Parallel Computing Environments

Simone de L. Martins[*], Celso C. Ribeiro[§], and Noemi Rodriguez[‡]

Abstract: This article presents a survey of parallel computing environments available for the implementation of parallel algorithms for optimization problems. These parallel environments are composed by programming tools, performance evaluation tools, debuggers, and optimization libraries. Programming tools, such as PVM, MPI, and Linda, and parallel optimization libraries, such as CPLEX parallel solvers and OSLp, are described. Performance evaluation tools and debuggers are also presented. References to parallel implementations of optimization algorithms for solving combinatorial problems are given.

1 Introduction

The demand for faster and more efficient computer systems for both scientific and commercial applications has grown considerably in recent times. Physical limitations and high costs make it impossible to increase processor speed beyond certain limits. To overcome these difficulties, new architectures emerged introducing parallelism in computing systems. Currently, different types of high performance machines based on the integration of many processors are available. Among these, some of the most important are (Lau 1996): vector multiprocessors (a small number of high performance vector processors); massively parallel processors (hundreds to millions of processors connected together with shared or distributed memory); and networked computers (workstations connected by a medium or high speed network, working as a high performance virtual parallel machine).

Parallelism leads to the need for new algorithms, specifically designed to exploit concurrency, because many times the best parallel solution strategy cannot be achieved by just adapting a sequential algorithm (Foster 1995; Kumar et al 1994). In contrast to sequential programs, parallel algorithms are strongly dependent on the computer architecture for which they are designed. Programs implementing these

[*] National Research Network, Estrada Dona Castorina, 110, Rio de Janeiro 22460-320, Brazil. E-mail: simone@nc-rj.rnp.br
[§] Catholic University of Rio de Janeiro, Department of Computer Science, R. Marquês de São Vicente 225, Rio de Janeiro 22453-900, Brazil. E-mail: celso@inf.puc-rio.br
[‡] Catholic University of Rio de Janeiro, Department of Computer Science, R. Marquês de São Vicente 225, Rio de Janeiro 22453-900, Brazil. E-mail: noemi@inf.puc-rio.br

algorithms are developed and executed in parallel computing environments, composed by a parallel computer and the associated software (Chapman et al 1999; Wolf and Kramer-Fuhrmann 1996). The software consists of parallel programming tools, performance tools and debuggers associated to them, and some libraries developed to help in solving specific classes of problems (Blackford et al 1997; IBMb 1999; Merlin et al 1999; Nieplocha et al 1994; Plastino et al 1999). Programming tools typically provide support for developing programs composed by several processes that exchange information during execution. The performance of systems developed with a programming tool and the available support for communication and synchronization among the various components of a parallel program are some important issues. The programming tool should also match the programming model underlying the parallel algorithm to be implemented.

Ease of programming and the time spent by a programmer to develop a program may be sometimes fundamental. Performance tools can help a programmer to understand and improve parallel programs, by monitoring the execution of a parallel program and producing performance data that can be analyzed to locate and understand areas of poor performance. Debugging parallel programs can be a hard task, due to their inherent non-determinism. Parallel debugging tools developed to specifically debug parallel programs can be very helpful in finding bugs of a program.

This text is organized as follows. In Section 2, we introduce some basic parallel programming concepts related to memory organization, communication among processors, and parallel programming models. Some programming tools available for the development of parallel programs are discussed in Section 3. Next, we present in Section 4 some performance tools associated with these programming tools. The use of debuggers is discussed in Section 5. Two parallel optimization libraries are presented in Section 6. Some concluding remarks are drawn in the last section.

2 Parallel programming concepts

Parallel programs consist of a number of processes working together. These processes are executed in parallel machines, based on different memory organization methods: shared memory or distributed memory (Elias 1995; Kumar et al 1994). In shared memory machines (Protic et al 1998), all processors are able to address the whole memory space, as shown in Figure 1. Task communication is achieved through read and write operations performed by the parallel tasks on the shared memory. The main

advantage of this approach is that access to data is very fast and processor communication is done through the common memory. However, system scalability is limited by the number of paths between the memory and processors, which can lead to poor performance due to access contention. In order to eliminate this disadvantage, some local memory can be added to each processor. This memory stores the code to be executed and data structures that are not shared by the processors. The global data structures are stored in the shared memory, as shown in Figure 2. One extension to this type of architecture consists of removing all physical memory sharing, as shown in Figure 3. For both models, references made by one processor to the memory of another processor are mapped by the hardware that interconnects memory and processors. Since access time to local memory is much smaller than to remote memory, this model can improve memory access time. A drawback of these architectures is that the control of memory access, necessary to maintain data consistency, has to be done by the programmer.

In the context of distributed memory machines, memory is physically distributed among the processors. Each processor can only address its own memory, as shown in Figure 4. Communication among processes executing on different processors is performed by messages passed through the communication network. Networks of workstations are important examples of this architecture, whose importance has been growing in the last years. Besides allowing for scalability and fault-tolerance, networks of computers present good cost/performance ratios when compared to other parallel machines. This has led to a recent explosion in the use of *clusters* of computers, which are basically high speed networks connecting commercial off-the-shelf hardware components.

