TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Workshop on PVM, MPI, Tools, and Applications

Arndt Bode, Thomas Ludwig
Vaidy Sunderam, Roland Wismüller
(Herausgeber)

TUM-I9535
SFB-Bericht Nr.342/18/95 A
November 1995

TUM–INFO–11-95-I35-100/1.–FI

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

Workshop on PVM, MPI, Tools, and Applications

Arndt Bode, Thomas Ludwig
Vaidy Sunderam, Roland Wismüller
(Herausgeber)

Contents

Heterogeneous Concurrent Computing with PVM: Recent Developments and Future Trends *

V. S. Sunderam
Department of Math and Computer Science,
Emory University, Atlanta, GA 30322, U. S. A.

Abstract

Heterogeneous network-based distributed and parallel computing is gaining increasing acceptance as an alternative or complementary paradigm to multiprocessor-based parallel processing as well as to conventional supercomputing. While algorithmic and programming aspects of heterogeneous concurrent computing are similar to their parallel processing counterparts, system issues, partitioning and scheduling, and performance aspects are significantly different. In this paper, we discuss design and implementation issues in heterogeneous concurrent computing, in the context of the PVM system, a widely adopted software system for network computing. In particular, we highlight the system level infrastructures that are required, aspects of parallel algorithm development that most affect performance, system capabilities and limitations, and tools and methodologies for effective computing in heterogeneous networked environments. We present recent developments and experiences in the PVM project, and comment on ongoing and future work.

1 Introduction

We discuss parallel and distributed computing on networked heterogeneous environments. As used in this paper, these terms, as well as "concurrent" computing, refer to the simultaneous execution of the components of a single application on multiple processing elements. While this definition might also apply to most other notions of parallel processing, we make a deliberate distinction, to highlight certain attributes of the methodologies and systems discussed herein — namely *loose coupling*, physical and logical *independence* of the processing elements, and *heterogeneity*. These characteristics distinguish heterogeneous concurrent computing from traditional parallel processing, normally performed on homogeneous, tightly coupled platforms which possess some degree of physical independence but are logically coherent.

Concurrent computing, in various forms, is becoming increasingly popular as a methodology for many classes of applications, particularly those in the high-performance and scientific computing arenas. This is due to numerous benefits that accrue, both from the applications as well as the systems perspectives. However, in order to fully exploit these advantages, a substantial infrastructural framework is required — in the form of novel programming paradigms and models, systems support, toolkits, and performance analysis and enhancement mechanisms. In this paper, we focus on the latter aspects, namely the systems infrastructures, functionality, and performance issues in concurrent computing.

*Supported by NASA grant NAG 2-828, DoE grant DE-FG05-91ER25105, NSF grant ASC-9214149, and German Science Foundation grant SFB 342.

1.1 Heterogeneous, Networked, and Cluster Computing

One of the major goals of concurrent computing systems is to support heterogeneity. Heterogeneous computing refers to architectures, models, systems, and applications that comprise substantively different components, as well as to techniques and methodologies that address issues that arise when computing in heterogeneous environments. While this definition encompasses numerous systems, including reconfigurable architectures, mixed-mode arithmetic, special purpose hardware, and even vector and input-output units, we restrict ourselves to systems that are comprised of networked, independent, general-purpose computers that may be used in a coherent and unified manner. Thus, heterogeneous systems may consist of scalar, vector, parallel, and graphics machines that are interconnected by one or more (types of) networks, and support one or more programming environment/ operating system. In such environments, heterogeneity occurs in several forms:

- *System architecture* - heterogeneous systems may consist of SIMD, MIMD, scalar, and vector computers.
- *Machine architecture* - individual processing elements may differ in their instruction sets and/or data representation.
- *Machine configurations* - even when processing elements are architecturally identical, differences such as clock speeds and memory contribute to heterogeneity.
- *External influences* - as heterogeneous systems are normally built in general purpose environments, external resource demands can (and often do) induce heterogeneity into processing elements that are identical in architecture and configuration, and further, cause dynamic variations in interconnection network capacity.
- *Interconnection networks* - may be optical or electrical, local or wide-area, high or low speed, and may employ several different protocols.
- *Software* - from the infrastructure point of view, the underlying operating systems are often different in heterogeneous systems; from the applications point of view, in addition to operating systems aspects, different programming models, languages, and support libraries are available in heterogeneous systems.

