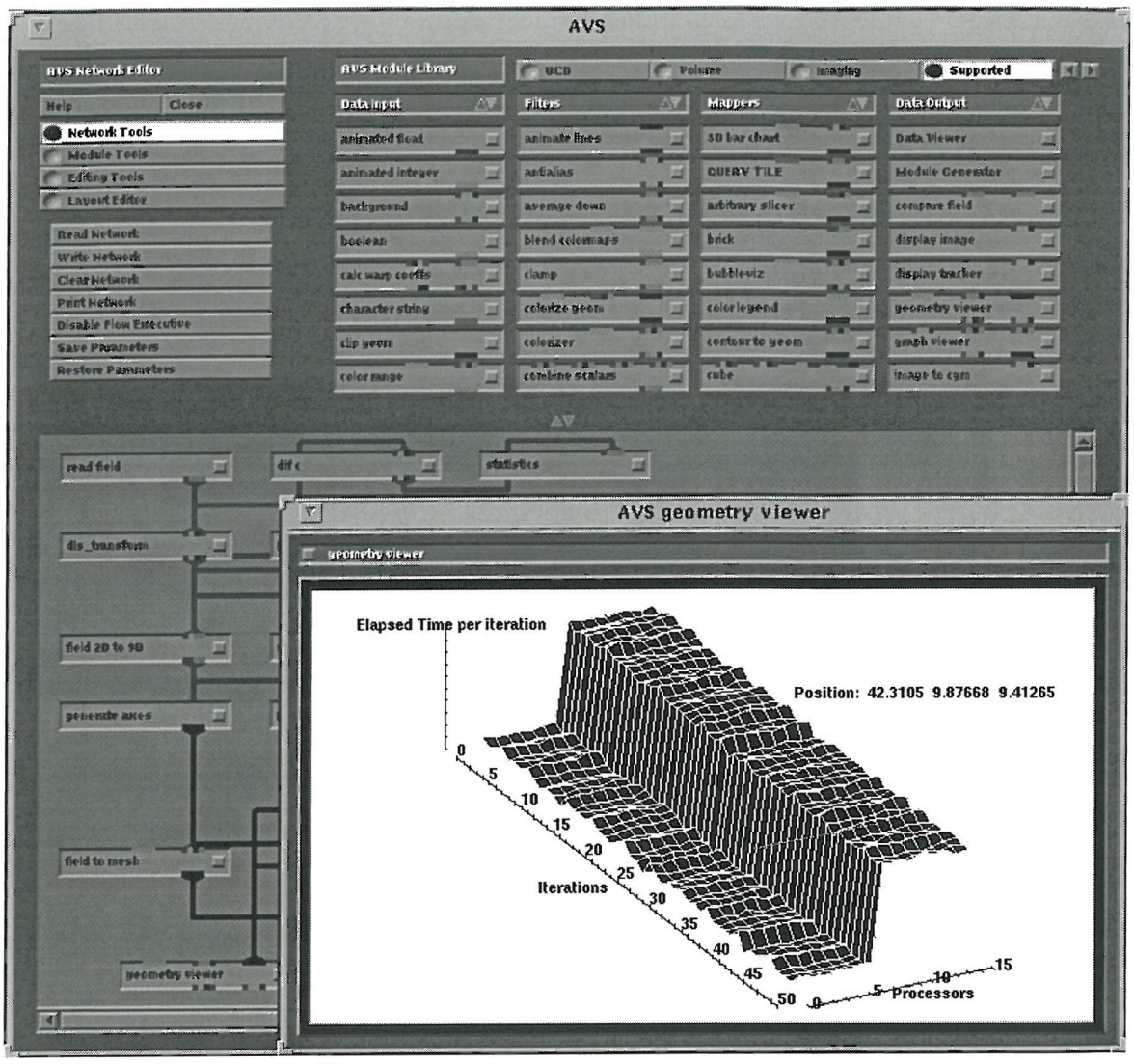

Figure 5.6: AVS Data Network generating a Do-Loop-Surface display.

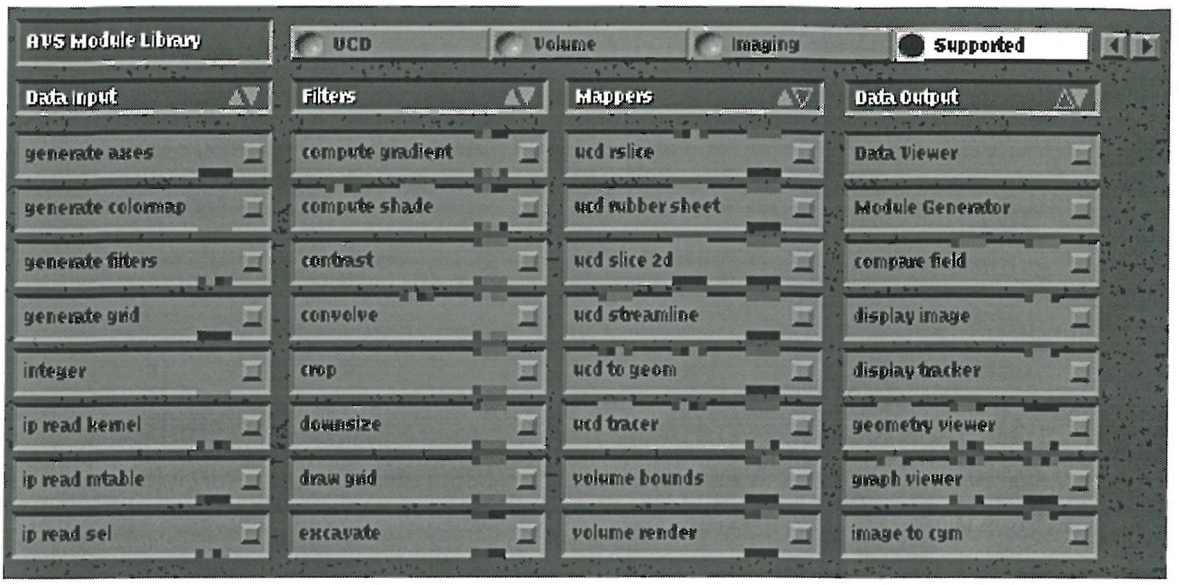

Figure 5.7: AVS Module Library. Several modules are already provided by AVS and they can be used for further data analysis.

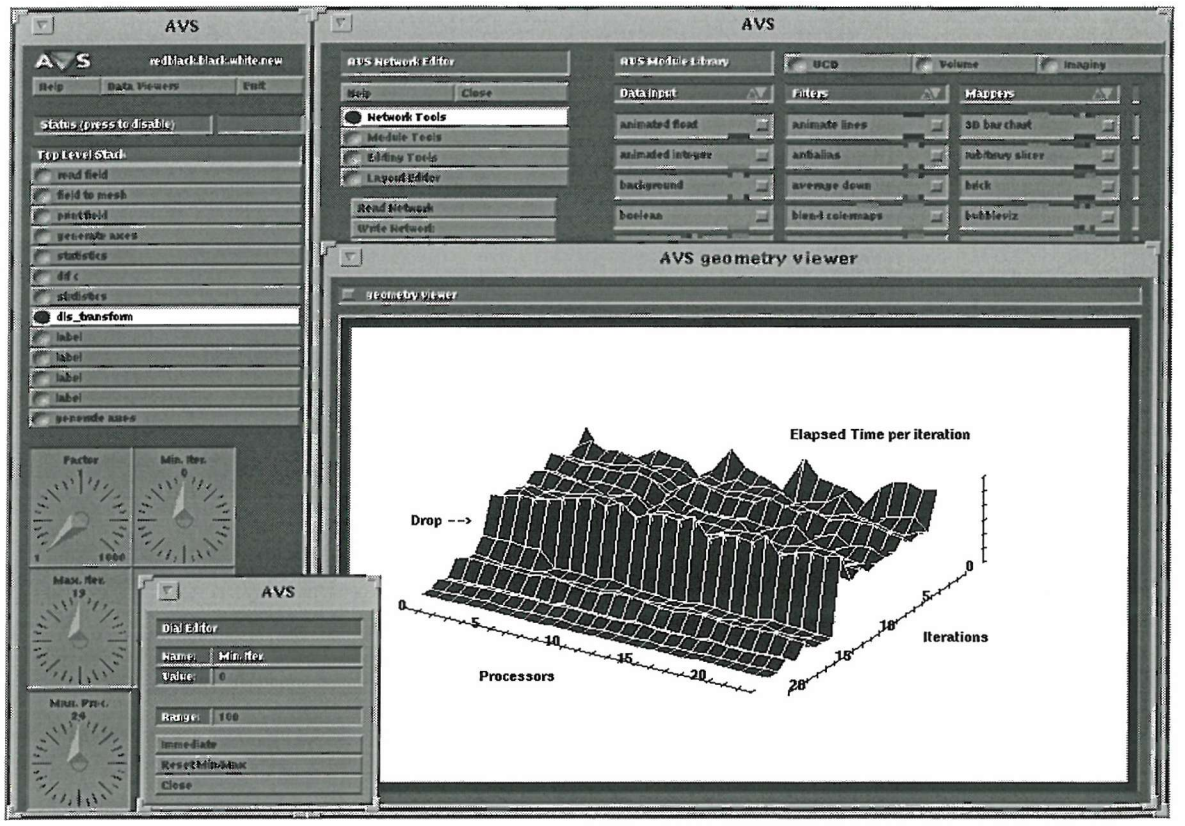

Figure 5.8: The *dls_transform* module controls the current iterations and processors being displayed. In this way only relevant information need be analyzed.

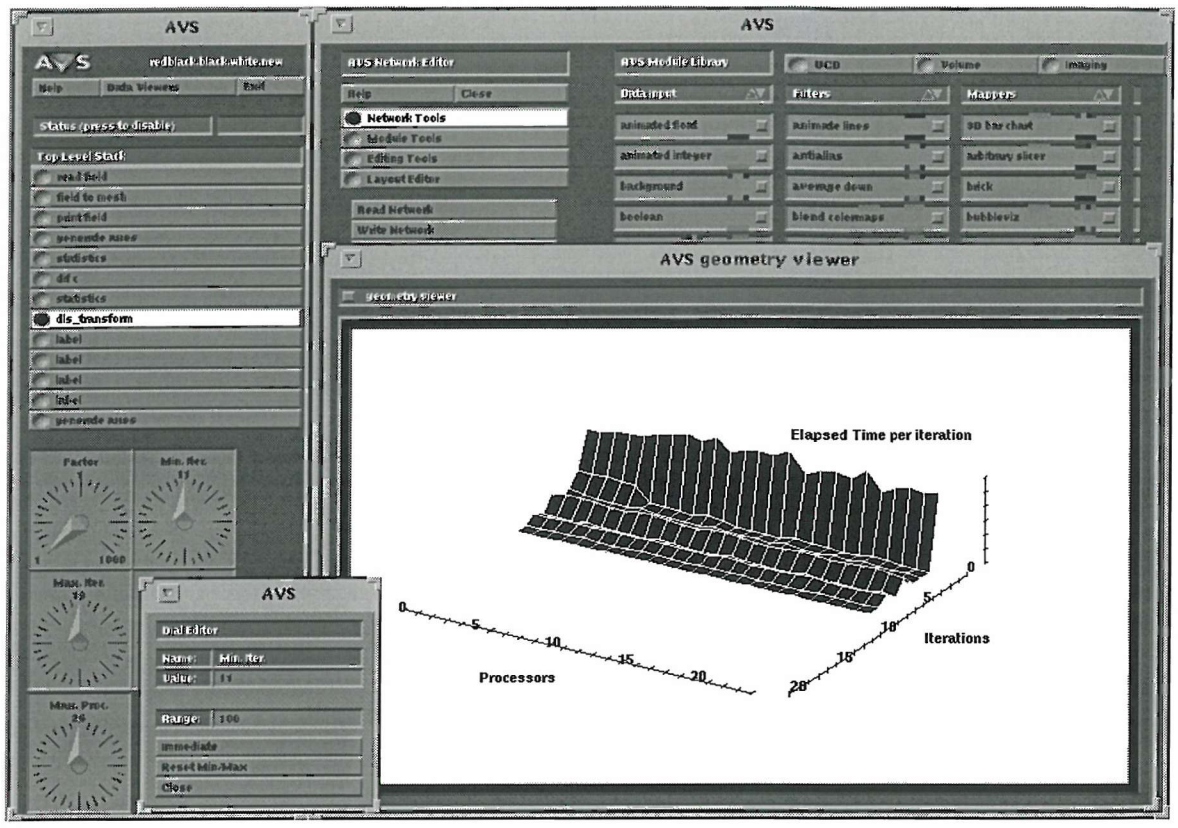

Figure 5.9: The *dls_transform* module controls the current iterations and processors being displayed. In this figure the first 10 iterations have been deleted.

programming user-specific analysis) and support multiple domains of analysis (e.g. temporal, spatial, and frequency domains). The objective of multiple-domain analysis methods is to selectively focus on a particular aspect of the performance while hiding the contributions from other aspects [21].

Additional advantages of this approach are:

- Scalability of analysis and display regardless of the amount of data.
- Extensibility of analysis.
- Data importing and exporting capabilities.
- Extensive image processing capabilities [154].

Experimentation with such tools will allow us to learn more about how to use visualization techniques to represent performance data and to design, modify, and produce new tools which incorporate this new knowledge.

5.3 Interaction with other Performance Tools

In this section, we describe our experience using the DLS displays together with two other performance tools: *Lebep* and a Performance Estimator. The purpose of this experiment is to illustrate how the above tools can be used to address the problem of improving application performance from different perspectives. The parallel benchmark generator, *Lebep* [155], provides the user with a workload specification language with which to describe proposed algorithms. Given an algorithm specification, the parallel benchmark generator produces a template program with the specified workload characteristics. This facility allows for rapid performance prototyping of different solution strategies. The parallel performance estimator [156] predicts the expected execution time of parallel programs by analyzing their use of the hardware units and simulating data movement within the memory hierarchy. This tool does not require the target hardware platform, and is able to identify the workload placed on the system hardware components.

5.3.1 Case Study - Red Black Relaxation

This case study shows how the three tools, *Lebep*, the performance estimator, and the DLS displays can be used to assist in developing efficient parallel code. The problem we consider is parallelising a sequential red/black simple over-relaxation method for a 64 node Thinking Machines CM-5. The problem size used is a 1024x1024 element matrix.

To determine the best data partitioning method and number of processors usually requires running the complete parallel program using different numbers of processors for each data partition. This is costly and in many cases impractical as it requires access to the parallel resource. The combined use of *Lebep* and the performance estimator, however, enables parallel algorithm selection and initial performance evaluation in the absence of the parallel platform. The method by which we achieved this is described below and shown in schematic form in figure 5.10.

The first stage in parallelising the sequential code is to obtain an estimate of the expected execution time on a single node, and to determine the relative use of functional

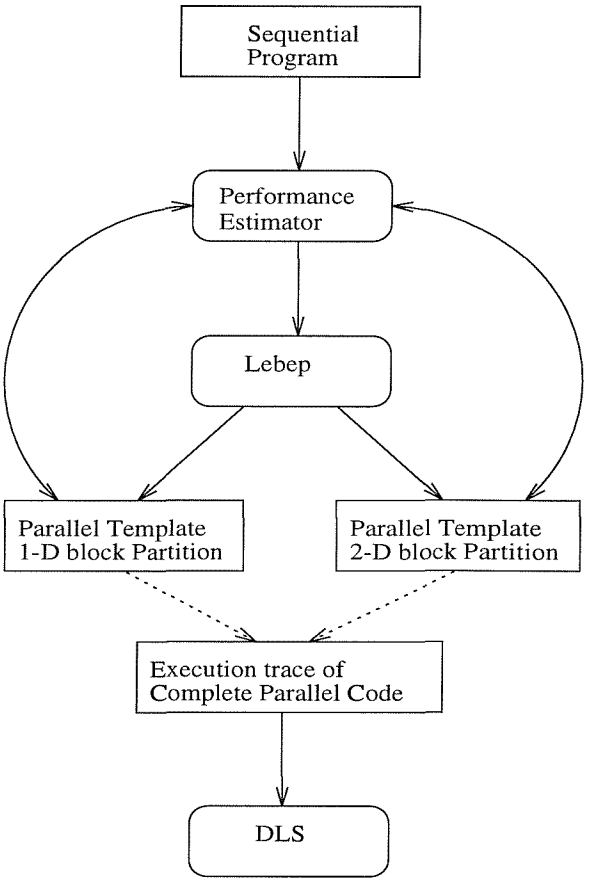

Figure 5.10: Interaction of the Performance Estimator, *Lebep* and DLS in parallelising programs.

units within the node. This is important for two reasons. First, it provides a base figure from which we can later judge the efficiency of parallel versions. Secondly, it can identify processing bottlenecks in the integer unit (IU), the floating point unit (FPU) and memory hierarchy. This latter type of information could suggest that a totally different solution method is required in order to achieve efficient processor utilization. An example of the output from the performance estimator for the sequential code on the CM-5 is shown in figure 5.11. Summarising this information, the FPU accounts for 61 percent of the total time, the IU for 10 percent and data movement for 29 percent[2]. This indicates that relatively high performance will be obtained on each processing node. One possible area of optimization could be improving on the 50 percent FP register hit rate, although the high cache hit rate does offset this substantially.

Given the sequential code analysis, we construct templates of the proposed parallel implementations using *Lebep*. The two implementations we consider are both data parallel on the 1024x1024 grid. The first of these implementations is block distributed in both dimensions; the second data partitioning strategy is block distributed only in the second dimension.

In this *Lebep* example, the code is SPMD with processors mapped to a two dimensional grid and identified by X-Y values. Edge data is exchanged between neighbouring processors in the order of East, South, West, and North. For the 1-D block distribution, edge data is exchanged between East and West neighbours only. Following the communication of boundary points, the amount of work performed on each processor is specified. This is given in terms of floating point operations, determined from the sequential code, and the percentage of floating point references that result in cache misses, which is determined by the performance estimator. Given these *Lebep* specification files, the PVM code generated by the *Lebep* translator is evaluated by the performance estimator. By using the performance estimator in this manner, the most appropriate data partitioning method and number of processors can be determined. It is notable that the most appropriate partitioning method and processor efficiency has been determined by using only template specifications in the absence of the target machine.

To verify our predictions we developed parallel implementations of both data partitioning strategies. Both implementations (including the version generated by *Lebep*) were run on the CM-5 for up to 25 processors. Figure 5.12 compares actual run times of the red/black code with the predicted code values given by the Performance Estimator, while figure 5.13 compares actual run times for the *Lebep* generated code, with the run times of the red/black relaxation code. To understand the difference between the projected and actual times, as well as these results in general, we used DLS displays.

The DLS displays for two data partitionings (5x5 and 1x25 examples) are shown in figures 5.14 and 5.15. From the pictures it is clear that the column partitioning produces a better result since the values are generally lower. Also, this DLS display is *flatter* with the only *hilly* section corresponding to initialization and synchronization in the first iteration. By comparison, the display for the 2-D block distribution (figure 5.14), shows a *drop* in execution time of about 20 percent in the 12th iteration. This is difficult to explain from algorithm analysis alone. Accordingly, we viewed the computation

[2]As the production code is compiled with the highest levels of optimization, we use the minimum possible execution time analysis provided by the performance estimator.

```
Execution Time Analysis (seconds)
---------------------------------
  - Integer Calculations : 0.130867
  - FP Calculations      : 0.784897
  - Function Calls       : 0
  - Message Passing      : 0
  - Data Access Time     : 0.367134
                           ----------
       Min Possible Time : 1.2829

  - Add. Integer Calc    : 0
  - Add. FP Calc         : 0
  - Array Subscript Calc : 2.82563 to 5.65126
                           ----------
       Max Possible Time : 6.93415

Functional Unit Usage (Percent of Max time)
-------------------------------------------
  - CPU                   : 95
  - Memory System         : 5
  - Message Passing System : 0
  - (Function Calls)      : 0

Functional Unit Usage (Percent of Min time)
-------------------------------------------
  - CPU                   : 71
  - Memory System         : 29
  - Message Passing System : 0
  - (Function Calls)      : 0

Memory Hierarchy analysis
-------------------------
                %Access Time     Hits      Misses    HitRate
                ------------   --------   --------   -------
  - Main Memory   :  64         787979          0     100
  - L1 Cache      :  36        4709908     524808      90
  - FP Registers  :   0        4184068    4188164      50
  - Int Registers :   0       15712255         23     100
```

Figure 5.11: Performance estimator output for the sequential program code on a CM-5 node.