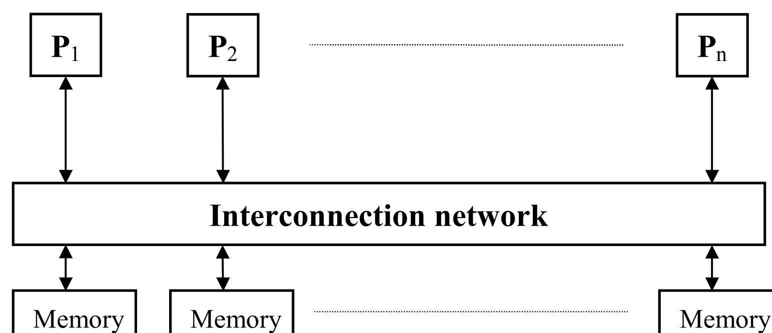

Figure 1: Shared memory architecture

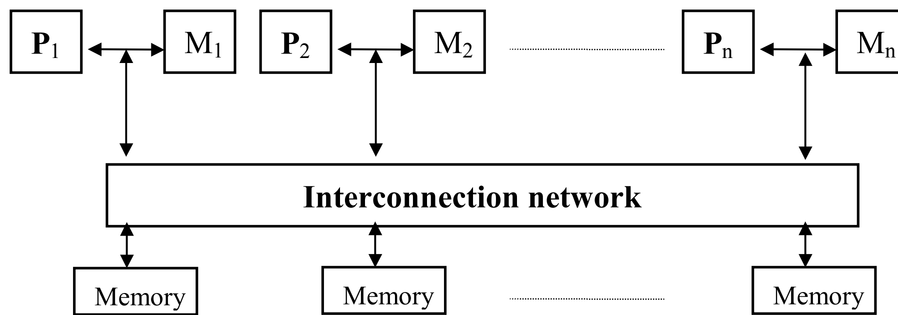

Figure 2: Shared memory architecture with local memory for processors

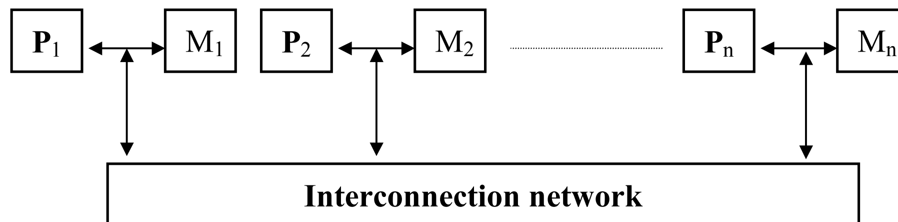

Figure 3: Shared memory architecture with only local memory for processors

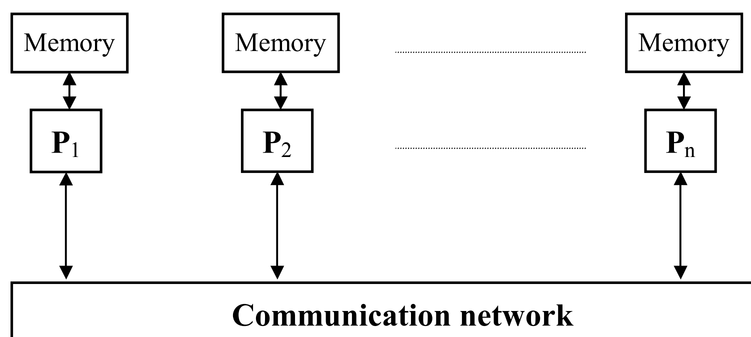

Figure 4: Distributed memory architecture

Two basic communication modes are used for the exchange of information among processes executing on a parallel machine. Tasks may communicate by accessing a common memory (shared memory paradigm), or by exchanging messages (message passing paradigm) (Andrews and Schneider 1983). In the shared memory communication model, tasks share a common address space, which they can asynchronously access. Mechanisms such as locks and semaphores (Andrews and Schneider 1983; Dijkstra 1968; and Vahalia 1996) are used to control access to the shared memory. This communication model can be directly used on different shared memory parallel machines, but can also be simulated on distributed memory machines. In the latter, the user works with a shared memory physically distributed among the processors. One way to implement this mechanism is to use a managing process that (i) translates each address of the shared memory to its actual physical location, and (ii) is responsible for sending messages to the processes in order to read or write data requested by a task.

In the message passing model, processors communicate by sending and receiving messages in blocking or non-blocking mode. In the blocking sending mode, a task sending a message stops its processing and waits until the destination task receives it, while in the non-blocking mode it sends the message and continues its execution. In the blocking receiving mode, a task stops its processing until a specific message arrives, while in the non-blocking mode it just checks whether there is a message for it and carries on its processing if not. The message-passing model can be directly used for distributed memory machines. It can also be very easily simulated for shared memory machines (Tanenbaum 1992).

Many parallel programming tools using shared-memory or message passing models are essentially sequential languages augmented by a set of special system calls. These calls provide low-level primitives for message passing, process synchronization, process creation, mutual exclusion , and other functions. Some of these tools are presented in next section.

There are two basic types of parallelism that can be explored in the development of parallel programs: data parallelism and functional parallelism (Foster 1995; Kumar et al 1994; and Morse 1994). In the case of data parallelism, the same instruction set is applied to multiple items of a data structure. This programming model can be easily implemented for shared memory machines. Data locality is a major issue to be considered, regarding the efficiency of implementations in distributed memory machines. Several languages called data-parallel programming languages have been developed to help with the use of data parallelism. One of these languages, HPF, is presented in the next section.

In the case of functional parallelism, the program is partitioned into cooperative tasks. Each task can execute a different set of functions/code and all tasks can run asynchronously. Data locality and the amount of processing within each task are important concerns for efficient implementations.

The most suitable approach varies from application to application. As discussed in (Foster 1995), functional and data parallelism are complementary techniques which may sometimes be applied together to different parts of the same problem or may be applied independently to obtain alternative solutions.

3 Parallel programming tools

Many programming tools are available for the implementation of parallel programs, each of them being more suitable for some specific problem type. The choice of the parallel programming tool to be used depends on the characteristics of each problem to be solved. Some tools are better e.g. for numerical algorithms based on regular domain decomposition, while others are more appropriate e.g. for applications that need dynamic spawning of tasks and irregular data structures. Many parallel programming tools have been proposed, and it would be impossible to survey all of them in this paper. We have chosen a few tools which are currently in a stable state of development and have been used in a number of applications, being representative of different programming paradigms.

3.1 PVM

PVM (*Parallel Virtual Machine*) (Geist et al 1994) is a widely used message passing library, created to support the development of distributed and parallel programs executed on a set of interconnected heterogeneous machines.

A PVM program consists of a set of tasks that communicate within a parallel virtual machine by exchanging messages. A configuration file created by the user defines the physical machines that comprise the virtual machine. A managing process executes on each of these machines and controls the sending and receiving of messages among them. There are subroutines for process initialization and termination, for message sending and receiving, for group creation, for coordinating communication among tasks of a group, for task synchronization, and for querying and dynamically changing the configuration of the parallel virtual machine. The application programmer writes a parallel program by embedding these routines into a C, C++, or FORTRAN code.

A public domain software can be downloaded from http://www.netlib.org/pvm3/index.html. PVM has been used on all the following systems: (i) PCs running Win 95, NT 3.5.1, NT 4.0, Linux, Solaris, SCO, NetBSD, BSDI, and FreeBSD, (ii) workstations and shared memory servers executing Sun OS, Solaris, AIX, Hpux, OSF, NT-Alpha, and Iris, and (iii) parallel computers, such as CRAY YMP, T3D, T3E, Cray2, Convex Exemplar, IBM SP-2, 3090, NEC SX-3, Fujitsu, Amdahl, TMC CM5, Intel Paragon, Sequent Symmetry, and Balance. The developers provide efficient support through electronic mail.

PVM has been used in several implementations of exact algorithms to optimization problems. A parallel implementation of the nested Benders solution algorithm for the large scale certainty equivalent LP of a dynamic stochastic linear program was developed and tested on networks of workstations and on IBM SP-2 (Dempster and Thompson 1997). Linderoth et al (1999) describe a parallel linear programming-based heuristic to solve set partitioning problems producing good solutions in a short amount of time. Zakarian (1995) presents an efficient method for large-scale convex cost multi-commodity network flow problems and its parallelization using a cluster of workstations. A unified platform for implementing parallell branch-and-bound algorithms was developed at the Prism Laboratory (Benaïchouche et al 1996, Le Cun et al 1995).