Research in heterogeneous systems is in progress in several areas [1, 2] including applications, paradigm development, mapping, scheduling, reconfiguration, etc., but the primary thrust has thus far been in systems, methodologies, and toolkits. This latter thrust has been highly productive and successful, with several systems in production-level use at hundreds of installations worldwide. The body of this paper will discuss the PVM (Parallel Virtual Machine) system that has evolved into a popular and effective methodology for heterogeneous concurrent computing.

It is worthwhile to note at this juncture, that heterogeneous processing is a superset of similar methodologies referred to as network computing and cluster computing. While the nomenclature is as yet informal, network computing may be considered equivalent to heterogeneous computing, but with rather less emphasis on application heterogeneity, mapping, and task partitioning aspects. Cluster computing is even more restrictive, in that it generally refers to usually identical workstation clusters that are used as a substitute for hardware multiprocessors. These classes of frameworks are distinct from *distributed systems*, although several features and technologies are common to both. Distributed systems are distinguished by their goal of maximal transparency, as well as their emphasis on traditional operating systems issues such as filesystems, process management, kernel services, etc. Figure 1 depicts the relationship between various concurrent computing paradigms.

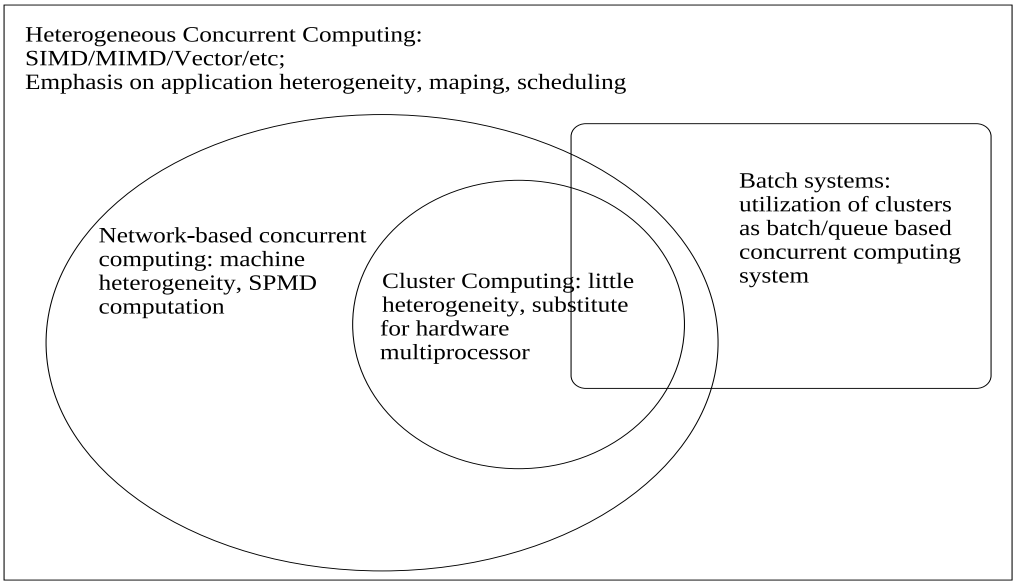


Figure 1: Heterogeneous, Network, and Cluster Computing

1.2 Applications Perspective

From the point of view of application development, heterogeneous computing is attractive, since it inherently supports function parallelism, with the added potential of executing subtasks on best-suited architectures. It is well known that different types of algorithms are well matched to different machine architectures and configurations, and at least in the abstract sense, heterogeneous computing permits this matching to be realized, resulting in optimality in application execution as well as in resource utilization. However, in practice, this scenario may be difficult to achieve for reasons of availability, applicability, and the existence of appropriate mapping and scheduling tools. Nevertheless, the concept is an attractive one and several research efforts are in progress in this area [3, 4].