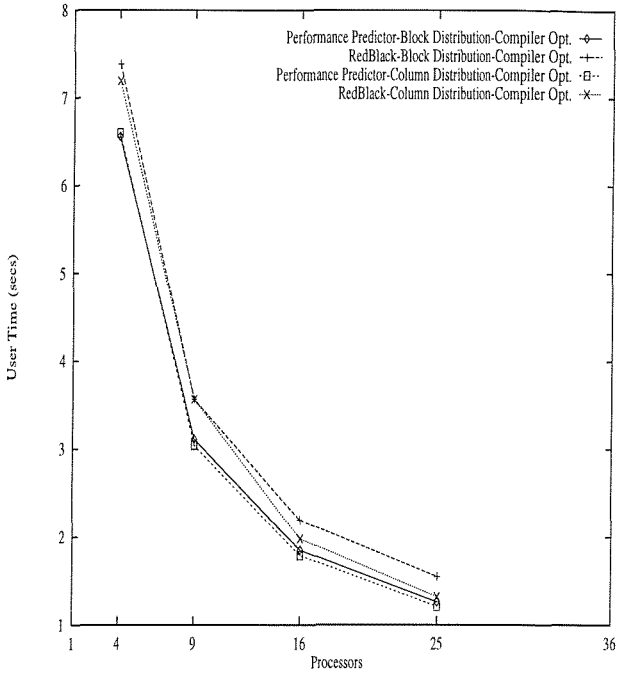

Figure 5.12: Summary of results: Comparison between estimated and actual execution times of the Red/Black Relaxation code. Notice that for both cases (performance predictor and real execution of the program) the column distribution provides better results.

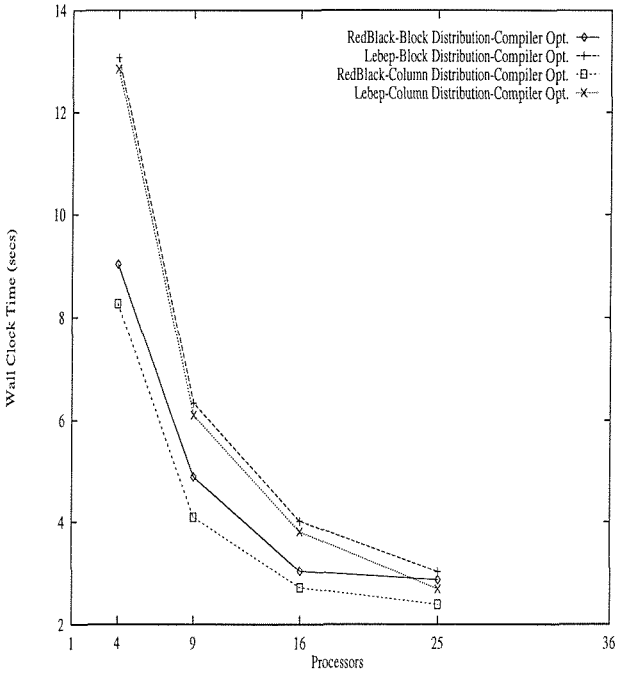

Figure 5.13: Summary of results: Comparison between the execution times of the *Lebep* templates and the Red/Black Relaxation code. Notice that *Lebep* also confirms that the column distribution provides better performance results.

section of the program in isolation using a DLS (figure 5.16). In this picture, we see that some processors are taking longer than others to complete the relaxation part of the algorithm in spite of having identical work loads.

The drop to a common plateau after the 12th iteration seems to indicate that the hills in iterations 0 to 11 are due to either initialization costs or system/administration overheads. Initialization costs of the red/black relaxation program are unlikely as this would require all processors to decrease in execution time after the 11th iteration. However, the DLS shows that some processors have a constant computation time for all iterations. A more plausible explanation is that certain administrative tasks are performed on some of the CM-5 processors soon after start up. This is easier to see if we use the DLS visualiser to represent the total time for blocks of 10 iterations (figure 5.17). The higher computation time for the first block of 10 iterations and the flattening off of the surface thereafter is clearly visible.

5.3.2 Comments and Conclusions

The DLS representation of performance, was successfully used to determine performance bottlenecks. By using DLS displays, similarities in the behaviour of the *Lebep* templates and the application were immediately apparent, despite their differences in computation and memory access patterns. This information could not be obtained from execution time alone. The DLS displays also enabled us to obtain an accurate view of the performance on the target machine. Figure 5.14, for example, helped us to identify a performance problem that it is still difficult to explain (why the execution time decreases after the 12th iteration of the algorithm), but that it is probably originated by certain administrative tasks performed on some of the nodes of the CM-5 soon after start up. It is interesting to notice that this situation is almost impossible to predict (one could say that it is not directly related to our application) but it is indeed affecting the performance of our program.

The differences in performance between processors and iterations in the 2-D block distributed case were distinct. Although these performance differences were initially thought to result from cache effects, the Performance Estimator revealed constant cache performance for all processors over all iterations. The Performance Estimator thus assisted in interpreting the DLS displays. Future plans for the DLS project will include a closer relationship with *Lebep* and the Performance Estimator, representation of cache/memory effects and experiments with larger number of processors.

Through the case study, the added value of using *Lebep*, the Performance Estimator, and the DLS displays in a collaborative manner has been demonstrated. From our experience, the current trend to develop integrated toolsets holds much promise for delivering powerful environments with unexpected benefits.

5.4 Related work

Parallel performance visualization has already been identified as a useful technique (e.g. [112], [9], and [22]). In this section, four papers related to abstract performance views and scalability are briefly discussed.

Rover and Wright [157] present VISTA, a framework to capture, process, and display performance information. In general, the views specific to VISTA are based on applying the techniques of data visualization and multidimensional graphics to

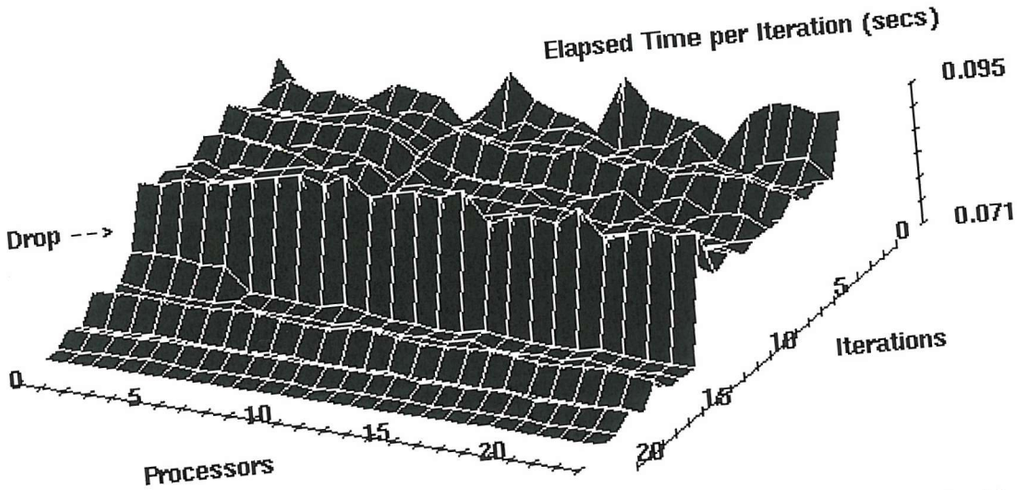

Figure 5.14: Do-Loop-Surface: Red Black Relaxation, Block Distribution, 5x5 grid of processors.

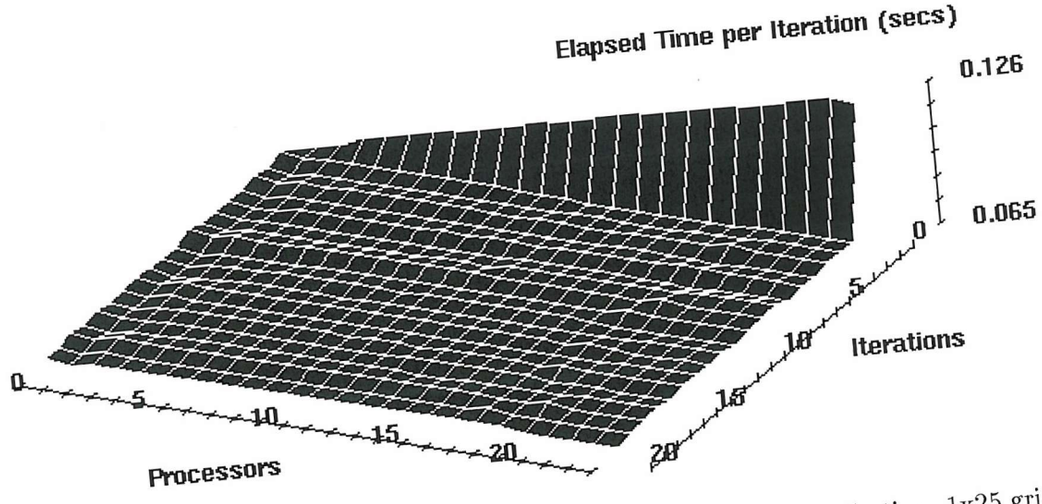

Figure 5.15: Do-Loop-Surface: Red Black Relaxation, Column Distribution, 1x25 grid of processors.

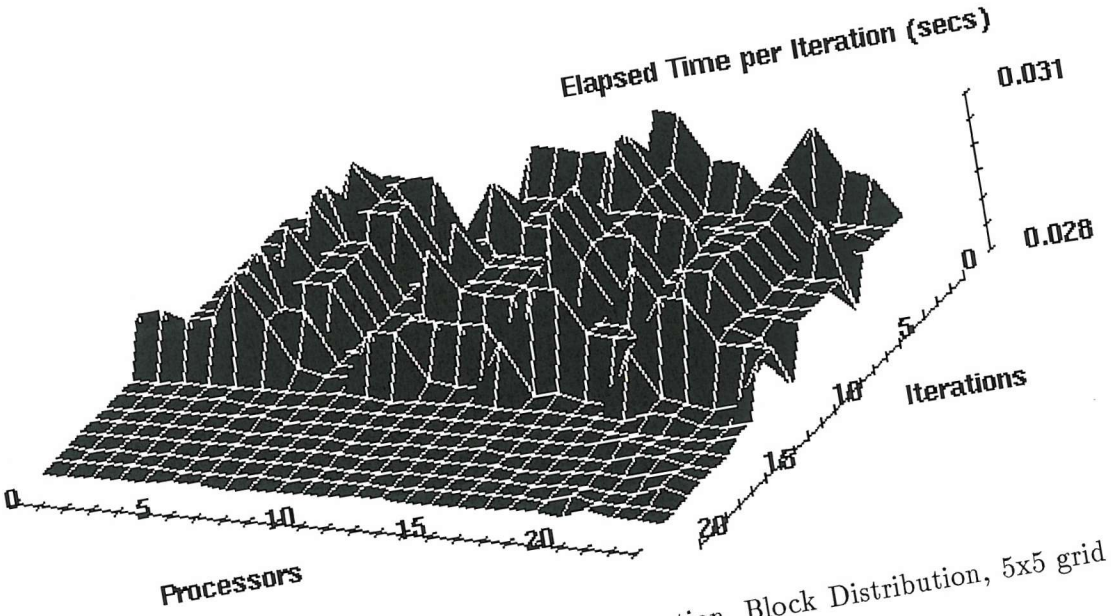

Figure 5.16: Do-Loop-Surface: Red Black Relaxation, Block Distribution, 5x5 grid of processors, CPU section of the algorithm only.

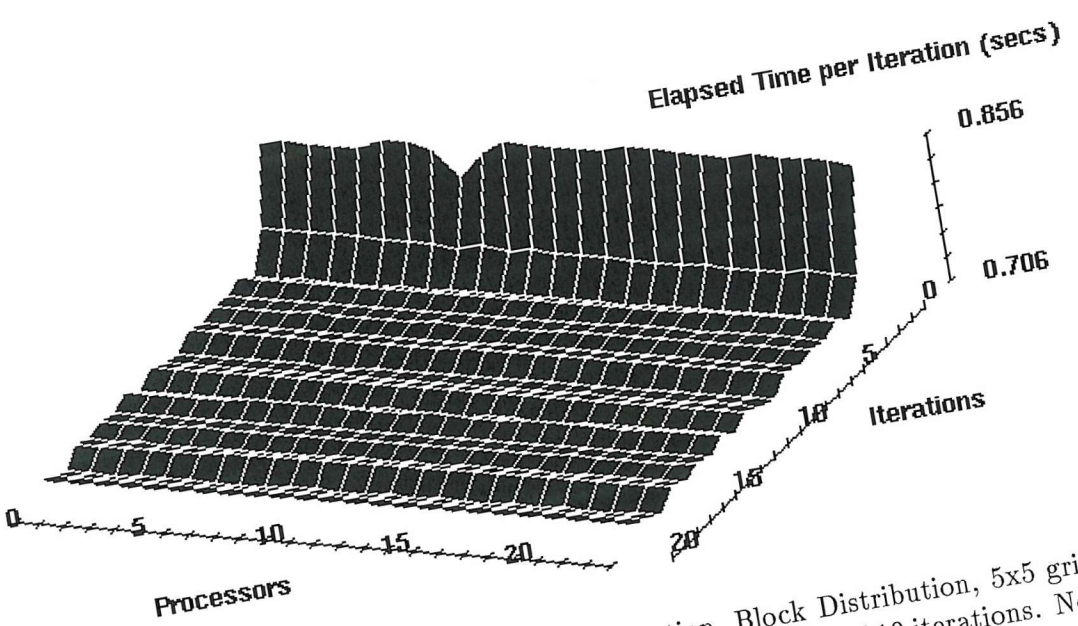

Figure 5.17: Do-Loop-Surface: Red Black Relaxation, Block Distribution, 5x5 grid of processors. In this picture, each iteration correspond to a *block* of 10 iterations. Notice that iteration 20 is representing iteration 200.

program performance visualization. VISTA is applicable to a large class of scalable programs and machines, specifically SPMD and data-parallel programs executing on distributed memory computer systems, and it is presented via a hierarchy of views. At least two levels of observation can be identified in VISTA: microscopic (lower- level) and macroscopic (higher-level). A microscopic perspective highlights the individual components of the system, and specific behaviour is presented in detail. A macroscopic view reflects the overall behaviour of the collection of components. The activity of any component becomes part of an aggregate activity of the system. These levels form a useful hierarchy which must be exploited for larger systems. A key and essential area of study is analyzing the microscopic versus macroscopic performance of large systems, where there is little experience [157].

Not only visual but also aural methods are being explored in order to represent parallel performance data. Visual and aural portrayals of parallel program execution are used to gain insight into how a program is working (e.g. PABLO [22]). The combination of portrayals in a coordinated performance environment provides the user with multiple perspectives and stimuli to comprehend complex, multidimensional run-time information [103]. Reasons for using auralization in general and in parallel computing in particular are documented by Francioni [104, 105]. Some interesting features of auralization are:

- By listening to the auralization while looking at a related graphical display, the speed of recognition and distinction of whole and partial programs may be increased over using either sound or graphics alone.
- The scalability of auralization, specifically for representing program behaviour, is essentially an unknown and remains to be empirically determined.

In general, visual-aural representations could be used effectively on large systems. However, suitable approaches must support a hierarchical presentation and/or logical grouping of information [103].

Another interesting line of research is proposed by Sarukkai et al [12]. They suggest that a programmer should begin the investigation into the causes of performance degradation with high level and abstract views, so that global trends can be seen. A set of guidelines for methodical application of views, proceeding from the highest level to the lowest level, is also presented in [12]. While a collection of views, animations, and general performance data can help uncover performance bugs, some guidelines or strategies are needed to direct the order in which views are examined. The guidelines that they suggest are:

- Dissatisfied with performance?
- Compare progress with *ideal* behaviour.
- Examine deviation in overall behaviour.
- Examine individual sections/PEs.
- Low Level investigation.

The work done by Couch in [158] deals with the problem of scalability. In that paper, global context is described by scalable execution views that do not change in format, size, meaning, or clarity as processors are added to an execution. Couch also states that one way to produce a scalable view is to categorize processors by behaviour and display category statistics. Categorical views are particularly useful when there is an inverse mapping for an arbitrary view region to the subset of processors whose behaviour was described in the region. The execution visualization tool Seeplex implements this form of category management to provide scalable execution views [158]. Seeplex manages view relationships using a data-flow visualization environment in the spirit of scientific visualization systems such as AVS. Two important issues are also addressed by Couch. The first one is that a scalable view is less useful if one cannot trace back from view features to raw data. The second one is that "*..visualization takes much time, effort, and experience to provide results. We hope to be made obsolete by an automatic analysis method that avoids the global search problem and all its difficulties. However, we do not think this likely*" [158].

Finally, another interesting research effort is the approach to parallel program analysis by LeBlanc, Mellor-Crummey and Fowler [125], which is based on a multiplicity of views of an execution. A synchronization trace, captured during execution, is used to construct a graph representation of the behaviour of the program. The user then manipulates this representation to create and tune visualizations using an integrated, programmable toolkit. They also state that tools should be structured so that a programmer can select views of a program execution in some reasonable sequence, where each view takes into consideration previous views and the programmer's current needs. One of the views presented in this work done by LeBlanc, Mellor-Crummey and Fowler, analyzes a similar surface (equivalent to a Do-Loop-Surface) displaying data transfer time, worker processes and rounds of computation. For a Gaussian Elimination experiment, they concluded that the display helped them to find an explanation of a particular performance problem. Although there are similarities between this visualization and the DLS display, we believe that the DLS display has several new features such as the integration with a Scientific Data Visualization Tool (AVS), providing further data analysis capabilities. Additionally, the idea of using this sort of display for performance analysis is evaluated in depth in this document.

6 DLS Tool Evaluation: Case Studies

The purpose of this Chapter is to illustrate and validate the usefulness of the DLS displays in analyzing parallel program performance. A set of case studies is presented including a Matrix Multiply, TRANS1 and FFT1 from the Genesis Benchmarks Suite, a Red/black relaxation algorithm, and a synthetic code to analyze cache/memory behaviour. Finally, we describe our experience using DLS displays at Argonne National Laboratory.

6.1 Matrix Multiply

The purpose of this case study is to demonstrate the usefulness of a DLS in analyzing the performance of a parallel algorithm. The parallel implementation of this Matrix Multiply algorithm works in the following way: the matrix A is distributed by rows and the matrix B by columns. At the stage i, the processor which has the row i send this row to all the other processors (broadcast) and then each processor starts computing the corresponding inner products in order to generate the result (each processor generates a set of elements of the resulting matrix).

A trace generation library was written in order to collect the required data per iteration. The only thing that the user has to do in order to obtain trace data is to add a call to the corresponding trace function at the beginning and at the end of the do-loop. If the number of iterations is very large, the trace can be generated only for a set of iterations (e.g. each 100 iterations) in order to reduce the amount of information produced. The hardware platform consisted of a 64 T800 Parsys Supernode and the program was written in C using PARMACS [43].

6.1.1 Results for a 100x100 matrix on 32 processors

The DLS of figure 6.2 represents the matrix multiplication algorithm for two matrices of dimension 100x100 (we could not use a larger problem due to lack of memory), running on 32 processors (initialization and terminating values have been removed in order to get a clearer picture). The values correspond to the main do-loop of the algorithm as it is illustrated in figure 6.1.

The DLS in figure 6.2, shows a clear pattern which is repeated three times. This pattern consist of a hilly section that goes through a diagonal line until the pattern is repeated again. Each time the pattern begins, there is a *mountain* which is higher from left to right. All these hilly sections represent communication delays, and they basically occur when a processor is broadcasting one particular row to the other processors. Since the processors are numbered from 0 to 31 from left to right in the picture, then processor 0 is the one which takes longer in the iteration where it has to broadcast a row, due to the fact that while it is sending that particular row to every other processor, those processors start doing their computation but processor 0 has to wait until it finish the broadcast in order to start doing its corresponding inner products (since the

```
 for (i=0;i<gnrowA;i++) {

  DLSGETT(1,i); /* Collection of data */

  if (isInHere(i)) {
    broadcast_async(MYPROC,matrixA[i/NUMPROC],ncolA*sizeof(float),MSG);
    memcpy(tmp_row,matrixA[i/NUMPROC],ncolA*sizeof(float));
  }
  else {
    RECV(tmp_row,ncolA*sizeof(float),&l,&s,&t,MATCH_TYPE(MSG));
  }
  for (j=0;j<ncolB;j++) {
    elem = 0;
    for (k=0;k<ncolA;k++) {
      elem = elem + tmp_row[k]*matrixB[j][k];
    }
    partialR[j].i = i;
    partialR[j].j = (j*NUMPROC) + MYPROC - 1;
    partialR[j].element = elem;
  }
  if (gncolB%NUMPROC < MYPROC)
    partialR[gncolB/NUMPROC].i = -1;
  /* Elements of row i ready */
  SENDR(HOSTID,partialR,((gncolB/NUMPROC)+1)*sizeof(RMATRIX),MSG_R);

  DLSGETT(1,i); /* Collection of data */

}
```

Figure 6.1: Main do-Loop of the Matrix Multiply Algorithm (node program).

communications are synchronous). For a matrix of size 100x100 and 32 processors, this occurs three times because most of the processors have 3 columns (100/32 = 3.12) and only a few of them have up to 4.

Now, one good question is whether we can improve the performance of the program in some way using the information of the DLS. One possible answer is trying to reduce those hilly parts of the surface (i.e. reducing the delays per processor). Since the communications are synchronous in this implementation of the algorithm, one possibility is to use asynchronous communication. The effect on the DLS when using asynchronous communication is remarkable (figure 6.3). In this display the hills are lower and the valleys deeper, which means lower values for the elapsed time per iteration. Although the initialization costs are the same, the improvement in performance is notable (from 6.48 seconds to 2.15 seconds).