Several parallel metaheuristics for combinatorial optimization problems were also implemented using PVM. Taillard et al (1997) describe a parallel implementation of the tabu search metaheuristic (Glover 1989, 1990; Glover and Laguna 1997) for the vehicle routing problem with soft time windows. A synchronous parallel tabu search algorithm for task scheduling in heterogeneous multiprocessor systems under precedence constraints was developed and implemented by Porto and Ribeiro (1995a, 1995b, 1996) and Porto et al (1999). Aiex et al (1998) present an implementation of a cooperative multi-thread parallel tabu search for the circuit partitioning (Andreatta and Ribeiro 1994). These last two implementations were executed on an IBM SP-2. The implementation of master-slave and multiple-population parallel genetic algorithms can be found in (Cantú-Paz 99).

3.2 MPI

MPI (*Message Passing Interface*) (Foster 1995; MPI Forum 1994; MPI Forum 1995; Gropp et al 1994; Gropp et al 1998) is a proposal for the standardization of a message passing interface for distributed memory parallel machines. The aim of this standard is to enable program portability among different parallel machines. It just defines a message passing programming interface, not a complete parallel

programming environment. For this reason, it does not handle issues such as parallel program structuring, and debugging. It does not provide any definition for fault tolerance support and assumes that the computing environment is reliable.

The core of MPI is formed by routines for point-to-point communication between pairs of tasks. These routines can be activated in two basic modes: blocking or non-blocking. Three communication modes are available: ready (a process can only send a message if there is a corresponding reception operation initiated); standard (a message can be sent even if there is no reception operation for it), and synchronous (similar to the standard mode, but with a sending operation considered as completed only after its destination processor initiates a reception operation). The selection of the mode to be used depends on the type of communication needs of the application and can have a great influence in the efficiency of the parallel program. Group communication routines are defined to coordinate the communication among tasks belonging to a predefined group of processors. These routines allow group data transmission providing the following operations, see Figure 5: sending the same data from one member of the group to all others (broadcast); sending different data from one member to all others (scatter); sending data from all members to one specific member (gather); sending data from all members to all members (allgather); and sending different data from each member to all others (alltoall).

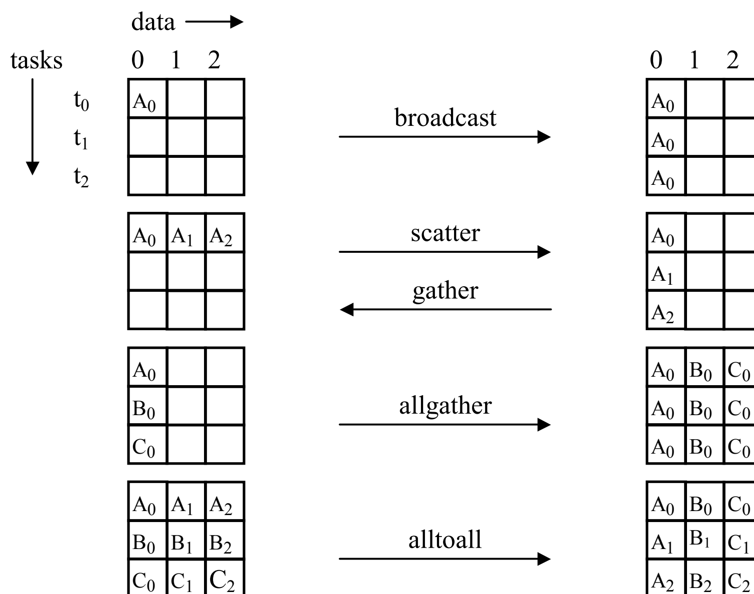

Figure 5: MPI routines for global communication

There is also a reduction operation for computing functions over data belonging to members of a group of processors. The reduction operation applies a predefined or a user defined function to data placed on the different participating processors. The result of this function can be sent to all group members, or just to one of them. One example is the determination of the maximum of a set of numbers, in which each number belongs to the address space of a different group task.

Many implementations of the MPI standard are available. All of them are based on a parallel virtual machine, composed by several connected heterogeneous computers (workstations, PCs, multiprocessors) and each of which executes a process used to control the exchange of messages among these machines. Among these implementations, *mpich* was developed at the Argonne National Laboratory based on the MPI 1.1 standard. It is free and can be downloaded from http://www-unix.mcs.anl.gov/mpi/mpich. This tool currently runs on the following parallel machines: IBM SP-2, Intel Paragon, Cray T3D, Meiko CS-2, TMC CM-5, NCUBE NCUBE-2, Convex Exemplar, SGI Challenge, Sun Multiprocessors, CRAY YMP, and CRAY C-90. It also runs on PCs (running Linux, FreeBSD, or Windows), and Sun, HPO, SGI, IBM RS/6000, and DEC workstations. LAM is another free package that can be downloaded from http:/www.mpi.nd.edu/lam, based on the MPI 2.0 standard and developed at the Laboratory for Scientific Computing of the University of Notre Dame. This implementation provides some monitoring functions, which are very helpful for managing processes running on the physical machines. It has already been tested and verified on Solaris, IRIX, AIX, Linux, and HP-UX. Both tools can be quite easily installed and used.

Some companies have also developed specific implementations for their parallel machines, such as Alpha, Cray, IBM, SGI, DEC, and HP. The Cornell Theory Center provides many helpful tutorials and FAQ sections about MPI at http://www.tc.cornell.edu/UserDoc/SP/.

We can find some parallel implementations of exact algorithms for solving optimization problems using MPI. A portable C++ class library was developed by Eckstein et al (1997) for portable, modular, parallel implementation of branch-and-bound and related algorithms. ZRAM (Marzetta et al 1998) is a portable parallel library of exhaustive search algorithms and has been modeled on a subset of the standard message passing interface MPI. Brüngger et al (1997) show the use of this library for solving some test-instances of the quadratic assignment problem. Homer (1997) describes the development and implementation of a parallel algorithm to approximate the maximum weight cut in a weighted undirected graph. A generic

framework for coarse grained parallel branch-and-bound algorithms supporting dynamic load balancing can be found in (Batoukov and Serevik 1999).

Several parallel algorithms using metaheuristics were implemented using MPI. Martins et al (1998, 1999) developed parallel implementations of greedy adaptive randomized search procedures (GRASP, see e.g. (Feo and Resende 1995)) to solve the Steiner problem in graphs. A general object-oriented library based on C++ and MPI, developed by Kliewer and Tschöke (1998), provides a framework to have a simulated annealing optimization system, which can be applied to many different optimization problems. Fachat and Hoffmann (1997) describe an implementation of ensemble based simulated annealing with dynamic load balancing to get maximum use of the available processing power. Salhi (1998) reports on a parallel implementation of a tool for symbolic regression, the algorithmic mechanism of which is based on genetic-programming. Rivera-Gallego (1998) presents a fast genetic algorithm to determine three-dimensional configurations of points that generate circulant Euclidean distance matrices and its parallelization.