In this respect, many classes of applications that would benefit substantively from heterogeneous computing have been identified. For example, a critically important problem which is ideally suited to heterogeneous computing is is global climate modeling. Simulation of the global climate is a particularly difficult challenge because of the wide range of time and space scales governing the behavior of the atmosphere, the oceans, and the surface. Parallel GCM codes require distinct component modules representing the atmosphere, ocean and surface and process modules representing phenomena like radiation and convection. Sampling, updating and manipulating this data requires scalar, vector, MIMD and SIMD paradigms, many of which can be performed concurrently. Another application domain that could exploit heterogeneous computing is computer vision. Vision problems generally require processing at 3 levels: high, medium and low. Low-level and some medium-level vision tasks often involve regular data flow and iconic operations. This type of computation is well-matched to mesh-connected SIMD machines. Medium-grained MIMD machines are more suitable for various high level and some medium level vision tasks which are communication-intensive and in which the flow of data is not regular. Coarse-grained MIMD machines are best matched for high-level vision tasks such as Image understanding/recognition and symbolic processing.

As previously mentioned however, the above aspect of heterogeneous concurrent computing is still in its infancy. Proof-of-concept research and experiments have demonstrated the viability of exploiting application heterogeneity, and many others are evolving. On the other hand, the systems aspect has matured significantly; to the extent that robust environments are now available for production execution of traditional parallel applications while providing stable testbeds for the evolving, truly heterogeneous, applications. We discuss the systems facet of heterogeneous concurrent computing in the remainder of the paper, with particular reference to the Parallel Virtual Machine (PVM) software infrastructure.