In terms of speedup, in figure 6.4 we can see the improvement gained by using asynchronous communication when a particular processor is broadcasting one of the rows that belongs to it. This improvement in speedup increases as the problem size increases. For n=100 (n=number of columns), the maximum speedup occurs at p=8 (p=number of processors) and for n=400 at p=32. For n=400 and synchronous communication, the speedup starts decreasing after p=16 while the speedup for the asynchronous version is still increasing up to p=32.

6.2 Genesis Benchmarks

FFT1 (Fast Fourier Transform) and TRANS1 (Matrix Transpose) are two programs from the Genesis Benchmarks Suite [142]. TRANS1 transposes a square matrix by dividing the original matrix in sub-matrices and exchanging them between *opposite* processors (see figure 6.5).

However, the distance between these processors is not the same (64 T800 Parsys Supernode) and in this 4x4 grid example the exchange of information between processor 3 and 12 becomes a bottleneck. If one adds an additional link between processors 3 and 12 (in order to make this distance shorter, figure 6.6) then one may expect a significantly better result.

The effect of adding one extra link to the grid topology makes an impressive impact on the DLS display. This additional link creates a direct communication line between the two most distant points of the grid, making communication much faster. The average elapsed time per iteration is reduced from 2.4 seconds to 1.8 seconds (see figures 6.9 and 6.10). In this example, the term *iteration* involves the execution of the whole matrix transposition (figure 6.7).

The FFT1 benchmark is based on the classical radix-2 1-dimensional FFT algorithm by Cooley and Tukey. The Fourier transform comprises two main phases: the generation of twiddle factors and the calculation of harmonic amplitudes (butterfly phase). The latter is commonly critical for the speed of computation. Figure 6.8 shows the section of the algorithm we are measuring.

In this experiment, the results of running the FFT1 benchmark on a 4x4 grid of processors, topology I, and on a 4x4 grid plus some additional links, topology II, (figure 6.11) are shown in figures 6.12 and 6.13.

In these figures, both DLS's are also compared against the Feynman display of ParaGraph. For topology I, synchronization points seem to be not as good as for

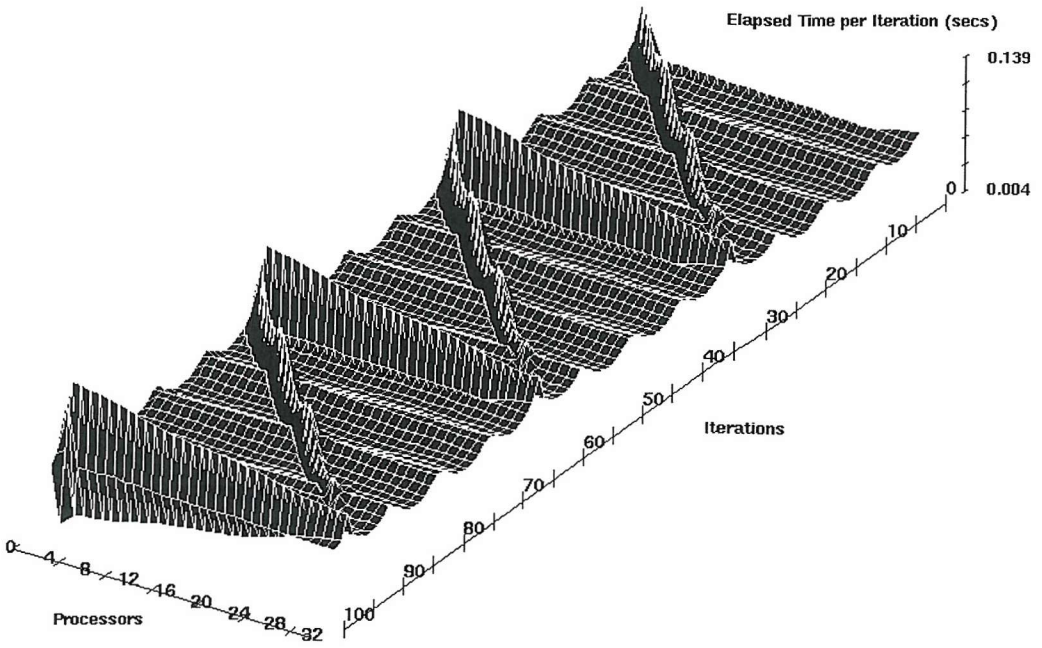

Figure 6.2: Matrix Multiply: Do-Loop-Surface (Synchronous communication). Notice the pattern that is repeated three times representing communication delays while a particular processor is broadcasting a row of the matrix to every other processor.

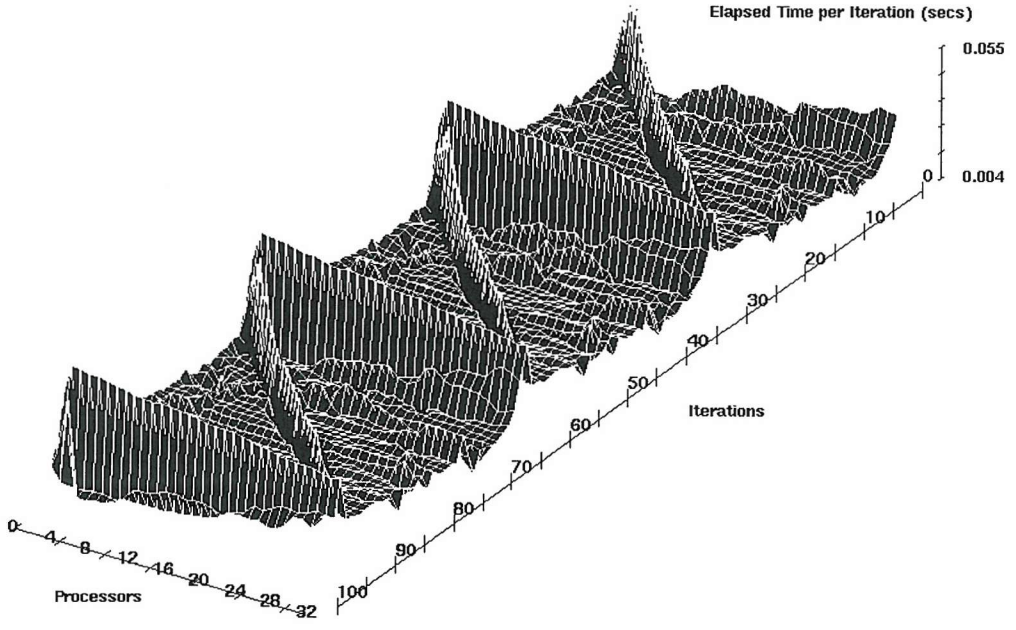

Figure 6.3: Matrix Multiply: Do-Loop-Surface (Asynchronous communication). Notice how the values are smaller, deeper valleys and a more intense blue in a colour picture (red=high values, blue=small values), than the previous figure.

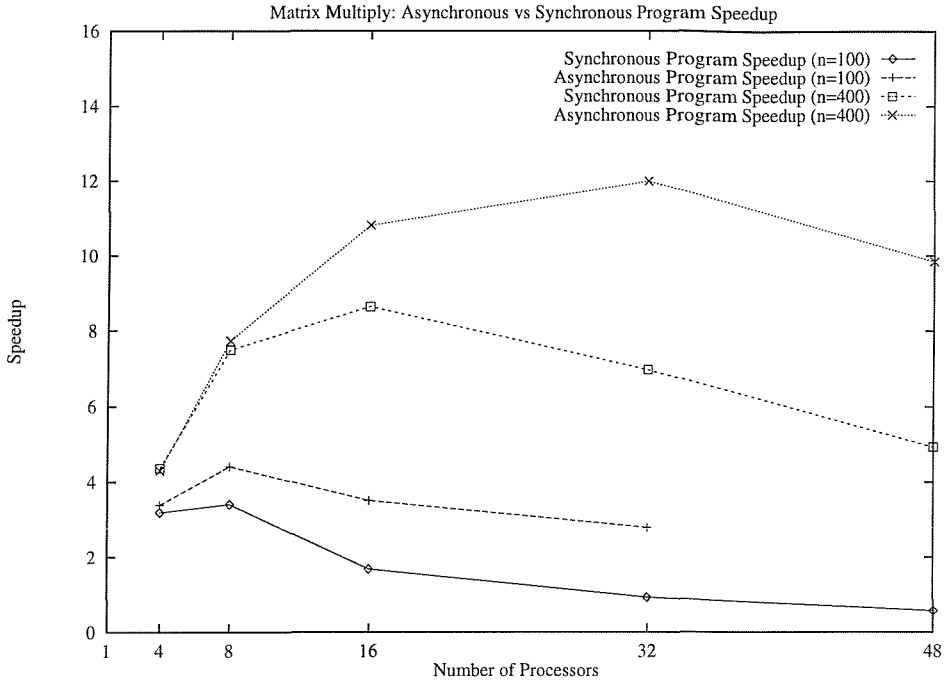

Figure 6.4: Matrix Multiply: Program Speedup for Synchronous and Asynchronous communications (n=100 and n=400).

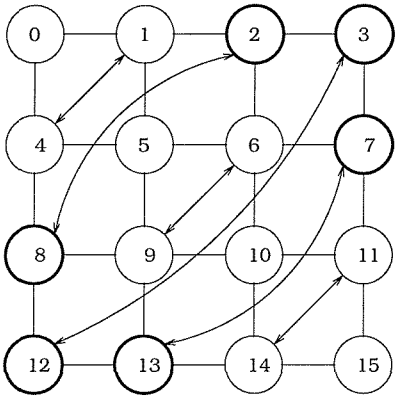

Figure 6.5: TRANS1 Genesis Benchmark. Exchanging of sub-matrices between *opposite* processors. Note that the distance between these processors is not the same. **Bold** circles indicate longer distances between two processors.

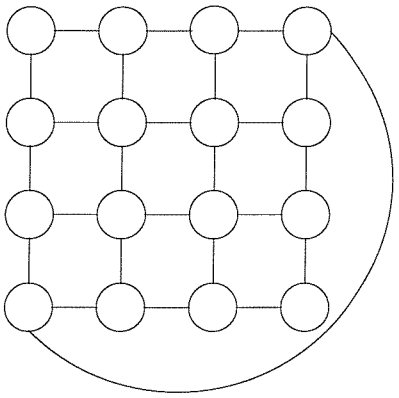

Figure 6.6: TRANS1 Genesis Benchmark. 4x4 Grid with additional link between processors 3 and 12.

```
      DO 100 COUNT=1,LOOPS

C     DLS_TRACE
       CALL DLSGET(1,COUNT)
C     DLS_TRACE

       DO 50 I = 1,D
          DO 40 J = 1,D
            TEMP(I,J) = DATA(J,I)
40        CONTINUE
50     CONTINUE

       IF (X.NE.Y) THEN
         SEND(TASKS(OPPOS),TEMP,DATLEN,10+COUNT)
         RECV(DATA,DATLEN,IL,IS,IT,MATCH_ID_AND_TYPE(TASKS(OPPOS),10+COUNT))
       ELSE
         DO 120 I=1,D
            DO 110 J=1,D
              DATA(I,J) = TEMP(I,J)
110         CONTINUE
120      CONTINUE
       ENDIF

C     DLS_TRACE
       CALL DLSGET(1,COUNT)
C     DLS_TRACE

100   CONTINUE
```

Figure 6.7: TRANS1 Genesis Benchmark. *Iterations* (number of times the algorithm is executed) being measured during the experiment.

```
C   ===           Butterfly phase - Fourier analysis              ===

      DO 200 LAYER = LOGP,1,-1

C     DLS_TRACE
      CALL DLSGET(1,LAYER)
C     DLS_TRACE

C   ===                   Calculation sublayer                    ===

      CALL FFTCT1(FINTR,FINTI,OMEGAR(INDTF),OMEGAI(INDTF),NTOTIN)
      INDTF = NTASK/(2**(LAYER-1))

C   ===                   Communication sublayer                  ===

      CALL DESTOM (LAYER,NTASK,DESTIN,OFFSET)
      IF(OFFSET .NE. 0) THEN
         DO 101 I = 0, NTOTIN/2-1
            BUFFL(I) = FINTR(I)
101      CONTINUE
         SEND(PROCN(DESTIN),BUFFL,BUFLEN,30+LAYER)
         RECV(BUFFL,BUFLEN,IL,IS,IT,MATCH_ID_AND_TYPE(PROCN(DESTIN),30+LAYER))
         DO 201 I = 0, NTOTIN/2-1
            FINTR(I) = BUFFL(I)
201      CONTINUE
      ELSE
         DO 301 I = 0, NTOTIN/2-1
            BUFFR(NTOTIN/2+I) = FINTI(NTOTIN/2+I)
301      CONTINUE
         SEND(PROCN(DESTIN),BUFFR,BUFLEN,30+LAYER)
         RECV(BUFFR,BUFLEN,IL,IS,IT,MATCH_ID_AND_TYPE(PROCN(DESTIN),30+LAYER))
         DO 401 I = 0, NTOTIN/2-1
            FINTI(NTOTIN/2+I) = BUFFR(NTOTIN/2+I)
401      CONTINUE
      ENDIF

C     DLS_TRACE
      CALL DLSGT(1,LAYER)
C     DLS_TRACE

200   CONTINUE
```

Figure 6.8: FFT1 Genesis Benchmark. *Iterations* being measured during the experiment.

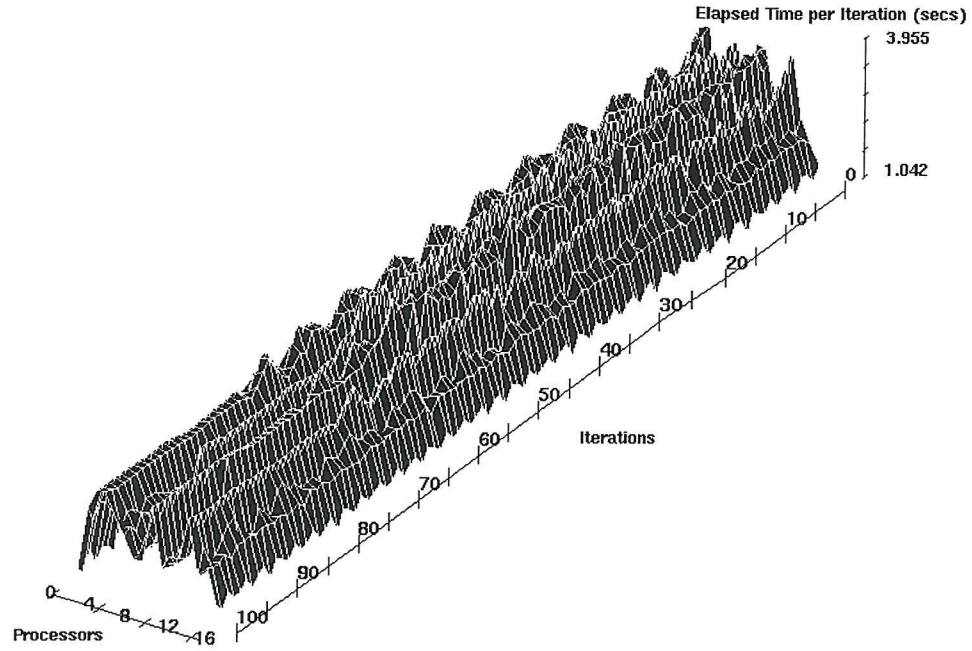

Figure 6.9: TRANS1 Genesis Benchmark. DLS display of a 100x100 matrix example on a 4x4 grid of processors (PARSYS Supernode, 16 processors).

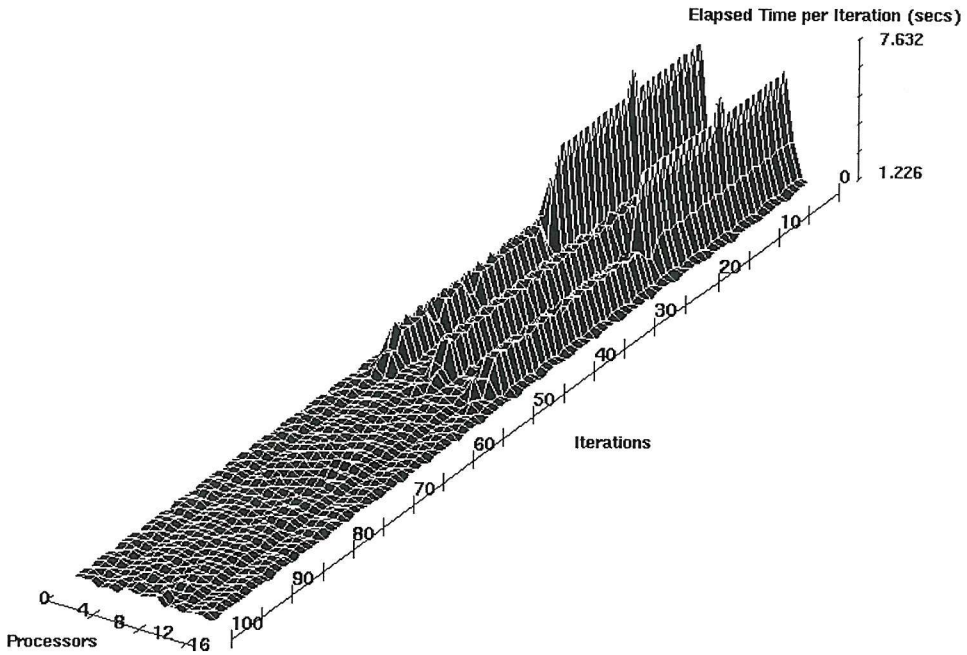

Figure 6.10: TRANS1 Genesis Benchmark. DLS display of a 100x100 matrix example using one additional link between the two more distant points of the grid (PARSYS Supernode, 16 processors).

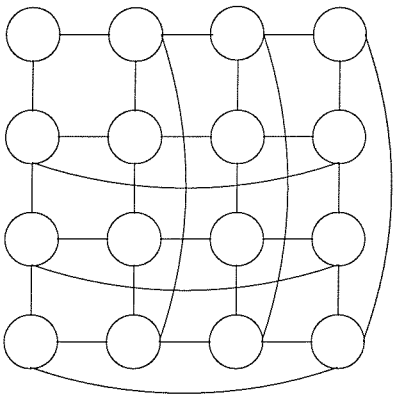

Figure 6.11: Grid of 4x4 processors plus some additional links.

topology II. The first and the third iterations have longer elapsed time than iterations 2 and 4. This fact can be seen in the DLS display (figure 6.12).

For topology II, every processor starts and ends communication at the same time. Some processors can send messages "faster" than others due to additional links, and then they keep waiting for the corresponding receive (orange lines on the Feynman display in a colour picture). The first three iterations have the same elapsed time, but the last one is shorter. This last remark can be clearly seen in the DLS display (figure 6.13). Even a small change in the elapsed time of a particular iteration may produce a dramatic effect on the DLS display as it is illustrated in this example.

6.3 Cache/Memory effects

The main purpose of this case study is to show how useful the DLS displays can be to understand cache/memory effects generated by changing the order in which a particular do-loop accesses a particular data structure (a matrix in this case). The hardware platform for this experiment is a CM-5 (64 Sparc nodes with 32 Mbytes of memory and 4 vector units each). The software platform is based on the PARMACS message-passing interface.

The do-loop under study can be seen in figure 6.14. Notice that each do-loop can be arbitrarily reordered, changing therefore the access to each matrix A, B, and C. The operation executed in the innermost do-loop is irrelevant, since the most important issue for this experiment is the way the memory hierarchy is used.

In figures 6.15, 6.16, and 6.17 the memory access pattern is displayed depending on the order the do-loop is executed (i.e. there are only 6 possible combinations for i,j,k).

Figures 6.18 to 6.23 show the results for this case study. Each figure represents one particular order of the 6 possible ones. The memory size, which is another important factor to be taking into account, is defined by the size of each matrix in the example. In this case, each matrix is 50x50x50.

The results in terms of performance, from best to worst, are:

- k,j,i; 0.465 seconds in average per iter, per processor (figure 6.23).
- k,i,j; 0.466 seconds in average per iter, per processor (figure 6.22).
- j,k,i; 0.481 seconds in average per iter, per processor (figure 6.21).
- i,k,j; 0.7 seconds in average per iter, per processor (figure 6.19).
- j,i,k; 1.004 seconds in average per iter, per processor (figure 6.20).
- i,j,k; 1.176 seconds in average per iter, per processor (figure 6.18).