3.3 Linda

Linda (Carriero et al 1989, 1994) is a language that offers a collection of primitives for process creation and communication, based on the concept of an associative shared memory. The shared memory is referred to as the *tuplespace* and consists of a collection of data registers (or *tuples*). Primitives offered by Linda are such that processes may write into the tuplespace, as well as read and delete tuples from the tuplespace according with some specified pattern. In the case of associative access to the memory, information is accessed by its contents, not by its address. Although the tuplespace is logically shared by the processes, it can be implemented in distributed memory environments, with different memory partitions distributed among the processors.

Whenever a process wants to communicate with another one, it generates a new data object (a tuple) and places it in the tuplespace. The receiver process may access this tuple because it is written in the shared memory space. To create a new concurrently executing process, the generator process performs an *eval* operation that creates a process tuple consisting of the fields specified as its arguments and then returns. This process tuple implicitly creates a process to evaluate each argument. While any of these processes runs, the process tuple is referred to as a "*live tuple*". When all processes complete their execution, the

values to be returned are placed in the corresponding field and the resulting data tuple is placed into the tuplespace. The live tuple carries out some specified computation on its own, independently of the process that generated it. The operations to manage tuples are extensions to C, C++, and FORTRAN. The Linda compiler generates executable programs. The programmer develops a parallel program implementation by writing a C, C++, or FORTRAN code using the basic Linda operations to access the tuplespace: read a tuple, write a tuple, place a new tuple in the tuplespace, take a tuple from the tuplespace, and create a new process.

Linda is a commercial system implemented and distributed by *Scientific Computing Associates* for shared memory and distributed memory machines. It runs on Cray T3D/T3E, DEC Alpha, Hitachi SR2201, and HP900/7xx machines. Network Linda is another product developed by *Scientific Computing Associates* (Scientific Computing Associates 1995) to implement Linda in distributed memory environments. This version runs on IBM RS/6000, IBM PowerPC, IBM SP-2, SGI Iris, Indy, Onyx, Power Indy and Power Challenge, and Sun Sparc machines.

An implementation in Linda of a genetic algorithm can be found in (Davis 1994). Aiex et al (1998) describe an implementation in Linda, running on an IBM SP-2, of a cooperative multi-thread parallel tabu search for the circuit partitioning problem, comparing the results with those obtained with the use of PVM.

3.4 HPF

FORTRAN 90 provides constructs for specifying concurrent execution based on data parallelism. HPF (High Performance FORTRAN Forum 1997; Knies et al 1994; Koelbel et al 1994; Merlin and Hey 1995) extends FORTRAN 90 with additional parallel constructs and data placement directives. A data parallel program developed using both languages is a sequence of operations that are applied to some or all elements of an array.

There are implicit and explicit constructs. In FORTRAN 90, the explicit construct `A=B*C` can be used to specify that each element of an array `A` is to be assigned the product of the corresponding elements of arrays `B` and `C`. A do-loop is an implicit parallel construct: the compiler determines whether the iterations are independent from one another, and, if so, executes them concurrently. A data parallel program is a

sequence of explicit and implicit parallel statements. Typically, a compiler translates these statements into a program that controls the distribution and the execution of these operations on the data at each processor. Thus, the responsibility for low level programming details is transferred from the programmer to the compiler.

There are six HPF compiler directives that specifically allow the programmer to control processor mapping and data distribution onto the physical processors: ALIGN, DISTRIBUTE, PROCESSORS, TEMPLATE, REDISTRIBUTE, REALIGN, and DYNAMIC. Initially, the programmer has a group of arrays or subsets of arrays which are related by the ALIGN directive. These aligned objects are then distributed to an abstract set of processors via the DISTRIBUTE directive. Finally, the resulting abstract set of processors is mapped to the physical processors via a compiler dependent procedure, which is not specified in HPF and can vary according to the implementation. REALIGN and REDISTRIBUTE are the dynamic forms of the DISTRIBUTE and ALIGN directives, allowing data mapping to change during program execution. The PROCESSORS directive is used to specify the shape and the size of a set of abstract processors. The TEMPLATE directive defines a conceptual object to be used for alignment and distribution operations.

There are some commercial implementations of HPF provided by Applied Parallel Research (Applied Parallel Research 1999), IBM (Gupta et al 1995), Digital (Harris et al 1995), and Portland Group (Schuster 1994). Adaptor (SCAI-GMD 99) is a public domain HPF compilation system, which offers the use of HPF for parallel computing on distributed memory machines with explicit message passing routines. FORGE (Applied Parallel Research 1999) is an integrated collection of interactive tools to support the parallelization of sequential FORTRAN programs. It consists of a FORTRAN program browser for code analysis, a distributed memory parallelizer, and a parallel performance profiler and simulator. It also provides FORTRAN 90 and HPF support.

The parallelization of a commodity trade model using FORGE is described in (Bergmark 1995 and Bergmark and Pottle 1994). The resulting program was executed in a SP1 machine.

3.5 Threads

Threads are a general operating systems concept, not specifically related to parallel programming. However, due to their importance in providing support for concurrent programming, and to their extensive use in many of the tools discussed in other sections, their understanding is essential to a parallel programmer.

Many operating systems and libraries have used the term *thread* in the last years to describe the execution of a sequence of statements, maintaining the process as the resource allocation unit. In this approach, several threads may execute in the context of a single process. These threads (sometimes called *lightweight processes*) can then communicate using the global memory allocated to the associated process (global variables). Programming with threads is specially useful in shared-memory machines, since in this case the global variables model the shared memory to which all processors have access. However, threads can be useful for parallel programming even in distributed memory environments, since it is often useful to use concurrency within each machine, for instance to allow overlapping communication and computation.

In some systems, the operating system is aware of the existence of threads, while in others they are implemented as user-level libraries. Switching between threads is much faster when thread management is done at user-level. Since in this case the scheduler is not aware of the existence of separate threads, care must be taken to avoid invoking any blocking system call, for this would imply in blocking all threads in the process. Both schemes are currently in use.

In 1995, IEEE defined a standard Application Program Interface (API) for programming with threads, known as *POSIX threads* or *Pthreads* (IEEE 95). Several operating systems currently adopt this standard, which is also implemented by a public library (Provenzano 1998, Butenhof 1997). The Pthreads interface basically defines routines for control, synchronization, and scheduling. The control group includes functions such as thread creation, elimination, and suspension. Synchronization is provided by mutual exclusion locks and condition variables. Finally, scheduling routines allow the programmer to define relative priorities for the several threads within a process.

A parallel cut generation procedure for service cost allocation in transmission systems (Aiex et al 1999) was implemented using Posix Threads on a Sun Starfire ENT10000 and results showed a significant reduction in terms of the overall elapsed times, with respect to the sequential version.