2 The PVM System

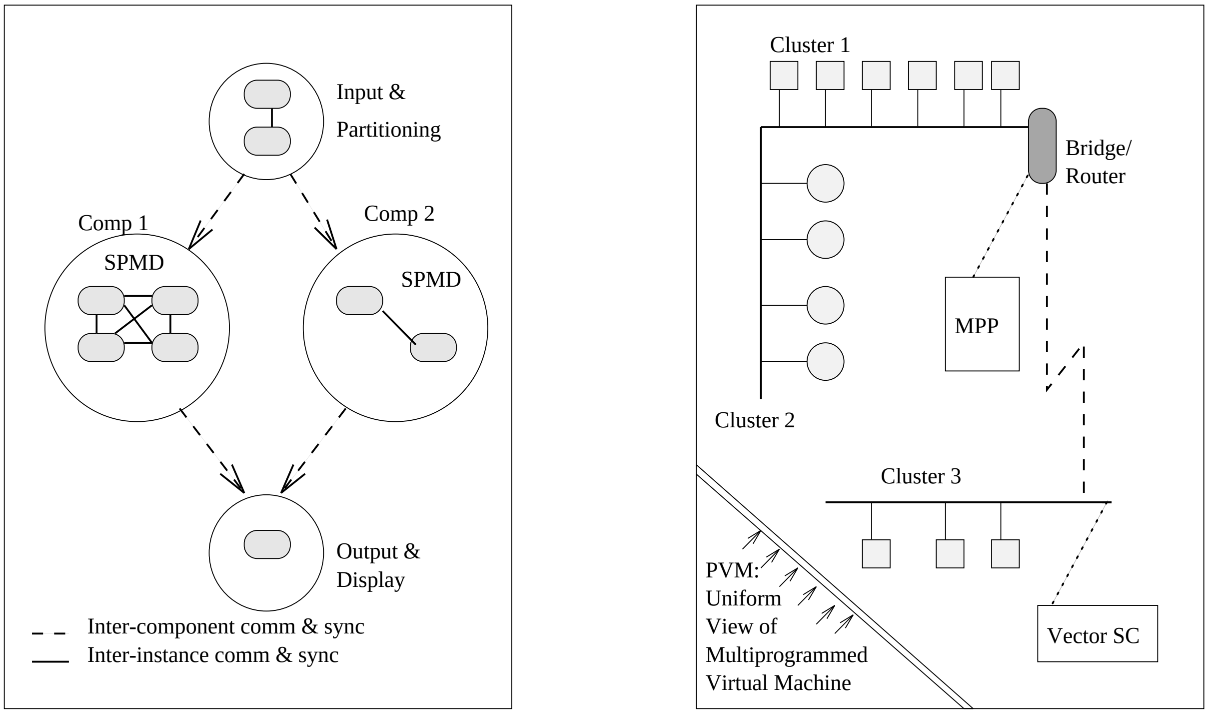
2.1 PVM Overview

PVM (Parallel Virtual Machine) is a software system that permits the utilization of a heterogeneous network of parallel and serial computers as a unified general and flexible concurrent computational resource. The PVM system [8] initially supported the message passing, shared memory, and hybrid paradigms, thus allowing applications to use the most appropriate computing model, for the entire application or for individual sub-algorithms. However, support for emulated shared-memory was omitted as the system evolved, since the message-passing paradigm was the model of choice for most scientific parallel processing applications. Processing elements in PVM may be scalar machines, distributed- and shared-memory multiprocessors, vector supercomputers and special purpose graphics engines, thereby permitting the use of the best suited computing resource for each component of an application.

The PVM system is composed of a suite of user-interface primitives supporting software that together enable concurrent computing on loosely coupled networks of processing elements. PVM may be implemented on a hardware base consisting of different machine architectures, including single CPU systems, vector machines, and multiprocessors. These computing elements may be interconnected by one or more networks, which may themselves be different (e.g. one implementation of PVM operates on Ethernet, the Internet, and a fiber optic network). These computing elements are accessed by applications via a standard interface that supports common concurrent processing paradigms in the form of well-defined primitives that are embedded in procedural host languages. Application programs are composed of *components* that are subtasks at a moderately large level of granularity. During execution, multiple *instances* of each component may be initiated. Figure 2 depicts a simplified architectural overview of the PVM computing model as well as the system.

Application programs view the PVM system as a general and flexible parallel computing resource. A translucent layering permits flexibility while retaining the ability to exploit particular strengths of individual machines on the network. The PVM user interface is strongly typed; support for operating in a heterogeneous environment is provided in the form of special constructs that selectively perform machine-dependent data conversions where necessary. Inter-instance communication constructs include those for the exchange of data structures as well as high-level primitives such as broadcast, barrier synchronization, mutual exclusion, and rendezvous. Application programs under PVM may possess arbitrary control and dependency structures. In other words, at any point in the execution of a concurrent application, the processes in existence may have arbitrary relationships between each other and, further, any process may communicate and/or synchronize with any other.

The PVM system is composed of two parts. The first part is a daemon, called *pvmd*, that executes on all the computers comprising the virtual machine. PVM is designed so that any user normal access rights to each host in the pool may install and operate the system. daemon on a machine. To run a PVM application, the user executes the daemons on a selected host pool, and the set of daemons cooperate via distributed algorithms to initialize the virtual machine. The PVM application can then be started by executing a program on any of these machine; the usual method is for this manually started program to spawn other application processes, using PVM facilities. Multiple users may configure overlapping virtual machines, and each user can execute several PVM applications simultaneously. The second part of the system is a library of PVM



(a) PVM Computation Model

(b) PVM Architectural Overview

Figure 2: PVM System Overview

interface routines (*libpvm.a*). This library contains user callable routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine.