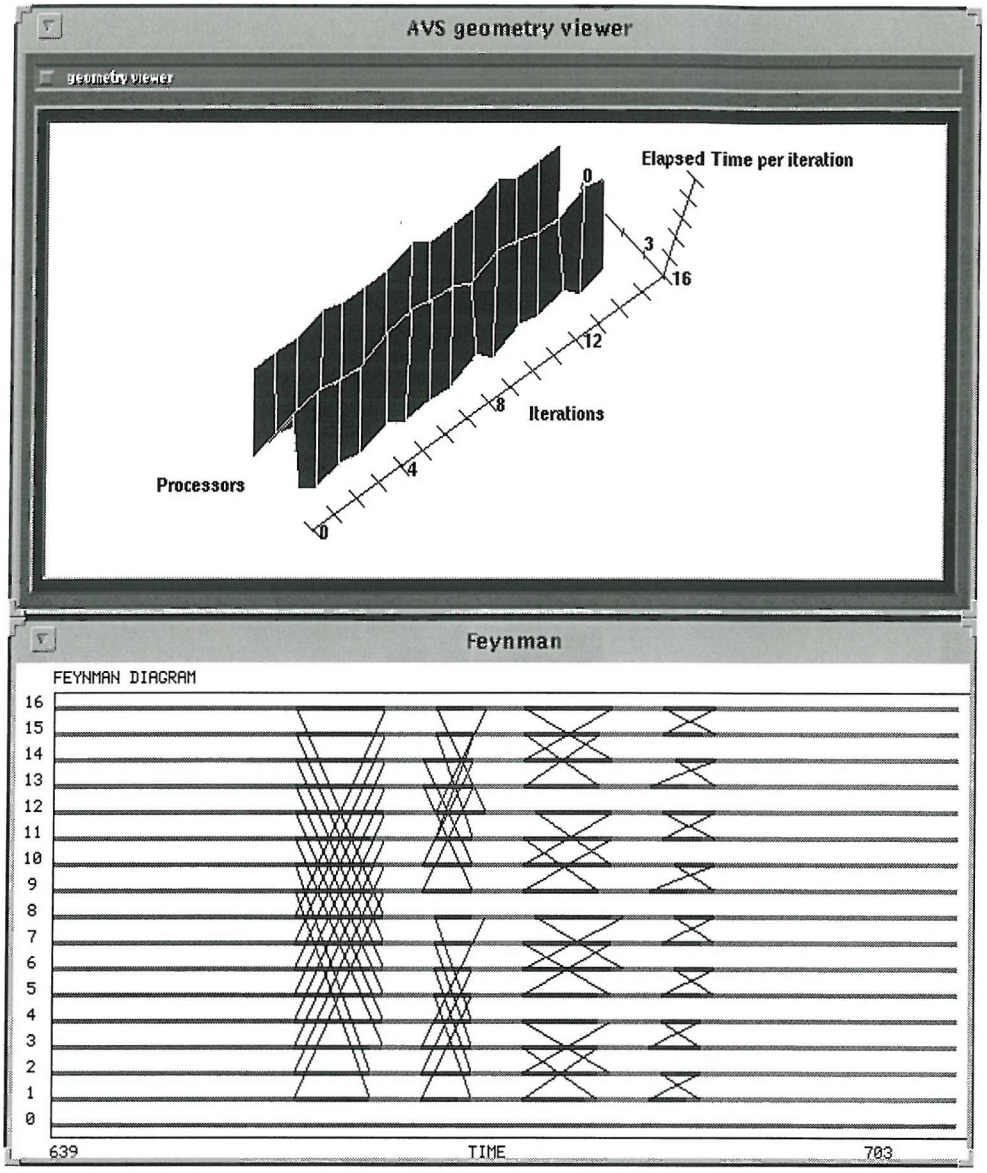

Figure 6.12: FFT1 Genesis Benchmark. Comparison between a Feynman display (ParaGraph) and a DLS. 4x4 Grid.

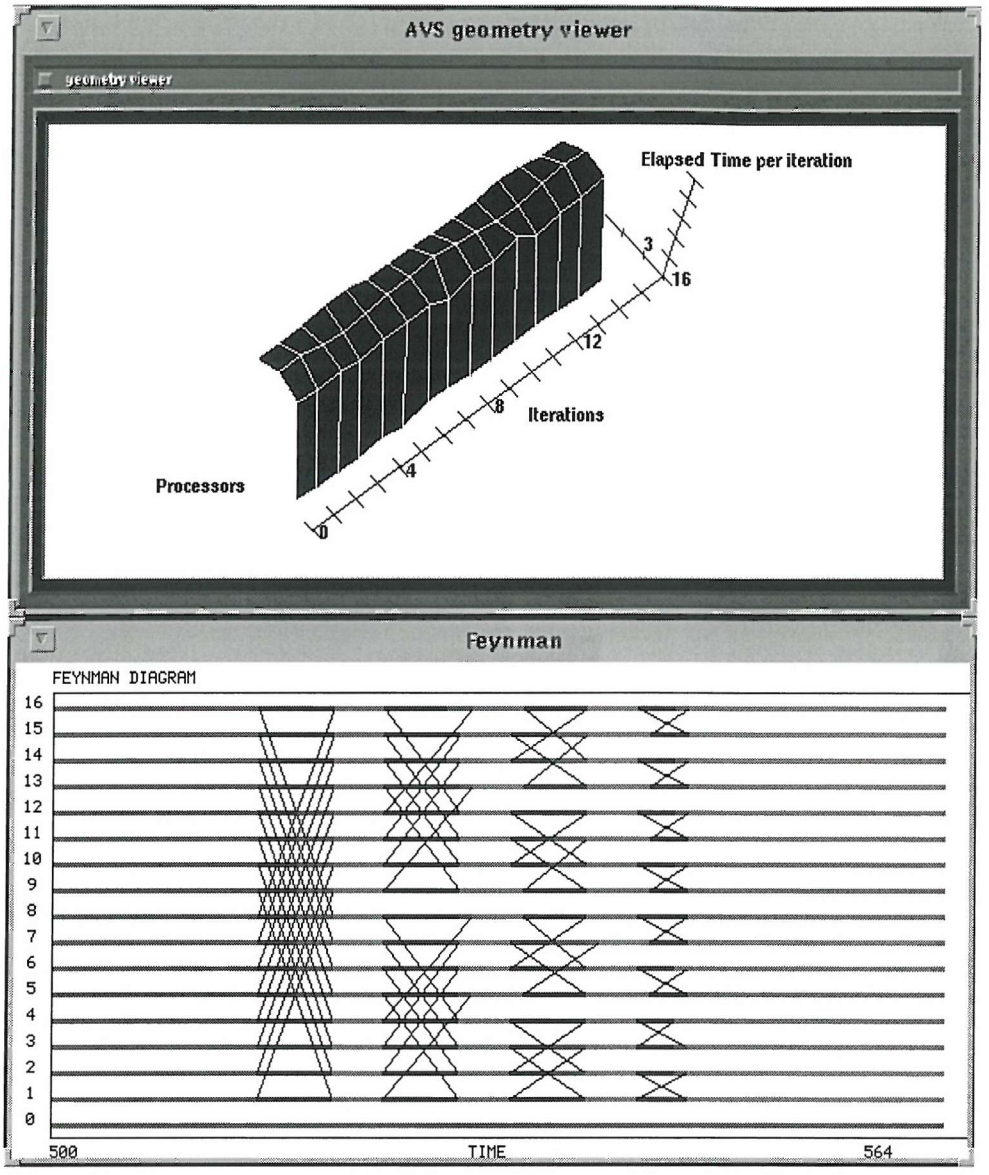

Figure 6.13: FFT1 Genesis Benchmark. Comparison between a Feynman display (ParaGraph) and a DLS. 4x4 Grid plus some additional links.

```
      do iter=1,50
C DLS_TRACE
        call dlsgett(1,iter)
C DLS_TRACE
        do i=1,maxlen
          do j=1,maxlen
            do k=1,maxlen
              matrixC(i,j,k)=(matrixA(i,j,k)*matrixB(i,j,k))+matrixA(i,j,k)
            enddo
          enddo
        enddo
C DLS_TRACE
        call dlsgett(1,iter)
C DLS_TRACE
      enddo
```

Figure 6.14: Case study: nested do-loops. Notice that each do-loop can be arbitrarily reordered, changing therefore the access to each matrix A, B, and C.

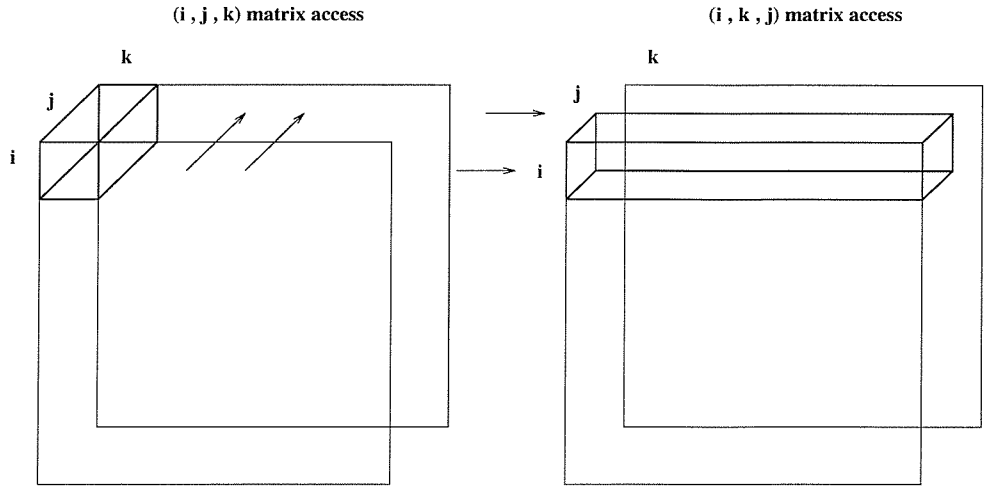

Figure 6.15: Memory access patterns for a 3D matrix. Access by rows.

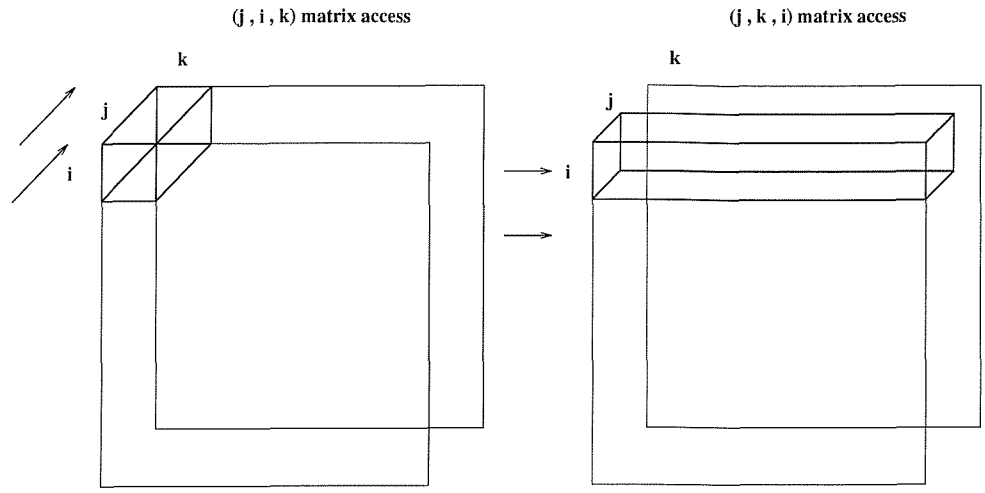

Figure 6.16: Memory access patterns for a 3D matrix. Access by columns (left) and by rows (right).

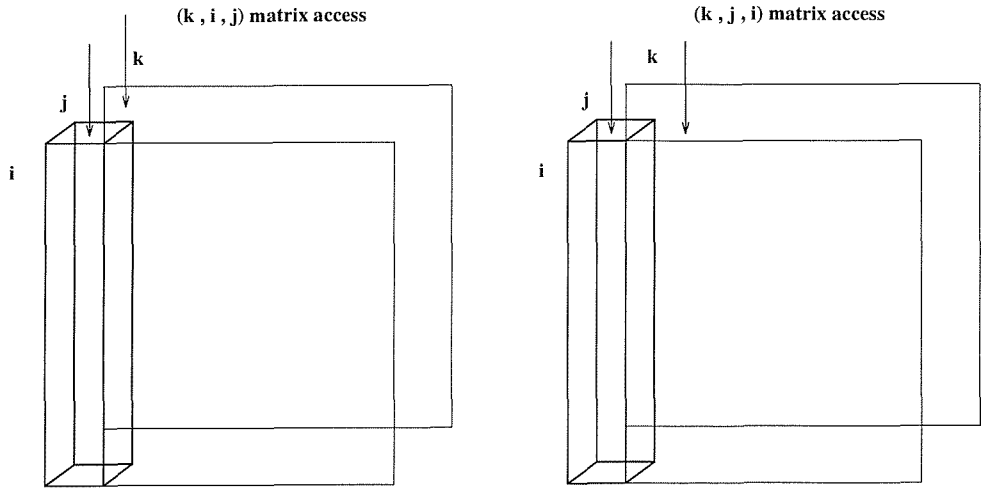

Figure 6.17: Memory access patterns for a 3D matrix. Access by columns.

It is clear that the best results are obtained when the matrices are accessed by columns, since this is the way FORTRAN stores data. The shape of the DLS is smooth for the better cases (figures 6.23 and 6.22) and it becomes hilly and irregular for the worse cases (figures 6.18 and 6.20), where the matrices are being accessed in a less efficient way. In the best case (figure 6.23), it is possible to see how the execution time of each iteration varies (increasing or decreasing) due to cache/memory effects, repeating this behaviour in a regular fashion.

One interesting variation of the experiment occurs when the size of the matrices is increased. In this example, the matrices are 75x75x75 (figures 6.24 and 6.25). The cases shown in the figures are the best (k,j,i) and the worst (i,j,k). Notice the dramatic change in the DLS display for the worst case. After a few iterations, there is a jump in the figure and the elapsed time per iteration increases in about 7 milliseconds. This situation might be produced by the increasing of the problem size, since a bigger problem would fill in the cache completely, generating therefore more misses. It is also interesting to see the following effect: the memory has been increased by a factor of 3.38 approximately and the average elapsed time per iteration has been increased by a factor of 3.93 (worst case). In general, the execution time increases faster when the size of the problem is increased.

The following example shows the result of a experiment of size 75x75x75 on 15 processors using the compiler optimization option (-O) on the CM-5 in time-sharing mode (figure 6.26), for the best matrix access pattern (k,j,i). This figure shows a very interesting behaviour: the first few processors (the first 7) execute the nested do-loops in less time (about 30 milliseconds less) than the remainder 8 processors. Inquiring about this situation with one of the system programmers of the CM-5, we found that due to a problem with the 602 floating point unit in the SPARC-2 chip set, some of the CM-5 nodes run more slowly than others when running serial f77/C codes. It is important to notice that we discovered this problem without having any previous knowledge about this strange behaviour of the floating point unit in the SPARC-2 chip, and with the help of the DLS display of figure 6.26.

6.4 Red Black Relaxation

Red/black is an example of a "small" real application. If we consider a rectangular data domain, red/black is a way of separating the grid into two halves: a red set and a black one with the additional constraint that a red cell is only surrounded by black cells in its non diagonal directions and viceversa. One solves the linear equation system first on the red cell subset by using the (old) values of the black cells and then computes the solution on the black cells by using the (new) values of the red cells. These steps are repeated until the desired result is obtained.

In this section, we show two examples of cache/memory effects for this application on a Meiko CS-2. The current machine configuration has two partitions each of four processors, a *vector* partition and a *scalar* partition. Each processor in the vector partition has a 66MHz HyperSparc CPU with a pair of Fujitsu uVP vector units. The processors in the scalar partition are 50MHz SuperSparc CPU's. The software platform is based on the PVM message passing interface.

The original idea of this experiment was to compare different strategies to calculate the computational part of the red/black algorithm. These strategies are *loop unrolling* and *loop merge* and they compute the same results as the original do-loop that makes the

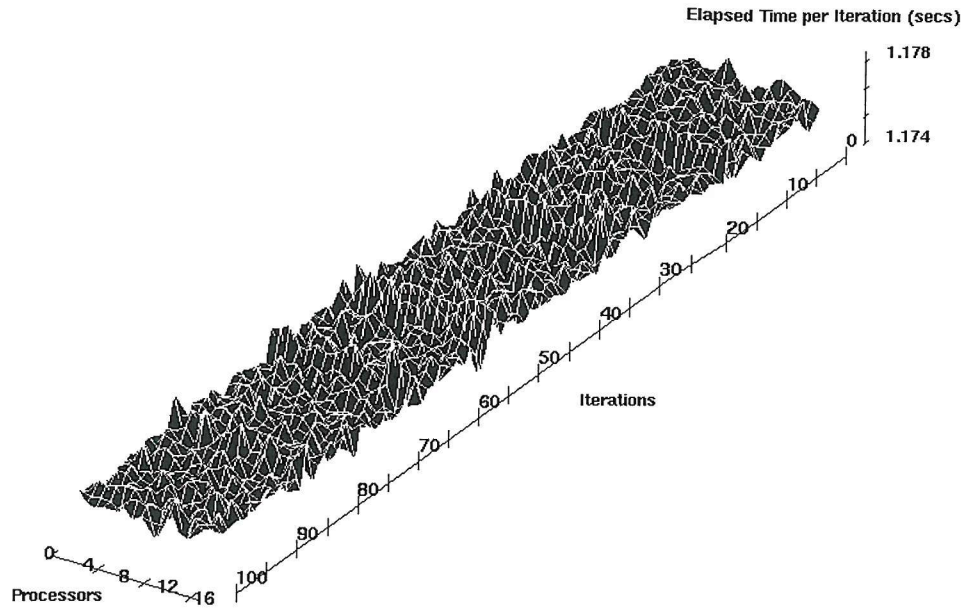

Figure 6.18: DLS: Matrix access using i,j,k. Average time per iteration: 1.176 seconds (CM-5, 31 processors).

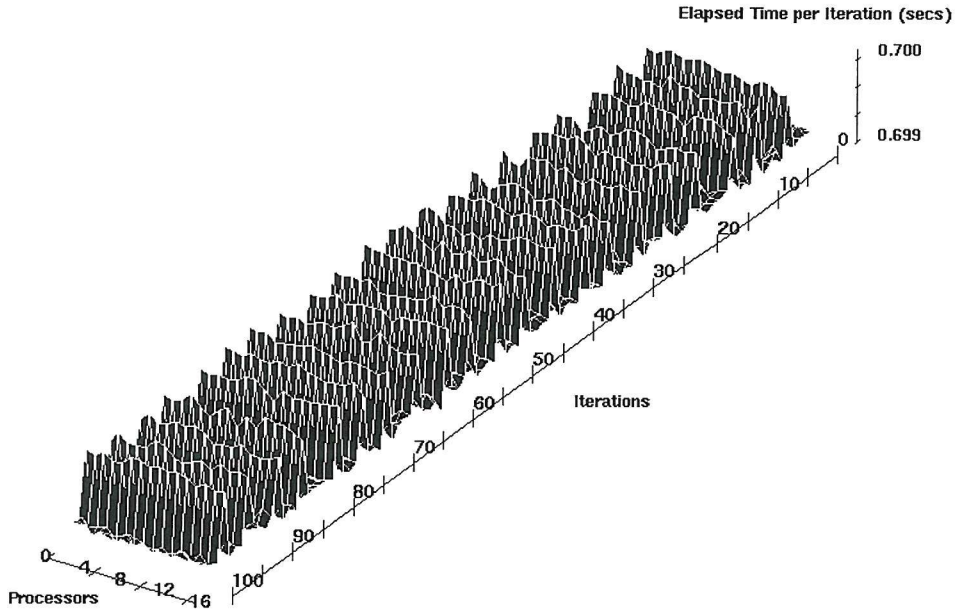

Figure 6.19: DLS: Matrix access using i,k,j. Average time per iteration: 0.7 seconds (CM-5, 31 processors).

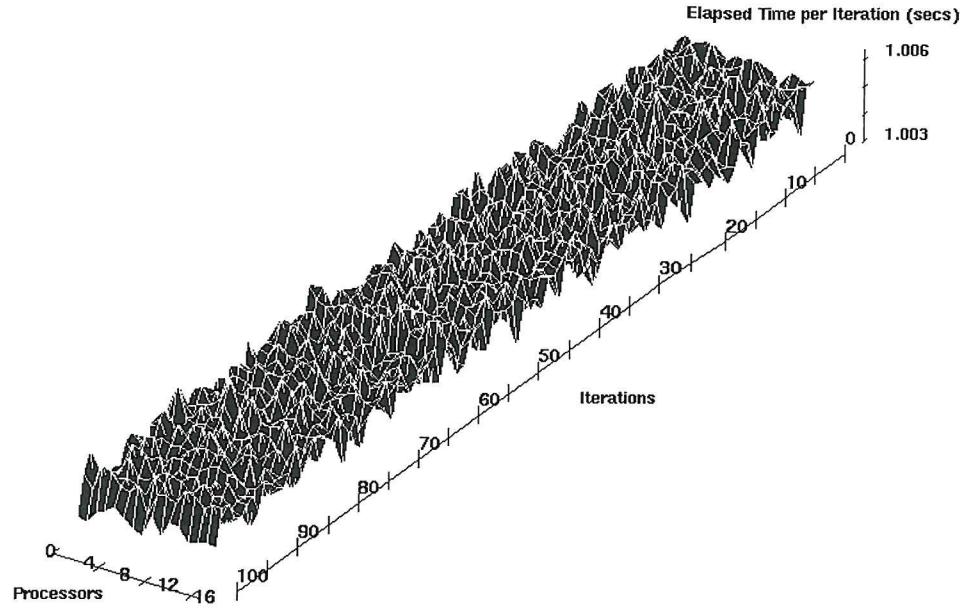

Figure 6.20: DLS: Matrix access using j,i,k. Average time per iteration: 1.004 seconds (CM-5, 31 processors).