Although the Pthreads API has been used in a number of parallel implementations, it is many times considered a "low-level standard", more targeted at concurrent and distributed applications than at high-performance computing (Dagum 1997, Dagum 1998). Besides, Pthreads is defined for C, and there is currently no standard binding between Pthreads and FORTRAN, making it difficult to use threads to parallelize existing FORTRAN codes on multiprocessor architectures. These reasons led to the proposal of the OpenMP industry standard. OpenMP was jointly defined by Digital Equipment Corp., IBM, Intel Corporation, Kuck & Associates Inc. (KAI), and Silicon Graphics/Cray Research. Information about OpenMP can be obtained from http://www.openmp.org. OpenMP consists basically of a set of compiler directives and API that extend FORTRAN (OpenMP Architecture Review Board 1997) (or alternatively, C and C++ (OpenMP Architecture Review Board 1998)).

Kuhn (1999) compares the use of Pthreads and OpenMP in a genetic application, a linkage analysis program. Hybrid approaches, using threads for fine-grained parallelism on multiprocessor machines and libraries such as PVM and MPI for coarse-grained distributed-memory programming, are discussed in (Bova et al. 1999).

4 Performance evaluation tools

It is very important to use empirical data in order to evaluate the performance of parallel implementations. The first step towards performance improvement consists in collecting suitable and reliable execution data (Foster 1995). There are three basic categories of data collectors: profilers, counters, and event traces.

Profilers record the amount of time spent in different parts of a program, usually in subroutines. This information is often obtained by sampling the program counter and generating a histogram of the execution frequencies. These frequencies are combined with compiler symbol table information to estimate the amount of time spent in different program components. Most profiling tools are vendor supplied and machine specific. Usually, the user specifies an option for the compiler to generate data for profiling during the execution of the program. A parallel profiler usually gathers and displays information from each task of the parallel program. Profilers can be very helpful to identify the most time consuming parts of a program, allowing program optimization to be more focused. They can also identify program components that do not scale well with the input size or with the number of processors.

Counters record each time a specific event occurs and may require some additional coding by the programmer. They can be used to record global events of the program, such as the number of procedure calls, the total number of messages, and the number of messages exchanged between pairs of processors, which cannot be obtained by profilers.

Event traces record each occurrence of specific events (such as procedure calls or message exchanged) and generate log files containing these records. They can be used to investigate relationships between communications and determine sources of idle time and communication bottlenecks. This approach generates a huge volume of data, which may be difficult to handle and store. The logging of this type of data tends to change program characteristics, masking the original sources of problems (probe-effect). Post-processing is difficult to be performed, due to the large amount of data, and the programmer may have to spend some effort tuning the data collection process in order to record only the relevant events. This technique usually requires some extra coding by the programmer.

Raw data produced by such data collectors is usually transformed to reduce the total data volume, so as to be more easily interpreted and exhibited by visualization tools. Histograms are often used to display data collected by profilers, counters, and event traces. They can display information such as average message size, total computation time, or communication volume, as a function of time. For event traces, some forms of display can provide detailed views of temporal relations between different processors. For example, charts where each bar represents the status of a processor (computing, communicating, or idle) as a function of time. In the remainder of this section, we present some public domain and commercial packages available for performance analysis of the software tools presented in Section 3.

Gprof is a commercial tool provided by IBM to profile parallel programs developed using the IBM MPI library. It shows execution times of the subroutines executed by each processor. Except gprof, all other tools described in this section are based on event traces.

Nupshot is a performance tool distributed with the Argonne National Laboratory *mpich* implementation of MPI. The user inserts calls to logging routines in the source code, compiles and links it with the profiling library called MPE MPI. When the code executes, a log file of time stamped events is generated. This log file can be visualized with nupshot, displaying the state of the processes during the execution of the program.

AIMS (Automated Instrumentation and Monitoring System) (Yan et al 1995, Yan and Sarukkai 1996) is a public domain package that can be downloaded from http://science.nas.nasa.gov/Software/AIMS/ for measurement and analysis of FORTRAN 77 and C message-passing programs written using communication libraries such as MPI and PVM. It can also be used to analyze programs developed with the Portland Group HPF. The source code should be instrumented by a graphical user interface that allows the programmer to specify what constructs should be traced. The source files are compiled and linked with the AIMS monitor library and can be run in the usual manner to produce trace or statistics files. These files can be analyzed by AIMS tools to show the execution of the program based on the trace file.

Paradyn (Miller et al 1995) is a tool for interactive run-time analysis for parallel programs, using message passing libraries such as PVM and MPI. It can be downloaded from http://www.cs.wisc.edu/paradyn/. Before running the program, the user defines performance visualizations, in terms of metric-focus pairs. Metrics can be, for example, processing time, blocking time, or message rates. Focus is a list of program components, such as procedures, processors, or message channels. For example, the user may specify a time visualization with a metric of processing time and focus on a particular subset of processes.

The Pablo performance analysis environment (DeRose et al 1998; Reed et al 1993) available from http://vibes.cs.uiuc.edu/ is a public domain package that consists of several components for instrumenting and tracing parallel MPI programs, as well as for analyzing the trace files produced by the instrumented executables.

VAMPIR (Visualization and analysis of MPI resource) (Galarowicz and Mohr 1998; Nagel et al 1996) is a commercial trace visualization tool from Pallas GmbH available for many different platforms. The programmer instruments the code and links it with the VAMPIRtrace library. The trace files generated by the MPI processes along program execution are automatically collected and merged into a single trace file. The latter can be viewed using VAMPIR with visualization displays configured by the user. It can also be used to analyze programs implemented with HPF translated into an MPI program. Information about this tool is available at http://www.pallas.de/pages/vampir.htm.

VT (Visualization Tool) can be used for post-mortem trace visualization or for on-line performance monitoring of parallel programs developed with the IBM implementation of MPI. It is a commercial tool provided by IBM and can only be used on the IBM SP parallel environment.

XPVM is a public domain software available from http://www.netlib.org/pvm3/xpvm that provides a graphical console and monitor for PVM. The user starts a PVM program using XPVM and every spawned task returns trace event information for real time analysis or for post-mortem playback from saved trace files.

5 Debugging a parallel program

In parallel programming, issues related to concurrency and distribution lead to non-determinism: repeated executions of one same program, over the same input data, may exhibit different behaviors. This can make the task of debugging a parallel program very hard. Some authors even suggest that the programmer should avoid non-determinism whenever possible (for instance, by restricting message origins when executing receive statements) (Foster 1995). This, however, may lead to the sub-exploitation of the potential parallelism of the application.

One common, straightforward way to debug a parallel program is by inserting instructions in the code to print relevant information. Although helpful, the output data lacks temporal information, since the information is not necessarily printed in the same order as generated by the parallel program, due to data buffering in the processors.

PVM and the LAM public domain implementation of MPI allow the programmer to launch debuggers on remote nodes. Tasks can then be executed under the debugger control, making it easier to follow program execution. However, this kind of technique may interfere with the execution of the parallel program, making it behave differently from the normal way it runs. XPVM also provides some simple debugging assistance to PVM programmers, by listing textual details from trace events and remote task output.