2.2 Installing PVM

The installation process for PVM is straightforward. PVM does not require special privileges to be installed. Anyone with a valid login on the hosts that make up a virtual machine can do so. PVM uses two environment variables when starting and running. Each PVM user needs to set these two variables to use PVM. The first variable is PVM_ROOT, which is set to the location of the installed `pvm3` directory. The second variable is PVM_ARCH, which tells PVM the architecture of this host and thus what executables to select from the PVM_ROOT directory. The following simple steps complete the basic setup of PVM:

- Set PVM_ROOT and PVM_ARCH in the user's .cshrc file
- Build PVM for each architecture type
- Create .rhosts file on each host listing all the hosts
- Create `$HOME/.xpvm_hosts` file listing all the hosts prepended by an "&".

Configuring and testing the PVM system is the next logical step. On any host on which PVM has been installed the following command:

```
% pvm
```

should result in a PVM console prompt signifying that PVM is now running on this host. Hosts may then be added to this virtual machine by typing at the console prompt:

```
pvm> add hostname
```

One may also delete hosts (except the one logged into) from the virtual machine by typing:

```
pvm> delete hostname
```

The PVM console supports several interactive commands such as: `conf` that lists the VM configuration, `ps -a` to show the status of all executing tasks, `spawn` to start applications and `halt` to dismantle the virtual machine.

For situations where interactive configuration is not desired, there is a hostfile option. The user can list the hostnames in a file one per line and then type:

```
% pvm hostfile
```

PVM will then add all the listed hosts simultaneously before the console prompt appears. There are several options that can be specified on a per host basis in the hostfile. These are described at the end of this chapter for the user who wants to customize his virtual machine for a particular application or environment. There are also other ways to start up PVM. The functions of the console and a performance monitor have been combined in a graphical user interface called XPVM, which is also available as part of the distribution.

2.3 Fundamentals of PVM Programming

Developing applications for the PVM system follows, in a general sense at least, the traditional paradigm for programming distributed memory multiprocessors such as the nCUBE or the Intel family of hypercubes. This is true for both the logistical aspects of programming as well as for algorithm development. However, there are significant differences in terms of (a) task management, especially issues concerning dynamic process creation, naming and addressing; (b) initialization phases prior to actual computation; (c) granularity choices; and (d) heterogeneity. In this chapter, we present a discussion of the general programming process for PVM and identify factors that impact functionality and performance.

Parallel computing, using a system such as PVM, may be approached from three fundamental viewpoints, based upon the organization of the computing entities, i.e. the processes. Within each, different workload allocation strategies are possible, and will be discussed later in this chapter. The first, and most common model for PVM applications can be termed “crowd” computing — where a collection of closely related processes, typically executing the same code, perform computations on different portions of the workload, usually involving the periodic exchange of intermediate results. This paradigm can be further subdivided into two categories

- The master-slave (or host-node) model in which a separate “control” program termed the master is responsible for process spawning, initialization, collection and display of results, and perhaps timing functions. The slave programs perform the actual computation involved; they are either allocated their workloads by the master (statically or dynamically) or perform the allocations themselves.
- The node-only model where multiple instances of a single program execute, with one process (typically the one initiated manually) takes over the above non-computational responsibilities in addition to contributing to the computation itself.

The second model supported by PVM is termed a “tree” computation. In this scenario, processes are spawned (usually dynamically as the computation progresses) in a tree-like manner, thereby establishing a tree-like, parent-child relationship (as opposed to crowd computations where a star-like relationship exists). This paradigm, although less commonly used, is an extremely natural fit to applications where the total workload is not known *a priori*, e.g. in branch-and-bound algorithms, alpha-beta search, etc. as well as to recursive “divide-and-conquer” algorithms.

The third model, which we term "hybrid" can be thought of as a combination of the tree- and crowd- models. Essentially, this paradigm possesses an arbitrary spawning structure, i.e. at any point in time during application execution, the process relationship structure may resemble an arbitrary and changing graph. It should be noted that these classifications are made on the basis of process relationships though they frequently also correspond to communication topologies — nevertheless, in all three it is possible for any process to interact and synchronize with any other. Further, as may be expected, the choice of model is application dependent, and should be selected to best match the natural structure of the parallelized program.