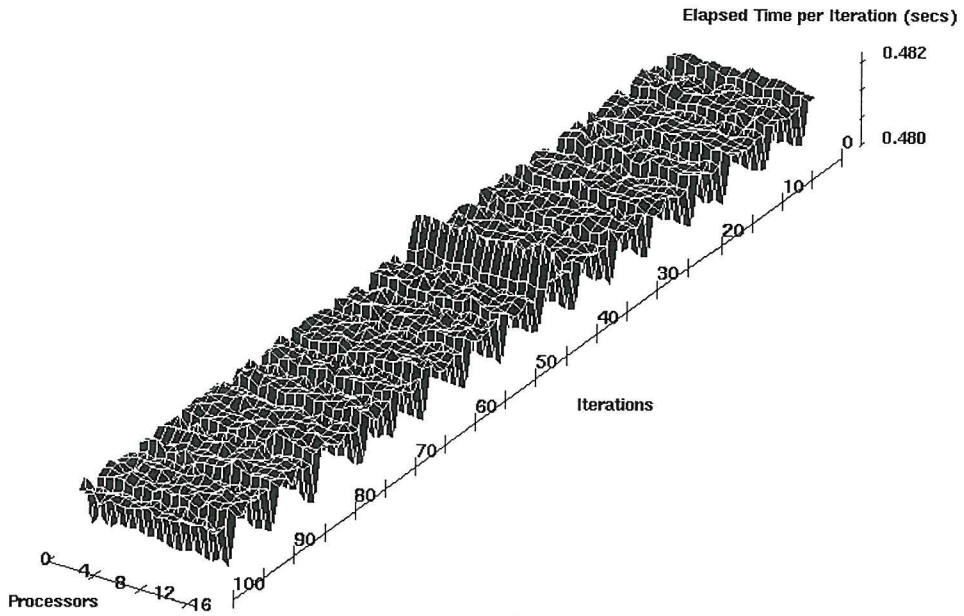

Figure 6.21: DLS: Matrix access using j,k,i. Average time per iteration: 0.481 seconds (CM-5, 31 processors).

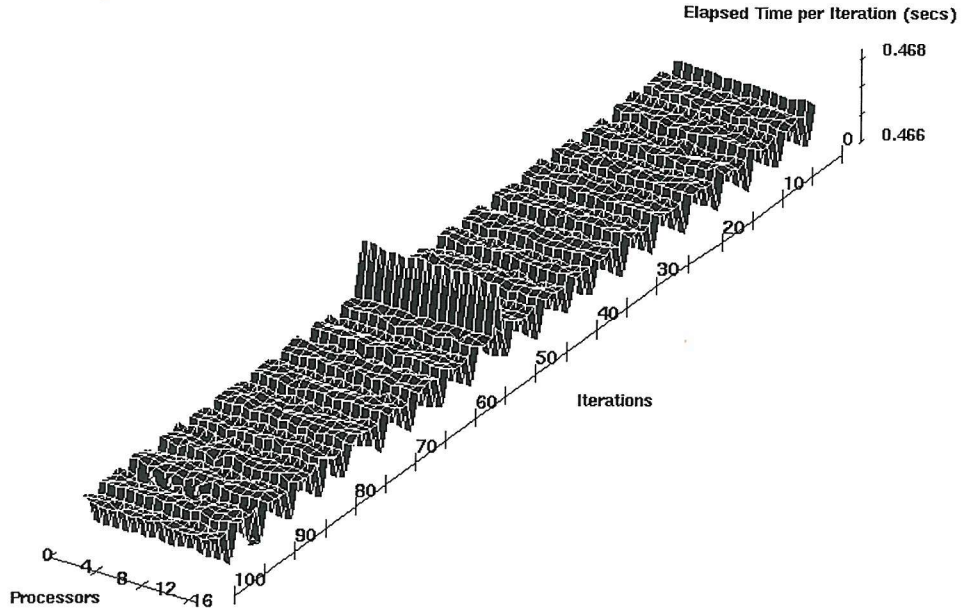

Figure 6.22: DLS: Matrix access using k,i,j. Average time per iteration: 0.466 seconds (CM-5, 31 processors).

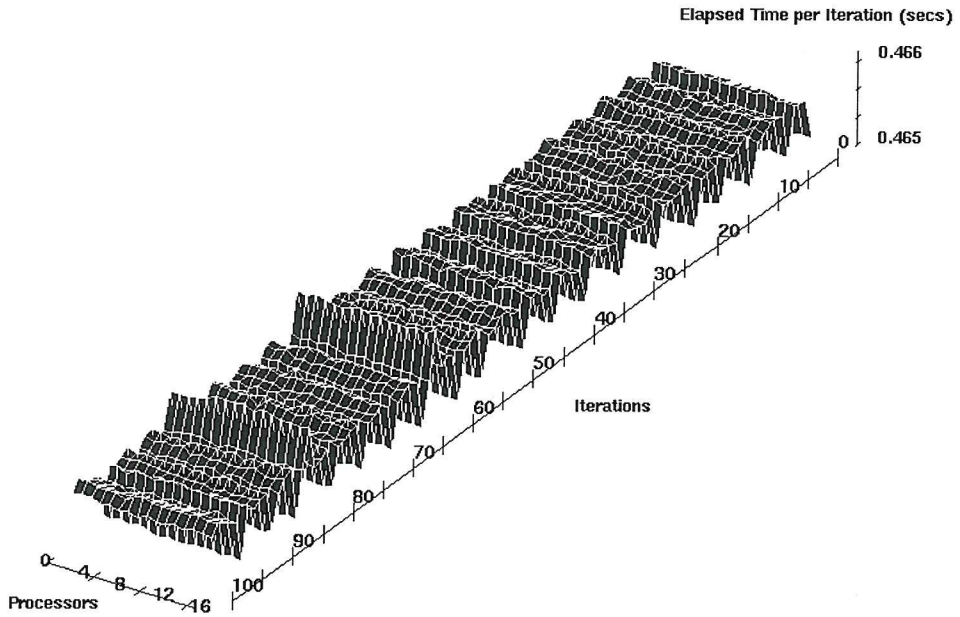

Figure 6.23: DLS: Matrix access using k,j,i. Average time per iteration: 0.465 seconds (CM-5, 31 processors).

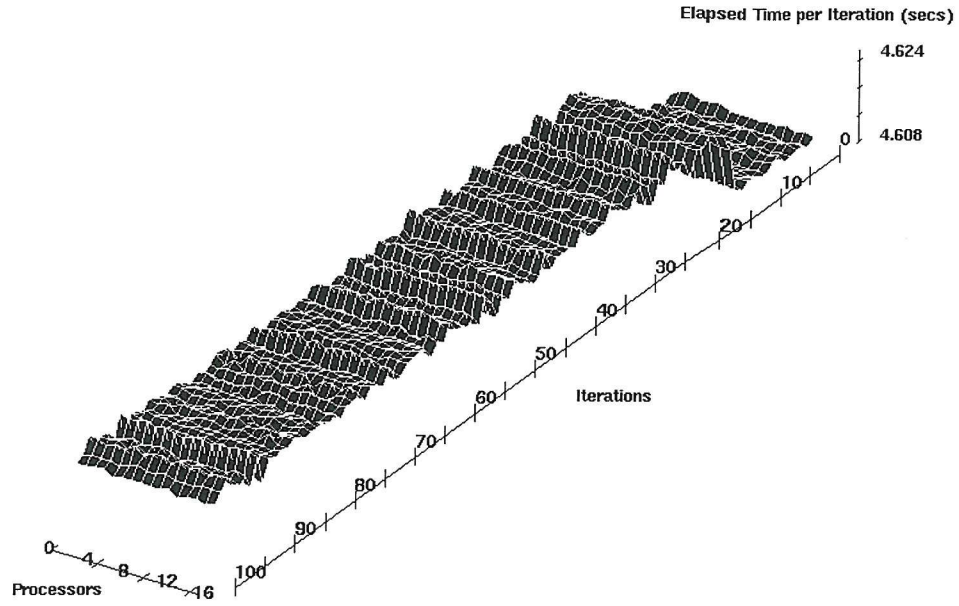

Figure 6.24: DLS: Matrix access using i,j,k. Average time per iteration: 4.619 seconds. Matrix size: 75x75x75. Size increased by a factor of 3.38 (CM-5, 31 processors).

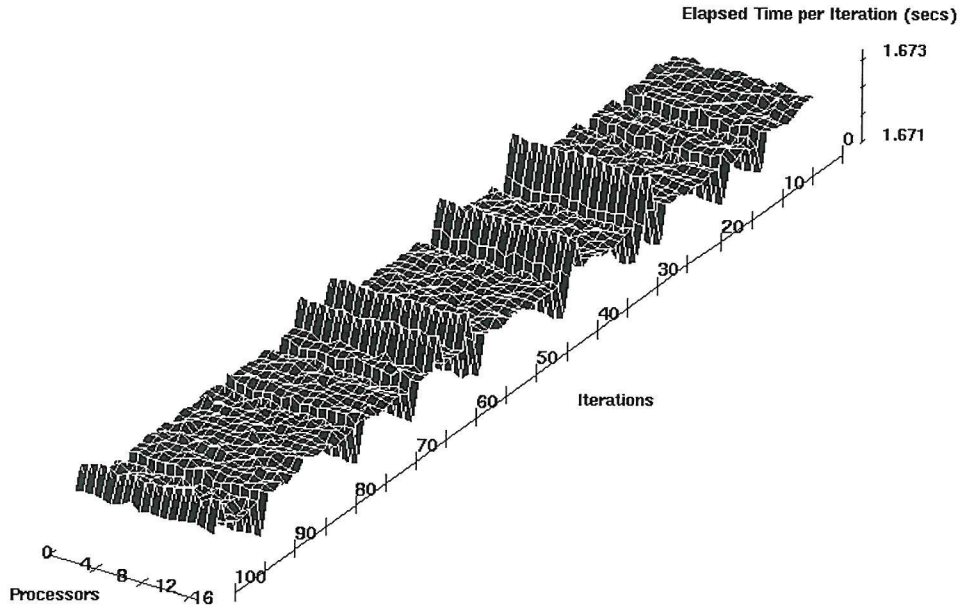

Figure 6.25: DLS: Matrix access using k,j,i. Average time per iteration: 1.672 seconds. Matrix size: 75x75x75. Size increased by a factor of 3.38 (CM-5, 31 processors).

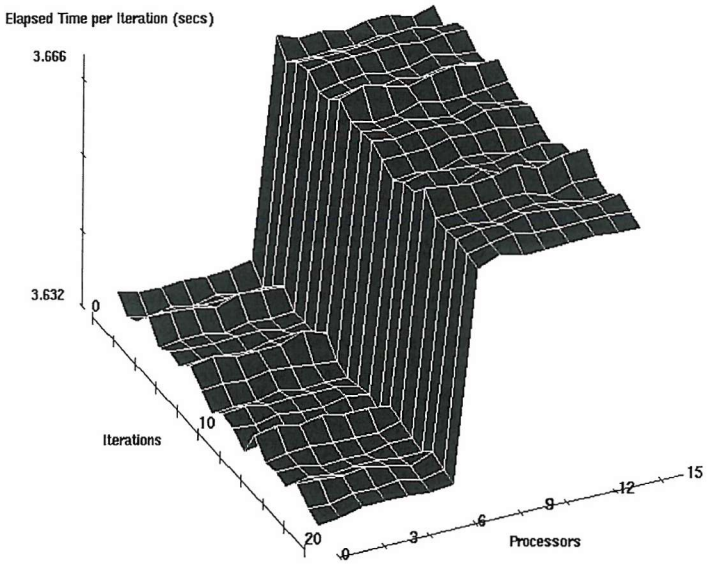

Figure 6.26: DLS: Matrix access using i,j,k. Average time per iteration: 3.649 seconds. Matrix size: 75x75x75. CM-5 in time-sharing mode. The compiler optimization option (-O) has been used in this example. Notice that there is a difference in execution time between the first 7 processors and the last 8 (CM-5, 15 processors).

computation in the red/black algorithm, but they operate with the data in a different way (i.e. different memory access patterns). However, we did not find a considerable difference in applying any particular strategy (the merge loop was slightly better), but we found an irregular behaviour. The very first iteration of the program was taking almost 100% longer than the following iterations. In order to test whether this situation was due to cache/memory effects, we prepared an example where a copy of the original 2D data array used in the algorithm was incorporated in the computation but only on two specific iterations (5th and 17th). Since this new array has not been referenced before, we would expect a longer execution time for these two iterations. The DLS of figure 6.27 illustrates a high initialization cost in one of the partitions (the scalar) as well as higher execution time in the 5th iteration (as we would expect) but only for the scalar partition. The surface of the DLS for the vector partition (first 4 processors from left to right in the figure) was flat and regular.

The problem is generated by the size of the data arrays. When this size is large enough (greater than 1 Mb for the scalar partition), we find a long initialization phase and subsequent irregular memory behaviour (hills in the picture). But, if the size of the data array is less than 1 Mb, the initialization costs disappear as well as the irregular memory effects and we get a flat surface again. The size of the off chip direct mapped cache for the scalar processors is precisely 1 Mb. We confirmed this analysis using a smaller data array. We obtained the expected result for the experiment using a data array which has not been referenced before in two particular iterations (4th and 16th), as illustrated in figure 6.28.

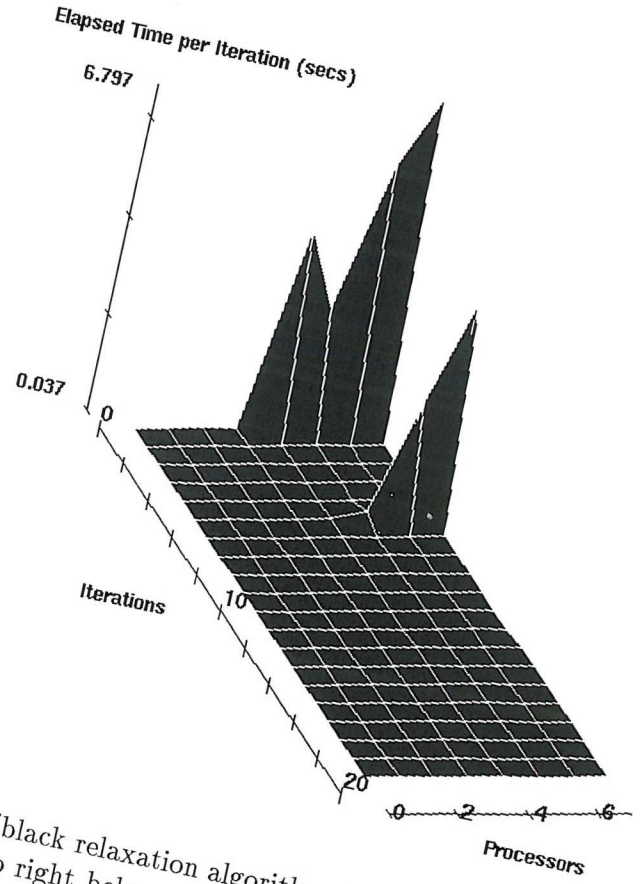

Figure 6.27: DLS: Red/black relaxation algorithm, Meiko CS-2, 8 processors. The first 4 processors from left to right belong to the *vector* partition and the remainding 4 to the *scalar* partition.

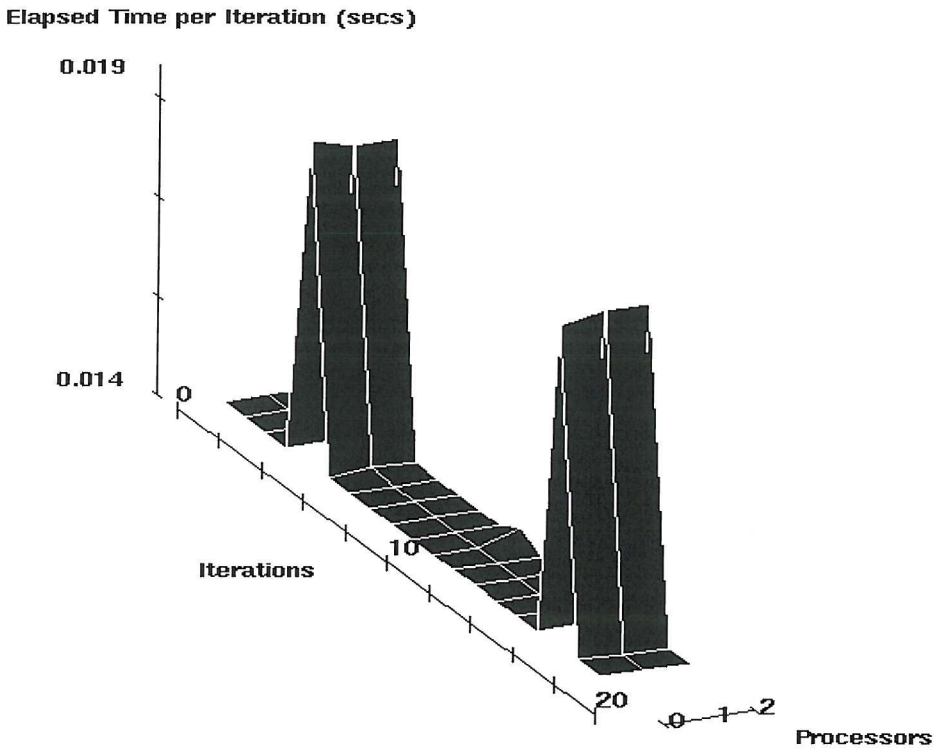

Figure 6.28: DLS: Red/black relaxation algorithm, Meiko CS-2, 3 processors, *vector* partition. Notice that the 4th and 16th iterations have longer execution time due to the usage of an unreferenced data array on those two iterations. We have eliminated the first iteration in order to obtain a clearer picture.

6.5 Experiments at Argonne National Laboratory

This section describes our experience during a study visit to Argonne National Laboratory (ANL) in Illinois, USA (March 1995). One of the main purposes of this visit was to test the scalability of the DLS displays for a large number of processors. In fact, we could run applications up to 128 processors on an IBM SP2. Also, we describe in this section how the DLS trace generation library was ported to MPI, making the generation of trace files portable. Finally, we discuss several experiments done during this visit as well as the interaction between DLS and Nupshot, a performance visualization software developed at Argonne.

6.5.1 DLS version for MPI

The DLS displays do not depend on any particular hardware platform or programming style. The only requirements in order to generate a DLS display are a DLS trace file and AVS. The DLS trace file has been generated in the past by writing a small trace library for the corresponding message-passing system (i.e. PARMACS and PVM). However, for MPI it was easier. The MPI version of Argonne comes with a separate environment called MPE (Multi-Processing Environment), which is a software provided by ANL to enhance message-passing programming with MPI, including graphics and profiling capabilities [159]. MPE has a set of profiling routines that are used to create log-files of events that occur during the execution of a parallel program. By using this library, one can measure the elapsed time for each iteration on a particular do-loop in a program, which is the information required to display a DLS (figure 6.29 illustrates how to use MPE to generate the information of a DLS trace file).

MPE is currently used to generate log-files for Nupshot, a performance visualization software developed at ANL. Nupshot is an enhanced version of Upshot, also developed at ANL. One of the main advantages of using MPE is that this library can be used with other vendor versions of MPI (since it is based on calls to MPI). In this way, we would be able to generate DLS trace files on any of the platforms supported by MPI. Also, MPE allow us to analyze several do-loops at the same time, making it possible to generate several DLS's on a single run. The only requirement to display this information as a DLS, is to transform the MPE log-file into the corresponding DLS trace file format required by AVS. For all the advantages that this approach give us, we think that a future freely available version of DLS will be based on MPI only.

6.5.2 The LINPACK Benchmark in MPI

In order to test the DLS version for MPI and the scalability of the DLS displays on the IBM SP2, we made several experiments. The first one consists of an MPI version of the LINPACK software [160, 159].

The LINPACK benchmark is a well known numerical code that is often used to benchmark computers for floating-point performance. This program solves the linear system $Ax = b$, where A is a *dense* matrix (i.e. A[i,j] != 0). The problem is solved by computing a factorization of A into PLU, where L is a lower triangular matrix, U an upper triangular matrix, and P is a permutation matrix that represents the exchange of rows used in the *partial pivoting* algorithm (strategy used by this code to improve numerical stability). The matrix A is distributed amongst the processors using the square block scattered decomposition described in [161], which is a practical and general

```
   .
   .
c
c Definition of the DLS state that we are going to measure. A state
c consist of two events. In our example, these events are 1 and 2
c

   if (myid .eq. 0) then
     CALL MPE_DESCRIBE_STATE(1, 2, "DLSijk", "blue:gray")
   endif
c
c Start generating tracing information
c
   CALL MPE_START_LOG()
   do iter=1,100
c
c First event: beginning of the do-loop
c
     CALL MPE_LOG_EVENT(1, iter, "DLSijk-start")
     do i=1,max_iter
       do j=1,max_iter
do k=1,max_iter
  c(i,j) = a(i,k)*b(k,j) + c(i,j)
        enddo
       enddo
     enddo
c
c Second event: end of the do-loop
c
     CALL MPE_LOG_EVENT(2, iter, "DLSijk-end")
   enddo

c
c Stop generation tracing information and save it in the file dls.log
c
   CALL MPE_STOP_LOG()
   CALL MPE_FINISH_LOG_("dls.log")
   .
   .
```

Figure 6.29: Generation of DLS trace data using MPE

purpose way of decomposing dense linear algebra computations. In problems, such as LU factorization, in which rows and/or columns become inactive as the algorithm progresses, this decomposition provides a good load balance (as it is shown in [161]).