Total View is a commercial debugger developed by Etnus Inc. (Etnus 1999) that can be used to debug MPI, PVM, HPF, and multi-threaded programs. It allows the programmer to set breakpoints and action points within parallel programs to control their execution. It also provides data visualization for variables associated with threads or processes, as well as to distributed HPF arrays. Although helpful for debugging parallel programs, it can demand some learning effort from the programmer before its use.

Tuplescope (Scientific Computing Associates 1995) is a very helpful tool for debugging Linda programs. This tool emulates the execution of the parallel program within one single machine. The program can execute in normal or step mode, as with a sequential debugger. Visual indications of process interaction

during the whole duration of the program run and data in specified tuples can be displayed using a graphical interface.

6 Parallel optimization libraries

Optimization libraries consist of a set of subroutines that can be used by application programmers to solve large optimization problems. They include functions to solve linear and mixed-integer programming problems and provide some features, such as control variables (routines to allow users to change a variety of default parameter settings controlling the algorithms), debugging capabilities (routines to provide statistics about element values, such as objective and constraint senses, bounds and coefficients in the constraint matrix, and storage used during execution), and file reading and writing (routines to read various input formats and to generate output solution files). Two widely known commercial optimization systems, CPLEX and OSL, provide parallel implementations of their serial optimization libraries.

CPLEX (CPLEX 1999) is a registered trademark from ILOG. It provides an optimization library for Unix systems and for PCs running DOS, Windows 3.1, Windows NT, Windows 95, and OS/2. The CPLEX parallel solvers work on shared memory multiprocessor parallel architectures, such as DEC, Hewlett-Packard, Silicon Graphics, Sun, and Intel based computer versions for which CPLEX parallel support has been established. Parallel versions of the Barrier Solver and Mixed Integer Solver are available for all parallel plataforms. Parallel simplex availability varies by platform. The user chooses the degree of parallelism to be exploited before starting an optimization and CPLEX automatically implements the parallelization.

OSLp, IBM's Parallel Optimization Subroutine Library (IBMa 1999), includes all functions of the serial library OSL. It runs on the SP IBM RS/6000 Scalable POWERparallel systems and on clusters of RS/6000s workstations. OSLp runs under the AIX parallel environment. OSLp provides a set of over 60 subroutines callable from application programs written in XL FORTRAN or C, to solve linear programming, mixed-integer programming, and quadratic programming optimization problems. Some of these routines use serial algorithms, while others use algorithms which exploit the capabilities of the parallel environment, so that multiple processors may concurrently perform computations on subtasks of the problem to be solved. Only minor changes to serial OSL applications are required to generate parallel

application programs and no explicit parallel coding is required. More information about this library can be found at http://www.rs6000.ibm.com/software/sp-products/oslp.html.

7 Concluding Remarks

The development of portable, efficient, and easy to use software tools, together with the availability of a wide range of parallel machines with large processing capacities, increasing reliability, and decreasing costs, has brought parallel processing into reality as an effective strategy for the implementation of computationally efficient techniques for the solution of large, difficult optimization problems. In this article, we have given descriptions of some parallel processing environments, together with examples of their application in the exact or approximate solution of several optimization problems and applications.

Judicious choice of a programming environment is essential to the development of a parallel program. Implementation issues may lead to programs that perform very poorly, independently of the quality of the underlying parallel algorithm. One possible source of problems is a mismatch between the proposed algorithm and the programming model supported by the development tool. It is also common for the program, once implemented, to have a different behavior from what was expected due to unforeseen communication and processing bottlenecks. Performance evaluation tools have a fundamental role in solving this kind of problem, since they can help the programmer in identifying the origin of such bottlenecks.

We surveyed in this paper only a small fraction of existing work on parallel computing environments. The amount of ongoing research activity in this field is witnessed by the number of workshops and conference sessions devoted to it, such as the "Support Tools and Environments" series in the Euro-Par conferences (Lengauer et al. 1997, Pritchard et al. 1998, Amestoy at al. 1999).

Among the current trends in parallel processing, the concept of cluster computing is one of the leading tendencies. Basically, a cluster is a collection of PCs or workstations connected via a network and using off-the-shelf components. The significant improvements in performance and reliability of PCs, workstations, and high-speed networks strongly contributed to turn the use of clusters into an appealing, low-cost alternative for parallel applications. Clusters can be put together very fast, using operating systems such as public domain Linux, networks such as Ethernet, SCI, or Myrinet, and software environments such as MPI or PVM. Some cluster systems are currently among those with better

cost/performance ratio, ranking among the fastest and most powerful machines. Clusters have been used for solving challenging applications in combinatorial optimization, stochastic programming, power systems, weather modeling, simulations, life sciences, computational fluid dynamics, image processing, data mining, aerodynamics, and astrophysics.

Bibliography

Aiex, R.M., S.L. Martins, C.C. Ribeiro, and N.R. Rodriguez. "Cooperative multi-thread parallel tabu search with an application to circuit partitioning", *Lecture Notes in Computer Science* 1457 (1998), 310-331.

Aiex, R.M., C.C. Ribeiro, and M. V. Poggi de Aragão. "Parallel cut generation for service cost allocation in transmission systems", *Proceedings of the Third Metaheuristics International Conference, (*1999), 1-7.

Amestoy, P., P. Berger, M. J. Daydé, I. S. Duff, V. Frayssé, L. Giraud, and D. Ruiz (eds*), Proceedings of the 5th International Euro-Par Conference* (LNCS 1685), (1999), 89-162.

Andreatta, A., and C.C. Ribeiro. "A graph partitioning heuristic for the parallel pseudo-exhaustive logical test of VLSI combinational circuits", *Annals of Operations Research* 50 (1994), 1-36.

Andrews, G., and F. Schneider. "Concepts and notations for concurrent programming", *Computing Surveys* 15 (1983), 3-43.

Applied Parallel Research. *Applied Parallel Research Home Page*, 1999. (http://www.apri.com)

Batoukov, R., and T. Serevik. "A generic parallel branch and bound environment on a network of workstations", *Proceedings of High Performance Computing on Hewlett-Packard Systems*, (1999), 474-483. (http://www.ii.uib.no/~tors/publications/)

Benaïchouche, M., V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. "Building a parallel branch-and-bound library", *Lecture Notes in Computer Science* 1054 (1996), Springer-Verlag, 201-231.

Bergmark, D.. "Optimization and parallelization of a commodity trade model for the IBM SP1/2 using parallel programming tools*", Proceedings of 1995 International Conference on Supercomputing*, (1995), 227-236.

Bergmark, D., and M. Pottle. *The optimization and parallelization of a commodity trade model for the IBM SP1, using parallel programming tools,* Technical Report CTC94TR181, Cornell Theory Center, 1994. (http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Download/ncstrl.cornell.tc/94-181)

Blackfor, L.S., J. Choi, A. Cleary, E. Dázevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK User´s Guide*, Society for Industrial and Applied Mathematics, 1997.