2.4 Crowd computations

In crowd computations, there are typically three phases. The first is the initialization of the process group; in the case of node-only computations, dissemination of group information and problem parameters, as well as workload allocation is typically done within this phase. The second phase is computation. Finally, results are collected and displayed or output, and the process group is disbanded or terminated.

The master-slave model is illustrated below, using the well-known Mandelbrot set computation which is representative of the class of problems termed "embarrassingly" parallel. The computation itself involves applying a recursive function to a collection of points in the complex plane, until the function values either reach a specific value or begin to diverge. Depending upon this condition, a graphical representation of each point in the plane is constructed. Essentially, since the function outcome depends only on the starting value of the point (and is independent of other points), the problem can be partitioned into completely independent portions, the algorithm applied to each, and partial results combined using simple combination schemes. However, this model permits dynamic load balancing, thereby permitting processing elements to share the workload unevenly. In this and subsequent examples within this chapter, we only show a skeletal form of the algorithms, and also take syntactic liberties with the PVM routines in the interest of clarity. The control structure of the master-slave class of applications is shown in Figure 1.

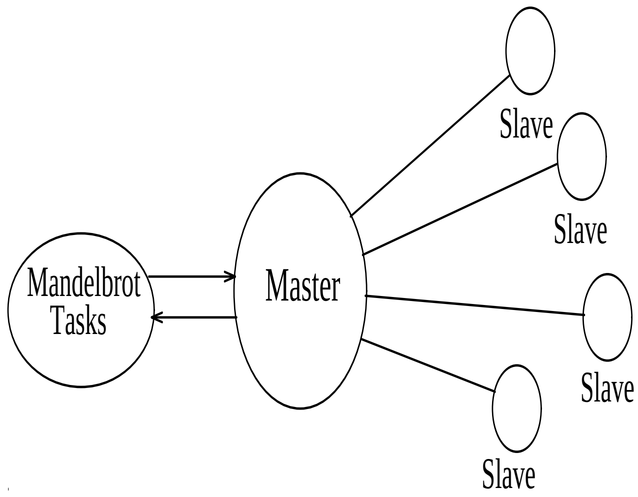


Figure 1: Master-Slave Paradigm

```
{Master Mandelbrot algorithm.}

{Initial placement}
```

```
for i := 0 to NumWorkers - 1
pvm_spawn(<worker name>)    {Start up worker i}
pvm_send(<worker tid>,999)  {Send task to worker i}
endfor

{Receive-send}
while (WorkToDo)
pvm_recv(888)         {Receive result}

pvm_send(<available worker tid>,999)
{Send next task to available worker}

display result
endwhile

{Gather remaining results.}
for i := 0 to NumWorkers - 1
pvm_recv(888)                {Receive result}
pvm_kill(<worker tid i>)     {Terminate worker i}
display result
endfor

----------------------------------------------------------------------

{Worker Mandelbrot algorithm.}

while (true)
pvm_recv(999)                          {Receive task}
result := MandelbrotCalculations(task) {Compute result}
pvm_send(<master tid>,888)     {Send result to master}
endwhile
```

The master-slave example described above involves no communication among the slaves. Most crowd computations of any complexity do need to communicate among the computational processes; we illustrate the structure of such applications using a node-only example for matrix multiply using Cannon's algorithm (programming details for a similar algorithm are given in a future chapter). The matrix multiply example, shown pictorially in Figure 2, multiplies matrix subblocks locally, and uses row-wise multicast of matrix `A` subblocks in conjunction with column-wise shifts of matrix `B` subblocks.

```
{Matrix Multiplication using Pipe-Multiply-Roll algorithm.}

{Processor 0 starts up other processes}
if (<my processor number> = 0) then
    for i := 1 to MeshDimension*MeshDimension
        pvm_spawn(<component name>, . .)
    endfor
endif

forall processors Pij, 0 <= i,j < MeshDimension
    for k := 0 to MeshDimension-1
```