This algorithm was run on an IBM SP2 at ANL[1]. This machine consists of 128 nodes and two compile servers. Each node is essentially an RS/6000 model 370. This model has a 62.5 MHz clock, a 32Kb date cache, and a 32Kb instruction cache. The main features of this system are [162]:

- 128Mb of memory per node.
- 1Gb local disk on each node (400Mb for users and the rest for paging and for the operating system).
- Full Unix on each node (IBM AIX 3.2.5).
- Each node is accessible by Ethernet from the Internet.
- High performance Omega switch for the interconnection network between processors. The SP2 interconnection network is a bidirectional multistage interconnection network. The SP2 method of packet transfer between nodes is related to *wormhole routing* [163]. Latency is 63 micro seconds and bandwidth is 35Mb/s.
- The maximum number of processors for this machine is 128, but 512 systems are available on special request.

Figure 6.30 shows the execution of the main do-loop of the LU decomposition (subroutine PDLUBR) for 100 processors (with N=6400 and NB=64, where N is the size of matrix and NB the block size used for wrapping). The figure illustrates a clear pattern of *blocks* repeated every 10 iterations and processors. This is due to the fact that the topology used by the algorithm is a grid of 10x10 processes. This is exactly the way the algorithm is designed to work [164]. If you assume that the processors are arranged as a 10x10 grid (10 rows and 10 columns), and the grid is composed of rows and columns, then the factorization of the block is performed by processor columns in a cyclic manner. During the very first iteration processors 0, 10, 20,..., 90 will be involved in the factorization. After factorization, the factored panel (or factored section of the matrix) has to be broadcast to the rest of the process columns. In this broadcast, processors {0, 1,..., 9}, {10, 11,..., 19},..., {90,..., 99} are involved (i.e. 10 broadcasts occur in parallel). After this step, there is an update operation on the block and this update is done in a processor row, and therefore only processors {0, 1,..., 9} are involved. After this, the updated block is broadcast to other process rows, so processors {0, 10, 20,..., 90}, {1, 11,..., 91},..., {9, 19,..., 99} are involved in the broadcast (all occur in parallel again). Then the above step is repeated starting with processor 1, that is in the processor column 1.

We can also see how every 10 processors, there is a *drop* in the execution for that particular iteration (creating the diagonal lines of the figure). All these patterns, correspond to the cyclic behaviour of the algorithm. Also, notice how the surface *decreases* as the iterations progress (due to the fact that more rows and columns become

[1]The author(s) gratefully acknowledge use of the Argonne High Performance Computing Research Facility. The HPCRF is funded principally by the U.S. Department of Energy Office of Scientific Computing.

inactive as the algorithm move forward). The cyclic patterns can also be seen using Nupshot (figure 6.31, for 9 processors). However, for 100 processors, it is more difficult to use this sort of display (at least on a single view, see figure 6.32).

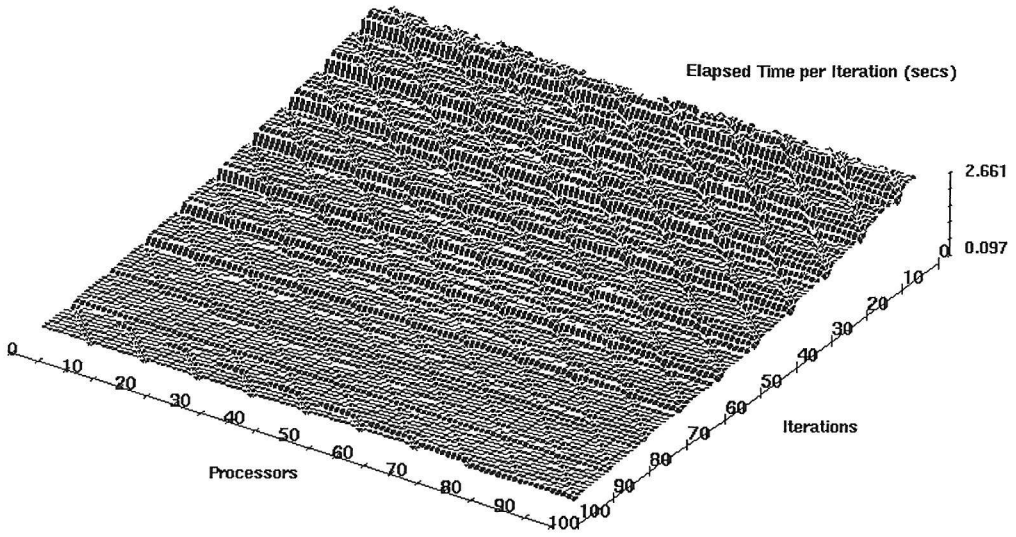

Figure 6.30: DLS display for LINPACK (IBM SP2, 100 processors). Notice the *cyclic* patterns every 10 iterations/processors. It is also important to say that in this single picture we are displaying the information of 100 iterations for 100 processors.

6.5.3 Comparison between ANL-MPI and IBM-MPI

Another interesting experiment was to compare the execution of the two current versions of MPI available for the IBM SP2 at ANL (i.e. IBM-MPI and ANL-MPI). The results can be seen in figures 6.33 and 6.34. It is clear from the pictures that both executions are *different*. In a colour picture, we would notice first that the values for the IBM-MPI case are higher than for the ANL-MPI case. Also, the *regularity* of the patterns has been affected. This fact can be deduced by the sequence of peaks present in the first 30 iterations of the algorithm. Nupshot was used in order to confirm this behaviour, and the results are illustrated in figures 6.31 and 6.35. The ANL version of MPI proved to be more efficient than the IBM version for this particular application.

6.5.4 Scalability test using 128 processors

Figures 6.36 and 6.37 illustrate one of our scalability tests using the full configuration of the IBM SP2 at ANL (128 processors). Also, this test allow us to see a very interesting behaviour that we have seen in similar cases before but on different machines (e.g. CM-5): some processors are taking longer (more than 20 percent in some cases) to execute the exact same code. The do-loop we are analyzing consist of statements that do not include any communication at all and they are repeated on every processor. We know that there are a few set of processors that run slower than others (i.e. processors 1, 17, 33, 49, 65, 81, 97, and 113), but we have found that for this particular test some other processors behave different than expected. As in our case study in Section 5.3.1 (figure 5.16), it seems that either there is also an additional overhead of the operating

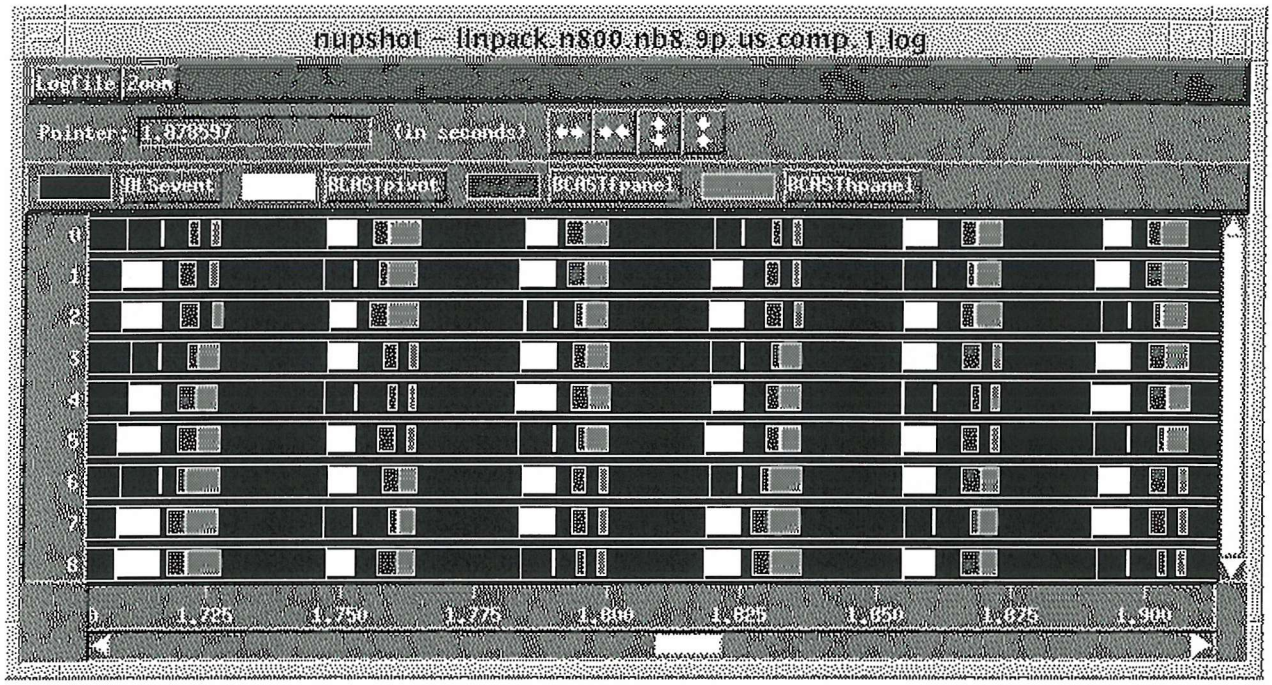

Figure 6.31: Nupshot: section of the LINPACK execution (IBM SP2, 9 processors). Notice the *cyclic* pattern in communications: first 3 processors, then processors 3 to 5, then processors 6 to 8, and so on. Communications correspond to the three broadcasts inside the main do-loop of the LU factorization of LINPACK.

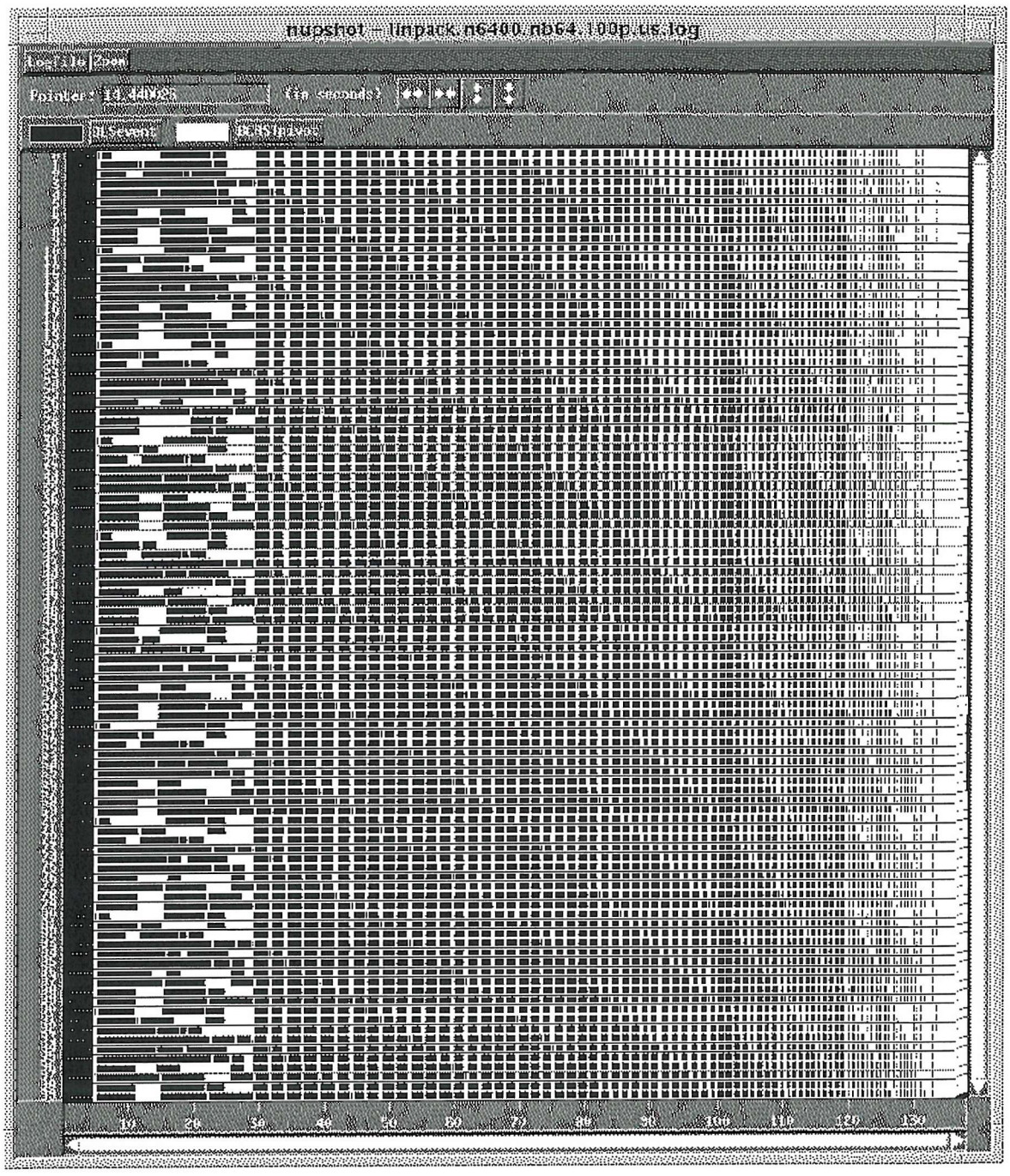

Figure 6.32: Nupshot: section of the LINPACK execution (IBM SP2, 100 processors). The view becomes too complex for a single picture.

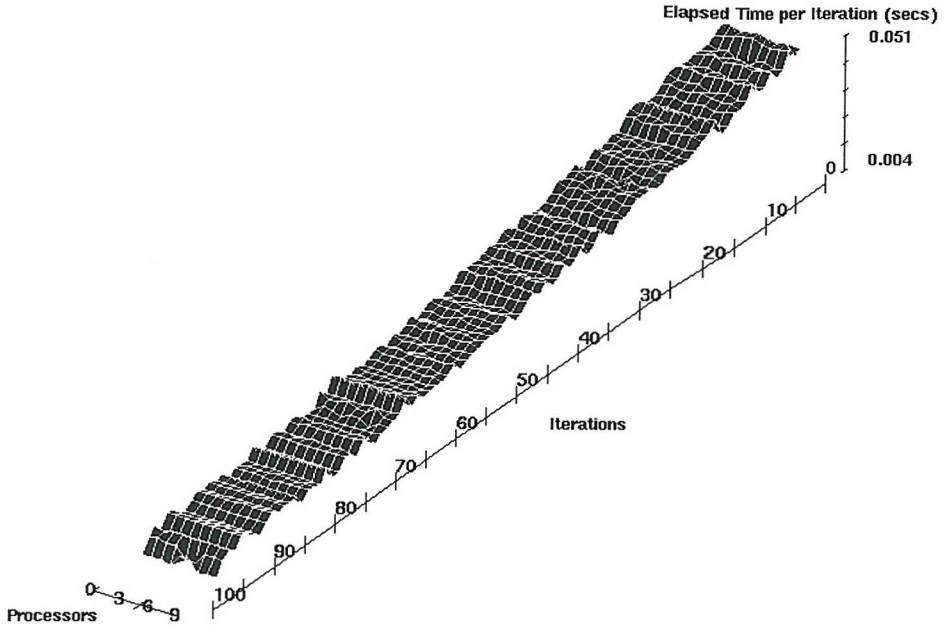

Figure 6.33: DLS: Execution of LINPACK using the ANL version of MPI for the IBM SP2 (mpich).

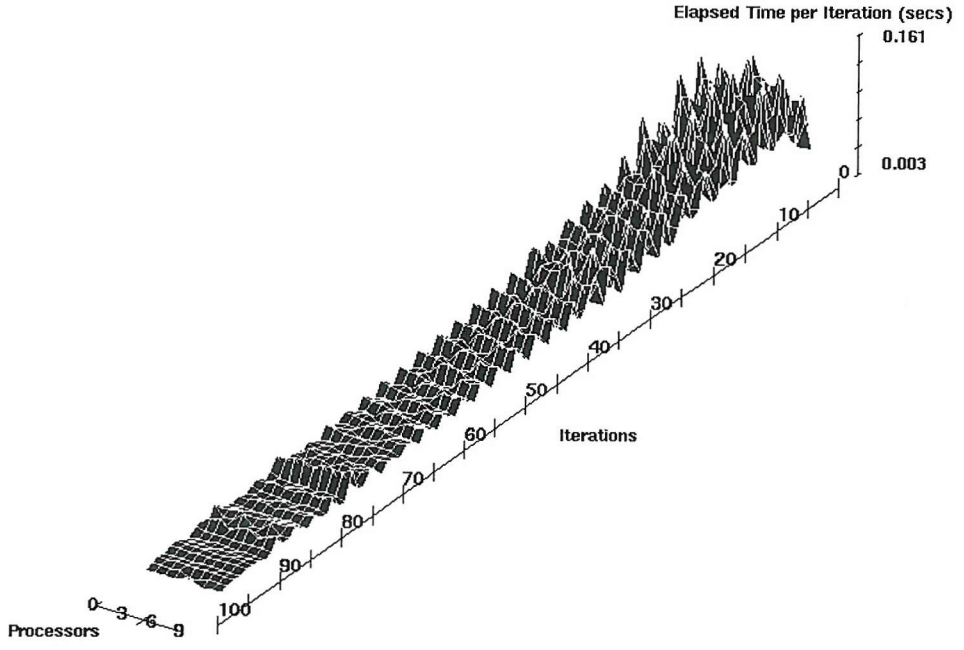

Figure 6.34: DLS: Execution of LINPACK using the IBM version of MPI for the IBM SP2. Notice the differences in terms of *regularity* that both pictures have. The only thing in common is the fact that the values decreases as the execution progress (which is a feature of the algorithm in general).

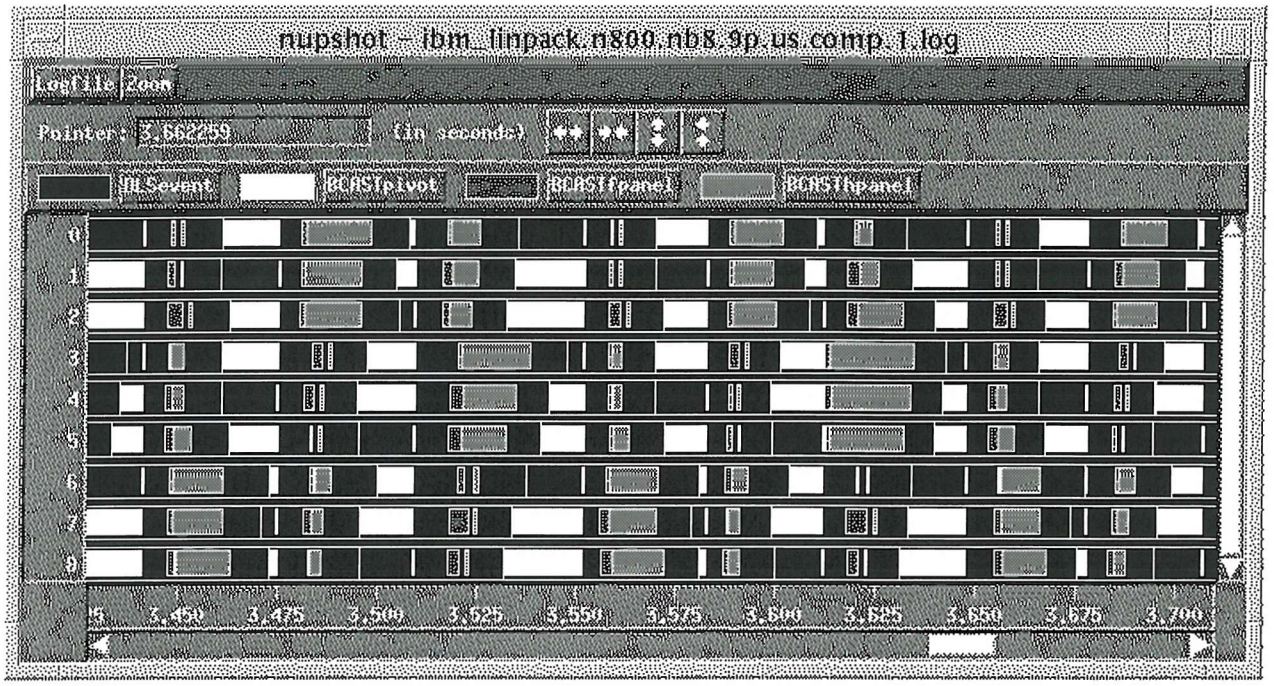

Figure 6.35: Nupshot: Section of LINPACK execution for the ANL version of MPI (IBM SP2, 9 processors). Notice the *irregularity* in the communication patterns.

system that affects the execution of our code or these set of processors are slightly *different* than the others. In terms of scalability, the picture easily represents 128 processors and 100 do-loop iterations. We believe that the DLS displays will scale in the same way as AVS scales. However, for thousands of processors, the trace files will be perhaps too large (especially if the number of do-loop iterations is large too) and the DLS could be more difficult to handle.