Bova, S., C. Breshears, R. Eingenmann, H. Gabb, G. Gaertner, B. Kuhn, B. Magro, S. Salvini, and V. Vatsa, "Combining message-passing and directives in parallel applications", *SIAM News*, 32:9 (1999), 1:10-11.

Brüngger, A., A. Marzetta, J. Clausen, and M.Perregaard. "Joining forces in solving large-scale quadratic assignment problems in parallel", *Proceedings of the 11th International Parallel Processing Symposium*, (1997), 418-427. (http://nobi.ethz.ch/group/sppiuk/progress98.html)

Butenhof, D. *Programming with POSIX threads*, Addison-Wesley, 1997.

Cantú-Paz, E. “Implementing fast and flexible parallel genetic algorithms”, In *Practical handbook of genetic algorithms* (L. D. Chambers, ed.), volume III, CRC Press, (1999), 65-84.

Carriero, N., D. Gelernter, and T. Mattson. “Linda in context”, *Communications of the ACM* 32 (1989), 444-458.

Carriero, N., D. Gelernter, and T. Mattson. “The Linda alternative to message-passing systems”, *Parallel Computing* 20 (1994), 633-458.

Chapman, B., F. Bodin, L. Hill, J. Merlin, G. Viland, and F. Wollenweber. “FITS – A light-weight integrated programming environment”, *Lecture Notes in Computer Science 1685* (1999), 125-134.

CPLEX. *Home page of CPLEX, a division of ILOG*, 1999. (http://www.cplex.com).

Dagum, L., *OpenMP: A proposed industry standard API for shared memory programming*, 1997. (http://www.openmp.org/index.cgi?resources+mp-documents/paper/paper.html)

Dagum, L., and R. Menon, “OpenMP: an industry standard API for shared memory programming”, *IEEE Computational Science & Engineering* 5 (1998), 46-55.

Davis, M., L. Liu, and J.G. Elias. “VLSI Circuit Synthesis Using a Parallel Genetic Algorithm”. *Proceedings of the IEEE’94 Conference on Evolutionary Computing*, (1994), 104-109.

Dempster, M.A.H, and R.T. Thompson, “Parallelization and aggregation of nested Benders decomposition”, *Annals of Operations Research* 81 (1997), 163-187. (http://www.jims.com.ac.uk/people/mahd.html)

DeRose, L., Y. Zhang, and D. A. Reed, “SvPablo: A multi-language performance analysis system”, *10th International Conference on Computer Performance Evaluation - Modelling techniques and tools*, (1998), 352-355.

Dijkstra, E.W. “Cooperating sequential processes”, in *Programming Languages* (F. Genvys, ed.), Academic Press (1968), 43-112.

Eckstein, J., W.E. Hart, and C.A. Philips. “An adaptable parallel toolbox for branching algorithms”, *XVI International Symposium on Mathematical Programming* (1997), Lausanne, p.82. (http://dmawww.epfl.ch/roso.mosiac/ismp97)

Elias, D. “Introduction to parallel programming concepts”, Workshop on Parallel Programming on the IBM SP, Cornell Theory Center, 1995. (http://www.tc.cornell.edu/Edu/Workshop)

Etnus Com. *Home page for Etnus Com.*, 1999. (http:www.etnus.com)

Fachat,,A., and K. H. Hoffman, “Implementation of ensemble-based simulated annealing with dynamic load balancing under MPI”, *Computer Physics Communications* 107 (1997), 49-53.

Feo, T.A., and M.G.C. Resende. “Greedy randomized adaptive search procedures”, *Journal of Global Optimization* 6 (1995), 109-133.

Foster, I. *Designing and building parallel programs: Concepts and tools for parallel software engineering*, Addison-Wesley, 1995.

Galarowicz, J., and B. Mohr. “Analyzing message passing programs on the Cray T3Ewith PAT and VAMPIR”, *Proceedings of fourth European CRAY-SGI MPP workshop* (H. Lederer and F. Hertwick, eds.), *IPP-Report des MPI für Plasmaphysik,IPP R/46*, 1998, 29-49. (http://www.kfa=juelich.de/zam/docs/autoren98/galarowicz.html).

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderman. *PVM: Parallel Virtual Machine - A user's guide and tutorial for networked parallel computing*, MIT Press, 1994. (http://www.netlib.org/pvm3/book/pvm-book.html)

Glover, F.. "Tabu Search - Part I", *ORSA Journal on Computing* 1 (1989), 190-206.

Glover, F.. "Tabu Search - Part II", *ORSA Journal on Computing* 2 (1990), 4-32.

Glover, F., and M. Laguna. *Tabu Search*, Kluwer, 1997.

Gropp, W., E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing-Interface*, MIT Press, 1994.

Gropp, W., S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference, Volume 2 - The MPI Extensions*, MIT Press, 1998.

Gupta, M., S. Midkiff, E. Schonberg, V.Seshadri, D. Shields, K.-Y.Wang, W.-M.Ching, and T. Ngo. "An HPF compiler for the IBM SP2", *Proceedings of the 1995 ACM/IEEE Supercomputing Conference* (1995). (http://www.supercomp.org/sc95/proceedings/417_SAMM/SC95.htm)

Harris, J., J. Bircsak, M. Bolduc, J. Diewald, I. Gale, N. Johnson, S. Lee, C.A. Nelson, and C. Offner., "Compiling high performance FORTRAN for distributed-memory systems", *Digital Technical Journal* 7 (1995), 5-38. (http://www.digital.com/info/DTJJ00)

High Performance FORTRAN Forum. High Performance FORTRAN language specification version 2.0, 1997. (http://www.crpc.rice.edu/HPFF/home.html)

Homer, S., "Design and performance of parallel and distributed approximation algorithms for maxcut", *Journal of Parallel and Distributed Computing* 41 (1997), 48-61.

IEEE (Institute of Electric and Electronic Engineering). *Information Technology - Portable Operating Systems Interface (POSIX) - Part 1 - Ammendment 2: Threads Extension*, 1995.

IBM. *Users Guide to Parallel OSL*, SC23-3824, 1999.

IBM. *Parallel ESSL version 2 release 1.2 Guide and Reference (SA22-7273)*, 1999. (http:/www.rs6000.ibm.com/resource/aix_resource/sp_books/essl)

Kliewer, G., and S. Tschöke. "A general parallel simulated annealing library (parSA) and its applications in industry", *PAREO'98: First meeting of the PAREO working group on parallel processing in operations research* (1998). (http://www.uni-paderborn.de/~parsa/)

Knies, A., M. O"Keefe, and T. MacDonald. "High Performance FORTRAN: A practical analysis", *Scientific Programming* 3 (1994), 187-199.

Koelbel, C., D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance FORTRAN Handbook*, The MIT Press, 1994.