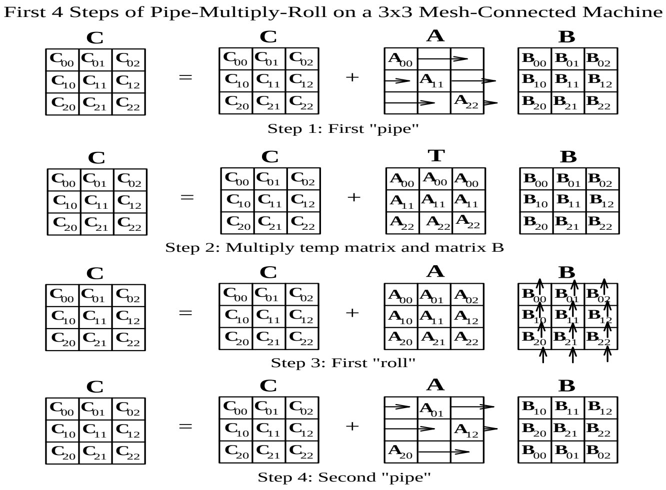


Figure 2: General Crowd Computation

```
        {Pipe.}
        if myrow = (mycolumn+k) mod MeshDimension
            {Send A to all Pxy, x = myrow, y <> mycolumn}
            pvm_mcast((Pxy, x = myrow, y <> mycolumn),999)
        else
            pvm_recv(999)   {Receive A}
        endif

        {Multiply.  Running totals maintained in C.}
        Multiply(A,B,C)

        {Roll.}
        {Send B to Pxy, x = myrow-1, y = mycolumn}
        pvm_send((Pxy, x = myrow-1, y = mycolumn),888)
        pvm_recv(888)       {Receive B}
    endfor
endfor
```

2.5 Tree computations

As mentioned earlier, tree computations typically exhibit a tree-like process control structure that also conforms to the communication pattern in many instances. To illustrate this model, consider a parallel sorting algorithm that works as follows. One process (the manually started process in PVM) possesses (inputs or generates) the list to be sorted. It then spawns a second process and sends it half the list. At this point, there are two processes each of which spawns a process and sends them one-half of their already halved lists. This continues until a tree of appropriate depth is constructed. Each process then independently sorts its portion of the list, and a merge phase follows where sorted sublists are transmitted upwards along the tree edges, with intermediate merges being done at each node. This algorithm is illustrative of a tree computation in which the workload is known in advance; a diagram depicting the process and an algorithmic outline are given below.