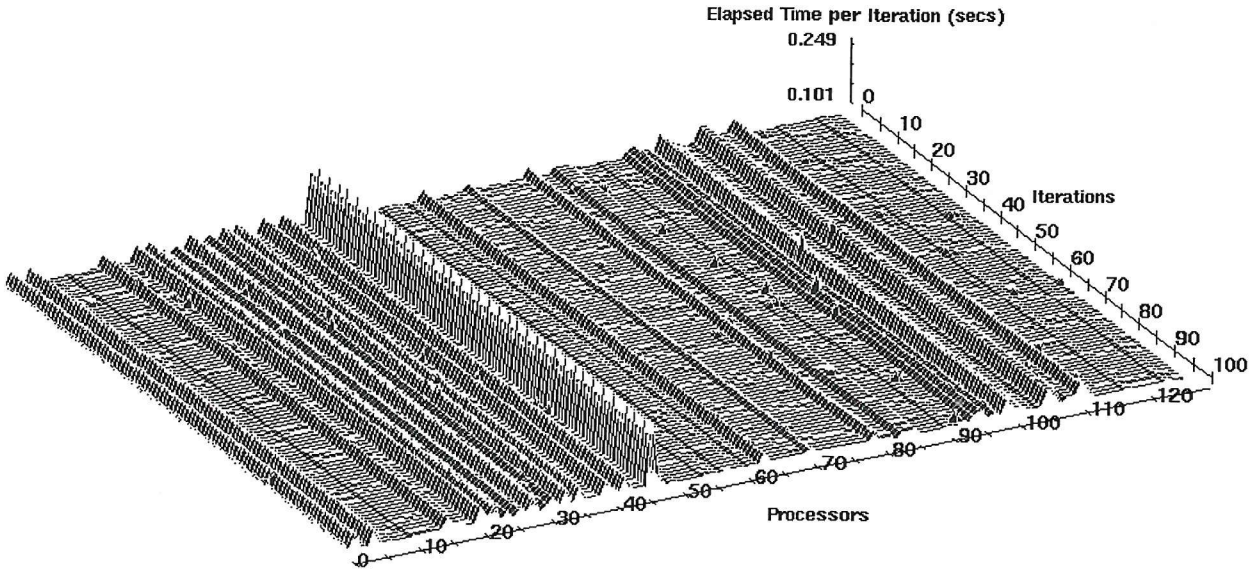

Figure 6.36: DLS: Scalability test (IBM SP2, 128 processors). Notice that some processors take longer to execute the same number of operations (there are no communications inside the do-loop being analyzed).

```
do iter=1,100
  CALL MPE_LOG_EVENT(1, iter, "DLSijk-start")
  do i=1,max_iter
    do j=1,max_iter
      do k=1,max_iter
        c(i,j) = a(i,k)*b(k,j) + c(i,j)
      enddo
    enddo
  enddo
  CALL MPE_LOG_EVENT(2, iter, "DLSijk-end")
enddo
```

Figure 6.37: Do-loop code of our scalability test. Notice that there are no communications inside the loop and that the same operations are repeated on every processor.

6.6 Comments and Conclusions

The experiments presented in this thesis in general and in this Chapter in particular, have successfully demonstrated that the DLS representation of performance is an useful tool for analyzing and evaluating parallel program performance. The following is a summary including good points, weak points, and possible future improvements for the DLS abstraction.

DLS - good points:

- Easy to use.
- Scalable representation of performance (tested up to 128 processors).
- Useful in understanding parallel program behaviour (e.g. Section 6.1), identifying hardware irregularities (e.g. figure 5.1), comparing and evaluating communication patterns and different versions of algorithms (e.g. Sections 6.5.3 and 5.3.1), identifying different sorts of system irregularities such as unexpected operating system overheads and memory behaviour (e.g. Sections 5.3.1 and 6.3), identifying load balancing problems, and validating results of other performance tools.
- DLS is not restricted to any particular architecture, message passing system or programming model. The only requirements are a DLS trace file and AVS.
- DLS has all the advantages of using a scientific data visualization tool such as AVS. One of the advantages of visualization is that it is a way of seeing the unseen [151].
- Generating the DLS trace files using MPE (Section 6.5.1) makes it possible to generate trace data on every architecture where MPI actually runs, assuring a good degree of portability.

DLS - weak points:

- DLS does not provide detailed information.
- DLS does not isolate the cause of the problem (but helps identify it).
- Volume of data can be difficult to handle if the problem has a very large number of iterations and/or processors (e.g. 1000).
- Visualization is useful, but it is important to give a warning about the *misuse* of visualization. Fred Brooks says that *"The purpose of visualization if to inform, not to impress. If you do inform, then you will impress"* [151]. There are many ways to confuse or impress the users with a nice picture without telling them any useful thing. One has to be very careful when using visualization in order to understand the images we see (e.g. Globus and Raible [165] give 14 ways to say nothing with Scientific Visualization).

DLS - possible improvements:

- One interesting issue is that we could characterize different sorts of performance problems by the *shape* of a DLS. In this way, different shapes would correspond to different problems or behaviours. The benefit for the user would be clear, since it would recognize previously defined problems by just looking at a single picture.

- For a very large number of processors, a dynamic notion of performance instrumentation and measurement can be used [130]. In this way, the amount of information produced can be dramatically reduced.

- We have been working hard to visualize the performance of the whole execution of a particular application. However, only those parts that are affecting the overall performance are relevant to our study. Therefore, one possible improvement would be to collect only *relevant* information, i.e. information related to *interesting* changes in the execution of our program (e.g. collect data only if the elapsed time per do-loop iteration is greater than a certain value).

7 Conclusions

High performance at low cost is the key reason for the existence of parallel machines. More complex problems can be solved by using large parallel computers and in the near future we will have these machines delivering TeraFlop/s performance. No one doubts the future importance of parallel computers to attain the highest performance. However, the delivery of adequate performance is not automatic and performance tools are required in order to help the programmer to *understand* the behaviour of a parallel program. If we can understand the behaviour of our parallel program, we might be able to tune and improve its performance. In recent years, a wide variety of tools have been developed for this purpose including tools for monitoring and evaluating performance and visualization tools. Some relevant issues to be mentioned about performance analysis tools that are derived from the research done in this thesis are:

- Portability, extensibility and scalability are key issues for the success of a performance analysis and evaluation environment.
- A performance analysis methodology is required in order to formalize the data analysis process, making it easier and safer.
- Performance instrumentation must have a low and predictable *invasiveness*, allowing the user to incorporate perturbation analysis models when required.
- Performance analysis is not an isolated process and may require the use of several and different tools. Thus, the ability of *exchanging* information is desirable (e.g. integration of tools by using PCTE [166]).
- Performance data representations that allow scalability must be developed at a higher level of abstraction (then given by present tools). Usefulness and understanding must be present in such representations, which need not be only visual. Alternatives such as aural representations should be explored.

With current technology, an MPP means a machine with a large number of processors (i.e. hundreds or even thousands of processors). Analyzing the performance of a program running on such a machine is not an easy task and even standard performance tools might not be useful. Performance visualization of parallel programs is important because it helps the user to understand complex performance phenomena. In a single picture, it is possible to see what is happening for the whole execution of a program. Additionally, for a human being it is easier to recognize visual patterns than to extract the same information from plain numbers. There are many tools designed for scientific visualization, such as AVS, that are used fundamentally to analyze data in areas such as chemistry, fluid dynamics, and different sorts of simulations of real phenomena (e.g. seismic simulation, reservoir simulation, weather prediction). In order to handle complex data, AVS provides very powerful built in capabilities. One

can transform and display data in many ways, rotate figures, zoom in and zoom out as necessary, change colour scales, create different views of the same data, analyze statistics, etc. The main purpose of integrating performance analysis with scientific data visualization is, precisely, to take advantage of these excellent capabilities for data analysis. Once we have produced the required performance data (e.g. by instrumenting the program's source code), we are ready for the analysis process. Nothing else is required. One of the advantages of this approach is that no tool development is necessary. This is important for several reasons: (1) we can spend more time analyzing our performance data and (2) we do not need to reinvent the wheel. To create a visualization program is not a trivial task at all and this effort may be incorporated to a more productive goal, i.e. the performance analysis process.

In this thesis we have presented our experience on performance analysis of parallel programs. The performance tool, ANDES, invasiveness measurements and the new parallel performance representation, DLS, show the importance of several issues related to the evaluation of parallel performance.

ANDES was an interesting starting point since it gave us the opportunity to learn more about the behaviour of parallel processes. If one wants to measure a parallel program, first we have to understand the internal complex behaviour of processes interacting in parallel. Additionally, we had to design a methodology in order to make the performance evaluation process more efficient. The first version of ANDES was restricted to one particular software platform. However, an enhanced version of ANDES that was developed for PARMACS allowed us to make ANDES more *portable*. Portability is a key issue for the success of a performance analysis tool. Finally, by *integrating* ANDES and *gnuplot* we find an example of the advantages that *extensibility* can give us. The use of existing tools is important because allow us to incorporate facilities and features that are not present in our tool, making the performance evaluation more complete.

Developing a performance analyzer is a very valuable experience. Amongst other things, we learned that performance instrumentation can perturb the application by a significant factor. It was clear for us that the study of *invasiveness* was the next step. Understanding invasiveness makes possible a better control over it. Low and predictable invasiveness is our goal if we want accurate performance data. However, this is a complex problem and more research is required for a better understanding of invasiveness effects and how to control them.

The experiments presented in this thesis have successfully demonstrated that the DLS representation of performance, our main contribution, is an useful tool for analyzing and evaluating parallel program performance. The DLS is scalable, portable, easy to use, useful, it can be used in a collaborative way with other tools and it has all the advantages of using a commercial scientific data visualization tool such as AVS. Additionally, using the MPE profiling library in order to produce the trace data required to display a DLS, makes it possible to have a very portable version based on MPI. Moreover, this version will be freely available in the near future.

Future research directions of this work should consider the use of DLS for different programming paradigms such as HPF, different applications such as databases and different architectures (e.g. mixtures of shared and distributed memory machines). Another area to explore is the use of DLS for the design of compilers (e.g. use a DLS to decide whether one do-loop ordering is better than other). Incorporating *sound* to a DLS will be an interesting test because it will add one additional dimension to

the display (e.g. for real time displaying, when a *relevant* event occurs). Some other important issues that must be considered in current and future research are:

- Integration of tools.
- Automatic tools for parallelizing sequential programs. If these tools become successful, are performance tools still necessary?
- Future of parallel processing (e.g. machines with a still larger number of processors? MIMD and/or Shared Memory Machines?).
- Closer interaction with the user.

Finally, we believe that this work has contributed towards a better understanding of parallel program performance and that our experience can be used for improving the design of future performance analysis environments. If parallel computers are here to stay, then parallel performance visualization is also here to stay.

Bibliography

[1] Eugenio Zabala and Richard Taylor, "Maritxu: Generic Visualisation on Highly Parallel Processing", in *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, Edinburgh, Scotland, April 1992.

[2] Larry Carter, "The RAM Model and the Performance Programmer", Research Report RC 16319, IBM Research Division, T.J. Watson Research Center, NY, November 1990.

[3] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title, "Data Visualization and Performance Analysis in the Prism Programming Environment", in *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, Edinburgh, Scotland, April 1992.

[4] Ian Glendinning, Stephen Hellberg, Piers Shallow, and Martin Gorrod, "Generic visualization and performance monitoring tools for message passing parallel systems", in *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, Edinburgh, Scotland, April 1992.

[5] Bernd Mohr, "Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible?", in *Environments and Tools for Scientific Parallel Computing*, Saint Hilaire du Touvet, France, September 1992.

[6] T. Anderson and E. Lazowska, "Quartz: A tool for tuning parallel program performance", in *Proc. of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990, pp. 115–125.

[7] Kayhan İmre, *Experiences with Monitoring and Visualising the Performance of Parallel Programs*, PhD thesis, University of Edinburgh, Department of Computer Science and Edinburgh Parallel Computing Centre, 1992.

[8] Allen Malony and Gregory Wilson, "Future directions in parallel performance environments", in *Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.

[9] M. Heath and J. Etheridge, "Visualizing the performance of parallel programs", *IEEE Software*, pp. 29–39, September 1991.

[10] Michael Heath, "Recent Developments and Case Studies in Performance Visualization using ParaGraph", in *Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.

[11] Parasoft Corporation, *ParaSoft Express. User's Guide*, 1990.

[12] Sekhar Sarukkai, Doug Kimelman, and Larry Rudolph, "A methodology for visualizing performance of loosely synchronous programs", in *Scalable High Performance Computing Conference, SHPCC-92.* April 1992, pp. 424–432, IEEE Computer Society.

[13] Barton Miller, "What to draw? when to draw? An essay on parallel program visualization", *Journal of Parallel and Distributed Computing*, vol. 18, no. 2, pp. 265–269, 1993.

[14] Larry Carter, "Private Electronic Mail Communication", January 1993.

[15] G. Almasi, B. Alpern, C. Berman, Larry Carter, and D. Hale, "A Case-study in performance programming: Seismic Migration", in *2nd Symposium on High Performance Computing.* October 1991, pp. 195–206, North Holland.

[16] John Merlin, "HPF Visualization Tools - Proposal", Tech. Rep., University of Southampton, April 1993.

[17] Brian Wylie, Michael Norman, and Lyndon Clarke, "High Performance Fortran: A Perspective", The University of Edinburgh, EPCC-TN92-05.04, May 1992.

[18] Advanced Visual Systems Inc., *AVS User's Guide, Release 4*, May 1992.

[19] Oscar Naím and Tony Hey, "Do-Loop-Surface: An Abstract Performance Data Visualization", *Lecture Notes in Computer Science*, vol. 797, pp. 367–372, April 1994.

[20] Oscar Naím, Tony Hey, and Ed Zaluska, "Do-Loop-Surface: An Abstract Representation of Parallel Program Performance", To be published in Concurrency-Practice and Experience, March 1995.

[21] Abdul Waheed, Bernd Kronmüller, Roomi Sinha, and Diane Rover, "A Toolkit for Advanced Perofrmance Analysis", in *International Workshop on Modeling, Analysis, and Simulation of Computers and Telecommunication Systems (MASCOTS'94), Durham NC*, January 1994.

[22] Daniel Reed, Ruth Aydt, Tara Madhyastha, Roger Noe, Keith Shields, and Bradley Schwartz, "The PABLO performance analysis environment", Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[23] Bernd Mohr, "SIMPLE: A Performance Evaluation Tool Environment for Parallel and Distributed Systems", in *EDMCC2*, Munich, April 1991.

[24] Oscar Naím, "Un analizador de desempeño para programas en C Paralelo (ANDES)", Master's thesis, Universidad Simón Bolívar, Computer Science and Information Technology Department, Caracas-Venezuela, March 1992, Supervisor: Dr. Alejandro Teruel.

[25] Oscar Naím and Alejandro Teruel, "ANDES: A Performance Analyzer for Parallel Programs", in *Transputer and Occam Research: New Directions*, 16th Technical Meeting of the World Occam and Transputer User Group (WoTUG-16). Sheffield, UK, March 1993, vol. 33, pp. 91–99, IOS Press, Amsterdam.

[26] Oscar Naím and Tony Hey, "Invasiveness of Performance Instrumentation Measurements on Multiprocessors", *IFIP TRANSACTIONS A-COMPUTER SCIENCE AND TECHNOLOGY*, vol. 44, pp. 319–328, 1994.

[27] Alistair Dunlop, Emilio Hernández, Oscar Naím, Tony Hey, and Denis Nicole, "A Toolkit for Optimising Parallel Performance", *Lecture Notes in Computer Science*, vol. 919, pp. 548–553, May 1995.

[28] Alistair Dunlop, Stephen Hellberg, Tony Hey, Oscar Naím, and David Pritchard, "Environments, Performance Analysis, and Data Visualisation", in *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, Tennessee, USA, May 1994, SIAM, Edited by Jack Dongarra and Bernard Tourancheau.

[29] Thinking Machines Corporation, *Connection Machine CM-5. User's Guide*, August 1993.

[30] Meiko Limited, *CS-2 Product Description*, 1992, Brochure.

[31] BBN Systems and Technologies, *TotalView User's Guide*, April 1994, Version 3.

[32] IBM, *IBM Solutions: Scalable POWERparallel System 2*, Brochure.

[33] Al Geist, Adam Beguelin, Jack Dongarra, Kiang Weicheng, Robert Manchek, and Vaidy Sunderam, "PVM 3 Users's Guide and Reference Manual", Tech. Rep. ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[34] Adam Beguelin, "Xab: A Tool for Monitoring PVM programs", in *Workshop on Heterogeneous Processing*, Los Alamitos, California, April 1993, pp. 92–97, IEEE Computer Society Press.

[35] Adam Beguelin, Jack Dongarra, A. Geist, and V. Sunderam, "Visualization and Debugging in a Heterogeneous Environment", *Computer*, vol. 26, no. 6, pp. 88–95, 1993.

[36] Kohl, Jim, "XPVM Technical Documentation", PVM Version 3.3.x.

[37] "Message Passing INterface Forum", Tech. Rep. Technical Report CS-93-214, November 1993, Document for a standard message-passing interface.

[38] Clarke, Lyndon and Glendinning, Ian and Hempel, Rolf, "The MPI Message Passing Interface Standard", Tech. Rep., 1994, Technical report.

[39] "High Performance Fortran Language Specification", Draft, High performance Fortran Forum, November 1994, Version 1.1.

[40] Bernd Mohr, "Performance evaluation of parallel programs in parallel and distributed systems", in *VAPP IV / CONPAR 90*, Zürich, September 1990.

[41] University Erlangen-Nürnberg, *SIMPLE User's Guide Version 5.3. Part A: TDL Reference Guide; Part B: POET Reference Manual; Part C: Tools Reference Manual; Part D: FDL/VARUS Reference Guide*, 1992.

[42] B. Tourancheau, X. Vigouroux, and M. van Riek, "The massively parallel monitoring system a truly parallel approach to parallel monitoring", in *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, Edinburgh, Scotland, April 1992.

[43] R. Hempel, *The ANL/GMD Macros (PARMACS) in FORTRAN for Portable Parallel Programming using the Message Passing Programming Model. User's Guide and Reference Manual*, Pallas GmbH, 1991.

[44] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, *A Users' guide to PICL, A portable instrumented communication library*, Oak Ridge National Laboratory, 1991, ORNL/TM-11616.

[45] Ian Glendinning, Vladimir Getov, Stephen Hellberg, Roger Hockney, and David Pritchard, "Performance Visualisation in a Portable Parallel Programming Environment", in *Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.

[46] M. Ortega, "Implementación de un Algoritmo Paralelo para la Descomposición LU", Trabajo para la materia Algoritmos Paralelos de la Maestría en Ciencias de la Computación de la Universidad Simón Bolívar, Caracas-Venezuela, April 1991.

[47] M. Abrams, N. Doraswamy, and A. Mathur, "CHITRA - Visual Analysis of Parallel and Distributed Programs in the time, event, and frequency domains", *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 6, pp. 672–685, 1992.

[48] Vincent Guarna, Dennis Gannon, Yogesh Gaur, and David Jablonowski, "FAUST: An Environment for Programming Parallel Scientific Applications", in *Proceedings of Supercomputing'88*, Orlando, Florida, November 1988, pp. 3–10.

[49] Thomas LeBlanc and Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, April 1987.

[50] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski, "IPS-2: The second generation of a parallel program measurement system", *IEEE Transactions on Parallel and Distributed Systems*, pp. 206–217, April 1990.

[51] Frank Lange, Reinhold Kroeger, and Martin Gergeleit, "JEWEL: Design and Implementation of a Distributed Measurement System", *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 6, pp. 657–671, November 1992.

[52] Teemu Kerola and Herb Schwetman, "Monit: A performance monitoring tool for parallel and pseudo-parallel programs", in *Proceedings of the 1987 ACM SIGMETRICS Conference*, May 1987.

[53] A. Goldberg and J. Hennessy, "Mtool - an integrated system for performance debugging shared memory multiprocessor applications", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 28–40, 1993.

[54] Ziya Aral and Ilya Gertner, "Non-Intrusive and Interactive Profiling in Parasight", *ACM Sigplan Notices*, pp. 21–30, 1988.

[55] Daniel Pease, Arif Ghafoor, Ishfaq Ahmad, David Andrews, Kamal Foudil-Bey, Thomas Karpinski, Mohammad Mikki, and Mohamed Zerrouki, "PAWS: A performance evaluation tool for parallel computing systems", *IEEE Computer*, January 1991.

[56] Zary Segall and Larry Rudolph, "PIE: A programming and instrumentation environment for parallel processing", *IEEE Software*, vol. 2, no. 6, November 1985.

[57] David Notkin and et al, "Experiences with Poker", in *ACM SIGPLAN Notices*, 1988.