Kuhn, B., P. Petersen, E. O´Toole, and M. Daly, "Using SMP parallelism with OpenMP", *CCP11 Newsletter Issue* 9 (1999). (http://www.hgmp.mrc.ac.uk/CCP11/newsletter)

Kumar, V., A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing design and analysis of parallel algorithms*, Benjaming/Cummings, 1994.

Lau, L.. *Implementation of scientific applications in a heterogeneous distributed network*, PhD. Thesis, University of Queensland, Department of Mathematics, 1996. (http//www.acmc.uq.edu.au/~ll/thesis)

Le Cun, B., C. Roucairol, Benaïchouche, M., V.-D. Cung, S. Dowaji, and T. Mautor. "BOB: A unified plataform for implementating branch-and-bound like algorithms", Technical Report RR-95/16, PRiSM Laboratory, Université de Versailles, 1995. (http:/www.prism.uvsq.fr/english/parallel/cr/bob_us.html)

Lengauer, C., M. Griebl, and S. Gorlatch (eds.). *Proceedings of the 3th International Euro-Par Conference* (Lecture Notes in Computer Science 1300) (1997), 89-165.

Linderoth, J., E.K. Lee, and M.W.P. Savelsbergh. "A parallel linear programming based heuristic for large scale set partitioning problems", Industrial & Systems Engineering Technical Report, 1999. (http://www.isye.gatech.edu/faculty/Eva_K_Lee)

Martins, S.L., M.G.C. Resende, C.C. Ribeiro, and P. Pardalos. "A parallel GRASP for the Steiner tree problem in graphs using a hybrid local search strategy", to appear in *Journal of Global Optimization*, 1999.

Martins, S.L., C.C. Ribeiro, and M.C. Souza. "A parallel GRASP for the Steiner problem in graphs", *Lecture Notes in Computer Science* 1457 (1998), 285-297.

Marzetta, A., A. Brüngger, K. Fukuda, and J. Nievergelt. "The parallel search bench ZRAM and its applications", *Annals of Operations Research 90* (1999), 45-63. (http://www.baltzer.nl/anor/contents/1999/90.html)

Merlin, J., S. Baden, S. Fink, and B. Chapman. "Multiple data parallelism with HPF and KeLP", *Journal of Future Generation Computer Systems* 15 (1999), 393-405.

Merlin, J., and A. Hey. "An introduction to High Performance FORTRAN", *Scientific Programming* 4 (1995), 87-113.

Miller, B.P., M.D. Callaghan, J.M. Cargille, J. K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. "The Paradyn parallel performance tools", *IEEE Computer 28* (1995), 37-46.

Morse, H.S. *Practical parallel computing*, AP Professional, 1994.

MPI Forum. "A Message-Passing Interface Standard", *The International Journal of Supercomputing Applications and High Performance Computing* 8 (1994), 138-161.

MPI Forum. "A Message-Passing Interface Standard", 1995. (http://www.mpi-forum.org/docs/docs.html)

Nagel, W., A. Arnold, M. Weber, H. Hoppe, and K. Solchenbach. "VAMPIR: visualization and analysis of MPI resources", *Supercomputer 63* (1996), 69-80.

Nieplocha, J., R. J. Harrison, and R. J. Littlefield. "Global arrays: A portable "shared-memory" programming model for distributed memory computers", *Proceedings of Supercomputing 94* (1994), 340-349. (http://www.tc.cornell.edu/UserDoc/Software/Ptools/global/)

OpenMP Architecture Review Board, *OpenMP FORTRAN Application Program Interf*ace, 1997.

OpenMP Architecture Review Board, *OpenMP C and C++ Application Program Interf*ace, 1998.

Plastino, A., C.C. Ribeiro, and N. Rodriguez. "A tool for SPMD application development with support for load balancing", to appear in *Proceedings of the ParCo Parallel Computing Conference* (1999). (http://www.inf.puc-rio.br/~celso).

Porto, S.C., J.P. Kitajima, and C.C. Ribeiro. "Performance evaluation of a parallel tabu search task scheduling algorithm", to appear in *Parallel Computing*, 1999.

Porto, S.C., and C.C. Ribeiro. “Parallel tabu search message-passing synchronous strategies for task scheduling under precedence constraints”, *Journal of Heuristics* 1 (1995), 207-223.

Porto, S.C., and C.C. Ribeiro. “A tabu search approach to task scheduling on heterogeneous processors under precedence constraints”, *International Journal of High Speed Computing* 7 (1995), 45-71.

Porto, S.C., and C.C. Ribeiro. “A case study on parallel synchronous implementations of tabu search based on neighborhood decomposition”, *Investigación Operativa* 5 (1996), 233-259.

Pritchard, D., and J. Reeve (eds.). *Proceedings of the 4th International Euro-Par Conference* (LNCS 1470), (1998), 80-189.

Protic, J., M. Tomasevic, and V. Milutinovic. *Distributed Shared Memory*, IEEE Computer Society, 1998.

Provenzano, C. “Pthreads”. (http://www.mit.edu:8001/people/proven/pthreads)

Reed, D.A., R. A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera. “Scalable performance analysis: The Pablo performance analysis environment”, *Proceedings of the Scalable Parallel Libraries Conference*, IEEE Computer Society (1993), 104-113.

Rivera-Gallego, W. “ A genetic algorithm for circulant euclidean distance matrices”, *Journal of Applied Mathematics and Computation* 97 (1998), 197-208.

Salhi, A., “Parallel implementation of a genetic-programming based tool for symbolic regression”, *Information Processing Letters* 66 (1998), 299-307.

SCAI-GMD. *Home page for Adaptor*, 1999. (http://www.gmd.de/SCAI/lab/adaptor)

Scientific Computing Associates. *Linda user’s guide & reference manual*, version 3.0, 1995.

Schuster, V. “PGHPF from the Protland Group”, *IEEE Parallel & Distributed Technologies*, (1994), p.72.

Taillard, E., M. Gendreau, F. Guertin, J.-Y. Potvin, and P. Badeau. “A parallel tabu search heuristic for the vehicle routing problem with time windows”, *Transportation Research-C* 5 (1997), 109-122.

Tanenbaum,A., *Modern Operating Systems*, Prentice-Hall, 1992.

Vahalia U. *Unix Internals - The New Frontiers*, Prentice-Hall, 1996.

Wolf, K., and O. Kraemer-Furhmann. “An integrated environment to design parallel object-oriented applications”, *Lecture Notes in Computer Science* 1123 (1996),120-127.

Yan, J., S. Sarukhai, and P. Mehra. “Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit”, *Software Practice and Experience* 25 (1995), 429-461.

Yan, J., and S. Sarukhai. “Analyzing parallel program performance using normalized performance indices and trace transformation techniques”, *Parallel Computing* 22 (1996), 1215-1237.

Zakarian, A.A.. *Nonlinear Jacobi and ε-relaxation methods for parallel network optimization.* PhD thesis, Mathematical Programming Technical Report 95-17, University of Wisconsin, Madison, 1995. (http://www.cs.wisc.edu/math-prog/tech-reports/)