[58] Patrick Earl Mcclaughry, "Ptopp - a practical toolset for the optimization of parallel programs", Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[59] William Appelbe, Kevin Smith, and Charlie McDowell, "Start/PAT: A Parallel Programming Toolkit", *IEEE Software*, vol. 6, no. 4, pp. 29–40, July 1988.

[60] A. Wagner, S. Chanson, N. Goldstein, J. Jiang, H. Larsen, and H. Sreekantaswamy, "TIPS: Transputer-based interactive parallelizing system", Department of Computer Science, University of Columbia, 1991.

[61] D. Haban and D. Wybranietz, "A hybrid monitor for behavior and performance analysis of distributed systems", *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 197–211, February 1990.

[62] T. Bemmerl and Peter Braun, "Visualization of message-passing parallel programs with the topsys parallel programming environment", *Journal of Parallel and Distributed Computing*, vol. 18, no. 2, pp. 118–128, 1993.

[63] Allen Malony, D. Hammerslag, and D. Jablonowski, "Traceview - a trace visualization tool", *IEEE Software*, vol. 8, no. 5, pp. 19–28, 1991.

[64] Lorenz Schäfers and Christian Scheidler, "A Graphical Programming Environment for Parallel Embedded Systems", in *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, Edinburgh, Scotland, April 1992.

[65] Virginia Herarte and Ewing Lusk, "Studying Parallel Program Behavior with Upshot", Technical Report ANL-91/15, Argonne National Laboratory, Illinois, USA, 1991.

[66] E. Gabber, "VMMP: A practical tool for the development of portable and efficient programs for multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, July 1990.

[67] Roger Hockney, "A case study using Paragraph", Preliminary-draft 4, January 1992.

[68] Roger Hockney, "Performance parameters and benchmarking of supercomputers", *Parallel Computing*, vol. 17, pp. 1111–1130, 1991.

[69] Roger Hockney, "Parameterization of computer performance", *Parallel Computing*, vol. 5, no. 1, pp. 97–103, 1987.

[70] Roger Hockney and I. Curington, "$f_{1/2}$: A parameter to characterize memory and communication bottlenecks", *Parallel Computing*, vol. 10, no. 3, pp. 277–286, 1989.

[71] Roger Hockney, "A framework for benchmark performance analysis", *Supercomputer 48*, pp. 9–22, March 1992.

[72] Roger Hockney and C. Jesshope, *Parallel Computers 2: Architectures, Programming and Algorithms*, Adam Hilger, Bristol and Philadelphia, 1988.

[73] J. Gustafson, "Reevaluating Amdahl's law", *CACM*, vol. 31, no. 5, pp. 532–533, May 1988.

[74] Allan Karp and Flatt H., "Measuring parallel processor performance", *CACM*, , no. 33, pp. 532–533, May 1990.

[75] E. Carmona and M. Rice, "Modeling the serial and parallel fractions of a parallel algorithm", *Journal of Parallel and Distributed Computing*, vol. 13, no. 3, pp. 286–298, 1991.

[76] W. Nagel and M. Linn, "Benchmarking parallel programs in a multiprogramming environment - the PAR-BENCH system", *Parallel Computing*, vol. 17, no. 10-1, pp. 1303–1321, 1991.

[77] X. Sun and J. Gustafson, "Toward a better parallel performance metric", *Parallel Computing*, vol. 17, no. 10-1, pp. 1093–1109, 1991.

[78] J. Worlton, "Toward a taxonomy of performance metrics", *Parallel Computing*, vol. 17, no. 10-1, pp. 1073–1092, 1991.

[79] J. Hollingsworth, R. Irvin, and B. Miller, "The integration of application and system based metrics in a parallel program performance tool", in *Proc. of the 1991 ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, April 1991, pp. 189–200.

[80] Jeffrey Hollingsworth and Barton Miller, "Parallel program performance metrics: A comparison and validation", in *Proceedings of Supercomputing '92, Minneapolis*, Computer Science Department, University of Wisconsin-Madison, November 1992.

[81] M. Kumar, "Measuring parallelism in computation intensive scientific/engineering applications", *IEEE-TC*, pp. 1088–1098, September 1988.

[82] Charles Fineman and Philip Hontalas, "Selective monitoring using performance metric predicates", in *Scalable High Performance Computing Conference, SHPCC-92*. April 1992, pp. 162–165, IEEE Computer Society.

[83] G. Jones, "Measuring the business of a transputer", in *OCCAM User Group Newsletter*. INMOS, January 1990.

[84] X. Zhou, "Bridging the gap between Amdahl's Law and Sandia Laboratory's results", *Communications of the ACM*, vol. 8, no. 32, pp. 1014–1015, 1989.

[85] H. Jordan, "Interpreting parallel processor performance measurements", *SIAM J. Sci. Stat. Comput.*, vol. 8, no. 2, pp. 220–226, March 1987.

[86] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle, "Distributed performance monitoring: Methods, tools and applications", Revised version for IEEE Transactions on Parallel and Distributed Systems, 1992.

[87] Chyi-Ren Dow, Shi-Kuo Chang, and Mary Lou Soffa, "A Visualization System for Parallelizing Programs", in *Proceedings of Supercomputing '92, Minneapolis*, November 1992, pp. 194–203.

[88] Olaf Lubeck, Margaret Simmons, and Harvey Wasserman, "The Performance Realities of Massively Parallel Processors: A Case Study", in *Proceedings of Supercomputing '92, Minneapolis*, November 1992, pp. 403–411.

[89] E. Lusk, "Performance Visualization for Parallel Programs", *Theoretica Chimica Acta*, vol. 84, no. 4, pp. 377–384, 1993.

[90] D. Kimelman and T. Ngo, "The RP3 program Visualization Environment", *IBM Journal of Research and Development*, vol. 35, no. 5, pp. 635–651, 1991.

[91] D. Rover, G. Prabhu, and C. Wright, "Visualization of Program Performance on Concurrent Computers", *Lecture Notes in Computer Science*, vol. 507, pp. 154–160, 1991.

[92] E. Luque, R. Suppi, J. Sorribes, M. Mayosky, and M. Senar, "Simulation and Visualization Tools for Link-based Parallel Architectures", *Microprocessing and Microprogramming*, vol. 32, no. 1, pp. 479–486, 1991.

[93] K. Nichols, "Performance tools", *IEEE Software*, vol. 7, no. 3, 1990.

[94] H. Burkhart and R. Millen, "Performance-measurements tools in a multiprocessor environment", *IEEE Transactions on Computers*, vol. 38, no. 5, pp. 725–737, 1989.

[95] A. Ferscha, "A petri net approach for performance oriented parallel program design", *Journal of Parallel and Distributed Computing*, vol. 15, no. 3, pp. 188–206, 1992.

[96] G. Balbo, S. Donatelli, and G. Franceschinis, "Understanding parallel program behavior through petri net models", *Journal of Parallel and Distributed Computing*, vol. 15, no. 3, pp. 171–187, 1992.

[97] A. Kapelnikov, R. Muntz, and M. Ercegovac, "A methodology for performance analysis of parallel computations with looping constructs", *Journal of Parallel and Distributed Computing*, vol. 14, no. 2, pp. 105–120, 1992.

[98] D. Menasce and L. Barroso, "A methodology for performance evaluation of parallel applications on multiprocessors", *Journal of Parallel and Distributed Computing*, vol. 14, no. 1, pp. 1–14, 1992.

[99] N. Karonis, "Timing parallel programs that use message passing", *Journal of Parallel and Distributed Computing*, vol. 14, no. 1, pp. 29–36, 1992.

[100] C. Grassl, "Parallel performance of applications on supercomputers", *Parallel Computing*, vol. 17, no. 10-1, pp. 1257–1273, 1991.

[101] Alva Couch and David Krumme, "Portable execution traces for parallel program debugging and performance visualization", in *Scalable High Performance Computing Conference, SHPCC-92.* April 1992, pp. 441–446, IEEE Computer Society.

[102] Alva Couch and David Krumme, "Monitoring parallel executions in real time", 1990, IEEE Computer Society Press.

[103] Joan Francioni and Diane Rover, "Visual-Aural representations of performance for a scalable application program", in *Scalable High Performance Computing Conference, SHPCC-92.* April 1992, pp. 433–440, IEEE Computer Society.

[104] Joan Francioni, L. Albright, and J. Jackson, "Debugging parallel programs using sound", in *SIGPLAN Notices*, 1991, vol. 26, pp. 68–75.

[105] Joan Francioni, L. Albright, and J. Jackson, "The sounds of parallel programs", in *Proceedings of the Sixth Distributed Memory Computing Conference.* 1991, IEEE Computer Society.

[106] D. Dyer, "A dataflow toolkit for visualization", *IEEE Computer*, pp. 60–69, July 1990.

[107] E. Gelenbe, *Multiprocessor Performance*, Wiley Series in Parallel Computing, John Wiley and Sons, 1989.

[108] Ted Lewis and Hesham El-Rewini, *Introduction to Parallel Computing*, Prentice-Hall, International Editions, 1992.

[109] R. Nelson, D. Towsley, and A. Tantawi, "Performance analysis of parallel processing systems", *IEEE Transactions on Software Engineering*, vol. 14, no. 4, pp. 532–540, April 1988.

[110] T. Lehr, Zary Segall, D. Vrsalovic, E. Caplan, A. Chung, and C. Fineman, "Visualizing performance debugging", *IEEE Computer*, October 1989.

[111] C. Yang and B. Miller, "Performance measurement for parallel and distributed programs: A structured and automatic approach", *IEEE Transactions on Software Engineering*, vol. 15, no. 12, December 1989.

[112] Margaret Simmons and Rebecca Koskela, *Performance Instrumentation and Visualization*, ACM Press, Frontier Series, 1990.

[113] Margaret Simmons, Rebecca Koskela, and I. Bucher, *Instrumentation of Future Parallel Computing Systems*, ACM Press, Frontier Series, Addison Wesley, 1989.

[114] S. Thakkar, "Parallel Programming: A Performance Perspective", in *Parallel Computing: Performance Instrumentation and Visualization.* Addison Wesley, 1990, R. Koskela and M. Simmons.

[115] Robert Fowler, Thomas LeBlanc, and John Mellor-Crummey, "An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors", in *Proceedings, ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988, Special issue of SIGPLAN Notices, 24(1), January, 1989.

[116] R. McLaren and W. Rogers, "Instrumentation and Performance Monitoring of Distributed Systems", in *IEEE Computer Society Press*, 1990.

[117] M. Benten and H. Jordan, "Multiprogramming and the Performance of Parallel Programs", in *Proceedings, 3rd SIAM Conf. on Parallel Processing for Scientific Computing*, Los Angeles, California, December 1987.

[118] R. Carpenter, "Performance Measurement Instrumentation for Multiprocessors Computers", in *High Performance Computer Systems.* Ed. Elsevier Science Publishing Co, New York, 1988, E. Gelenbe.

[119] J. Roberts, J. Antonishek, and A. Mink, "Hybrid Performance Measurement Instrumentation for Loosely-Coupled MIMD Architectures", in *Proceedings of the Fourth Conf. on Hypercube Concurrent Computers and Applications*, Monterrey, California, March 1989.

[120] Wolfgang Kastner and Ulrich Schmidt, "Monitoring of distributed real-time systems: The versatile timing analyzer", in *Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.

[121] Peter Capon, "Understanding the behaviour of parallel systems", in *Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.

[122] Rainer Klar, "Event driven monitoring of parallel systems", in *Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.

[123] Umberto Villano, "Monitoring parallel programs running in transputer networks", in *Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.

[124] D. Jensen and Daniel Reed, "A performance analysis exemplar - parallel ray tracing", *Concurrency-Practice and Experience*, vol. 4, no. 2, pp. 119–141, 1992.

[125] T. LeBlanc, J. Mellor-Crummey, and R. Fowler, "Analyzing parallel program executions using multiple views", *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 203–217, 1990.

[126] Reinhold Weicker, "An Overview of Common Benchmarks", *IEEE Computer*, pp. 65–75, December 1990.

[127] Giovanni Chiola, "Special Issue on Petri Net Modeling of Parallel Computers", *Journal of Parallel and Distributed Computing*, vol. 15, pp. 169–170, 1992.

[128] Warren Harrison, "Tools for Multiple-CPU Environments", *IEEE Software*, pp. 45–51, May 1990.

[129] P. Dauphin, M. Kienow, and A. Quick, "Model-driven Validation of Parallel Programs Based on Event Traces", in *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, Edinburgh, Scotland, April 1992.

[130] Barton et al Miller, "The Paradyn Parallel Performance Measurement Tools", Technical report, Computer Science Department, University of Wisconsin-Madison, 1993.

[131] Arndt Bode and Peter Braun, "Monitoring and visualisation in topsys", in *Workshop on Performance Measurement and Visualization of Parallel Systems*, Moravany, Czecho-Slovakia, October 1992.

[132] Pallas, GmbH, *PA-Tools, Performance Analysis Tools*, 1991.

[133] Oscar Naím, "A Case Study using ANDES: A Performance Analyzer for Parallel Programs", February 1993.

[134] Oscar Naím and Alejandro Teruel, "Consideraciones sobre el Paralelismo en SIMPAR (FASE I)", Technical Report IT-1991-001, Department of Computer Science& Information Technology, Universidad Simón Bolívar, Caracas-Venezuela, 1991.

[135] Mauro Sanabria and Alberto Hung, "Construcción de un Parser de ANSI C", Master's thesis, Universidad Simón Bolívar, Computer Science Department, Caracas-Venezuela, October 1991.

[136] INMOS Limited, *ANSI C Toolset User Manual*, 1990.

[137] Matthew Reilly, *A Performance Monitor for Parallel Programs*, Academic Press, Inc., 1990.

[138] Dionisio Acosta and Margarita Fernández, "Generación de Variables Aleatorias en Paralelo", Trabajo de Grado, Ingeniería de la Computación, Universidad Simón Bolívar, Caracas-Venezuela, January 1992.

[139] Institut für Informatik, Technische Universität München, *TOPSYS User's Overview, Version 1.0*, December 1990.

[140] Allen Malony, Daniel Reed, and H. Wijshoff, "Performance-measurement intrusion and perturbation analysis", *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 4, pp. 433–450, 1992.

[141] J. Gait, "A Probe effect in Concurrent Programs", *Software Practice and Experience*, vol. 16, no. 3, pp. 225–233, 1986.

[142] V. Getov, Tony. Hey, R. Hockney, and I. Wolton, "The GENESIS Distributed-Memory Benchmark Suite - Release 2.1", Technical report, Southampton Novel Architecture Research Centre, University of Southampton, 1993.

[143] Mark Debbage and Mark Hill, *PARMACS on VCR Version 1.1 Installation Notes*, Department of Electronics and Computer Science, University of Southampton, September 1992.

[144] Meiko Limited, 650, Aztec West, Almondsbury, Bristol, BS12 4SD, UK., *CSTools for SunOS*, 1989, Vol. 1 and 2.

[145] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, *PICL, A portable instrumented communication library - C Reference Manual*, Oak Ridge National Laboratory, 1991.

[146] Stephen Hellberg, "PICL for Transputers - v1.0 Usage Notes", Technical report, Department of Electronics and Computer Science - University of Southampton, June 1992.

[147] Bernd Mohr, "Private electronic mail communication", Subject: Measuring Invasiveness, March 1993.

[148] Jörg Beier and Thomas Bemmerl, "Software monitoring of parallel programs", in *CONPAR 88*, 1988.

[149] D. Wybranietz and D. Haban, "Monitoring and Performance Measuring Distributed Systems", in *Proceedings ACM SIGMETRICS Conference*, Santa Fe, 1988, vol. 16, pp. 197–206.

[150] D. Wybranietz and D. Haban, "Monitoring and Measuring Distributed Systems", in *Performance Instrumentation and Visualization*, M. Simmons and R. Koskela, Eds., chapter 2, pp. 27–45. ACM Press, Frontier Series, Addison-Wesley Publishing Company, New York, 1990.

[151] Ken Brodlie, "Scientific Visualization - past, present and future", *Nuclear Instruments and Methods in Physics Research - A*, vol. 354, pp. 104–111, 1995.

[152] Alistair Dunlop, Emilio Hernández, Oscar Naím, Tony Hey, and Denis Nicole, "Collaborative Tools for Parallel Performance Optimization", Technical Report HPCC94-011, Dept. of Electronics and Computer Science, University of Southampton, Southampton S017 1BJ, UK, November 1994, Submitted to Scientific Programming.

[153] Oscar Naím and Tony Hey, "Visualization of cache/memory effects using Do-Loop-Surface displays", Technical Report HPCC94-003, Dept. of Electronics and Computer Science, University of Southampton, Southampton S017 1BJ, UK, 1994.

[154] Abdul Waheed and Diane Rover, "Performance Visualization of Parallel Programs", in *Visualization '93*, San Jose, California, October 1993.

[155] C. Figueira and E. Hernandez, "Benchmarks Specification and Generation for Performance Estimation on MIMD Machines", in *IFIP Transactions in Computer Science and Tecnology*, 1994, vol. 44.

[156] A.N. Dunlop, A.J.G. Hey, and D.A. Nicole, "Parallel performance estimating using memory hierarchy simulation", Technical Report HPCC94-008, Dept. of Electronics and Computer Science, University of Southampton, Southampton S017 1BJ, UK, 1994.

[157] Diane Rover and Charles Wright, "Visualizing the Performance of SPMD and Data-Parallel Programs", *Journal of Parallel and Distributed Computing*, vol. 18, pp. 129–146, 1993.

[158] Alva Couch, "Categories and Context in Scalable Execution Visualization", *Journal of Parallel and Distributed Computing*, vol. 18, pp. 195–204, 1993.

[159] Gropp, William and Lusk, Ewing and Skjellum, Anthony, *USING MPI: Portable Parallel Programming with the Message-Passing Interface*, Scientific and ENgineering Computation Series, MIT Press, 1994.

[160] Cwik, T. and van de Geijn, R. and Patterson, J., "Application of Massively Parallell Computation to Integral Equation Models of Electromagnetic Scattering (Invited Paper)", *J. Opt. Soc. Am. A*, vol. 11, no. 4, April 1994.

[161] Dongarra, Jack and van de Geijn, Robert and Walker, David, "LAPACK Working Note 43: A look at Scalable Dense Linear Algebra Libraries", Technical report, University of Tennessee, Oak Ridge National Laboratory, and University of Texas, May 1992.

[162] Gropp, William and Lusk, Ewing, *Users Guide for the ANL IBM SPx*, MCS, Argonne National Laboratory, January 1995, Draft.

[163] Stunkel, Craig and et al, "The SP2 Communication Subsystems", Tech. Rep., IBM Thomas J. Watson Research Center, August 1994.

[164] Bangalore, P., "Private electronic mail communication", Subject: MPI version of LINPACK, March 1995.

[165] Al Globus and Eric Raible, "Fourteen Ways to Say Nothing with Scientific Visualization", *IEEE Computer*, pp. 86–88, July 1994.

[166] Jonathan Jowett, "Emeraude v12: Pcte based core facilities for software engineering environment frameworks", July 1992.

A Appendices

A.1 Publications

- Oscar Naím, Tony Hey, Ed Zaluska
  Do-Loop-Surface: An Abstract Representation of Parallel Program Performance
  Dept. of Electronics and Computer Science, University of Southampton
  March, 1995. To be published in Concurrency-Practice and Experience.

- Alistair Dunlop, Emilio Hernández, Oscar Naím, Tony Hey, and Denis Nicole
  Collaborative Tools for Parallel Performance Optimization
  Dept. of Electronics and Computer Science
  University of Southampton, Southampton S017 1BJ, UK
  December, 1994. Submitted to Scientific Programming.

- Oscar Naím, Tony Hey
  Invasiveness of Performance Instrumentation Measurements on Multiprocessors
  IFIP TRANSACTIONS A-COMPUTER SCIENCE AND TECHNOLOGY
  Vol. 44, pp. 319-328, 1994.

- Oscar Naím, Tony Hey
  Do-Loop-Surface: An Abstract Performance Data Visualization
  HPCN Europe'94, April 1994, Munich, Germany.

- Oscar Naím, Tony Hey
  A toolkit for parallel performance optimisation
  HPCN Europe'95, May 1995, Milano, Italy.

- Alistair Dunlop, Stephen Hellberg, Tony Hey, Oscar Naím and David Pritchard
  Environments, Performance Analysis, and Data Visualisation
  Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing
  Published by SIAM, May, 1994.

- Oscar Naím, Alejandro Teruel
  ANDES: A Performance Analyzer for Parallel Programs
  Transputer and Occam Research: New Directions
  16th Technical Meeting of the World Occam and Transputer User Group (WoTUG-16). Sheffield, UK. Vol. 33, pp. 91-99, 1993.

A.2 Study visits

- 16th Technical Meeting of the World Occam and Transputer User Group (WoTUG-16). Sheffield, UK, 1993.
- Parallel Summer Institute, Prague, 1993.
- HPCN Europe 94, Munich, Germany, April, 1994.
- International Conference on Applications in Parallel and Distributed Computing, Caracas, Venezuela, April, 1994.
- HCM-MAP Meeting, Nice, France, January, 1995.
- Argonne National Laboratory, Illinois, 1995.
- HPCN Europe 95, Milano, Italy, May, 1995.