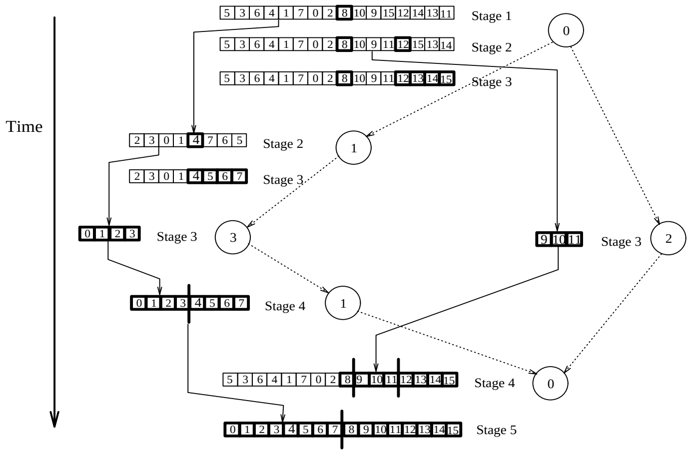


Figure 3: Tree-computation example

```
{ Spawn and partition list based on a broadcast tree pattern. }
for i := 1 to N, such that 2^N = NumProcs
forall processors P such that P < 2^i
pvm_spawn(...) {process id P XOR 2^i}
if P < 2^(i-1) then
midpt: = PartitionList(list);
{Send list[0..midpt] to P XOR 2^i}
pvm_send((P XOR 2^i),999)
list := list[midpt+1..MAXSIZE]
else
pvm_recv(999)    {receive the list}
endif
endfor
endfor

{ Sort remaining list. }
Quicksort(list[midpt+1..MAXSIZE])

{ Gather/merge sorted sub-lists. }
for i := N downto 1, such that 2^N = NumProcs
forall processors P such that P < 2^i
if P > 2^(i-1) then
pvm_send((P XOR 2^i),888)
{Send list to P XOR 2^i}
else
pvm_recv(888)    {receive temp list}
merge templist into list
endif
endfor
endfor
```

3 Workload Allocation

In the previous section, we discussed the common parallel programming paradigms with respect to process structure, and outlined representative examples in the context of the PVM system. In this section we address the issue of workload allocation, subsequent to establishing process structure, and describe some common paradigms that are used in distributed memory parallel computing. Two general methodologies are commonly used. The first, termed data decomposition or partitioning, assumes that the overall problem involves applying computational operations or transformations on one or more data structures, and further, that these data structures may be divided and operated upon. The second, called function decomposition, divides the work based on different operations or functions. In a sense, the PVM computing model supports function decomposition at the `component` level (components are fundamentally different programs that perform different operations) and data decomposition at the instance level i.e., within a component, the same program operates on different portions of the data.

3.1 Data Decomposition

As a simple example of data decomposition, consider the addition of two vectors, A[1..N] and B[1..N], to produce the result vector C[1..N]. Assuming P processes working on this problem, data partitioning involves the allocation of N/P elements of each vector to each process, that computes the corresponding N/P elements of the resulting vector. This data partitioning may be done either "statically" i.e. where each process knows *a priori* (at least in terms of the variables N and P) its share of the workload. It may also be done "dynamically", where a control process (e.g. the master process) allocates subunits of the workload to processes as and when they become free. The principal difference between these two approaches is with regard to a related concept, that of "scheduling". With static scheduling, individual process workloads are fixed; with dynamic scheduling, they vary as the computation progresses. In most multiprocessor environments, static scheduling is effective for problems such as the above vector addition example; however, in the general PVM environment, this is not necessarily true. The reason is that PVM environments based on networked clusters are susceptible to external influences; therefore, a statically scheduled, data partitioned problem might encounter one or more processes that complete their portion of the workload much faster or much slower than the others. This situation could also arise when the machines in a PVM system are heterogeneous, possessing varying CPU speeds and different memory and other system attributes.

In a real execution of even this trivial vector addition problem, an issue that cannot be ignored is input and output. In other words, how do the processes described above receive their workloads, and what do they do with the result vectors? The answer to these questions is dependent on the application and the circumstances of a particular run, but in general:

1 Individual processes generate their own data internally e.g. using random numbers or statically known values. This is only possible in very special situations or for program testing purposes.

2 Individual processes independently input their data subsets from external devices. This method is meaningful in many cases, but only possible when parallel I/O facilities are supported.

3 A controlling process sends individual data subsets to each process. This is the most common scenario, especially when parallel I/O facilities do not exist. Further, this method is also appropriate when input data subsets are derived from a previous computation within the same application

The third method of allocating individual workloads is also consistent with dynamic scheduling in applications where interprocess interactions during computations are rare or nonexistent. However, non-trivial algorithms generally require intermediate exchanges of data values, and therefore, it is only the initial assignment of data partitions that can be accomplished via these schemes. For example, consider the data partitioning method depicted in Figure 4.2. In order to multiply two matrices A and B, a group of processes is first spawned, using the master-slave or node-only paradigm. This set of processes is considered to form a mesh; the matrices to be multiplied are divided into subblocks, also forming a mesh. Each subblock of the A and B matrices is placed on the corresponding process, by utilizing one of the data decomposition and workload allocation strategies listed above. During computation, subblocks need to be forwarded or exchanged between processes, thereby transforming the original allocation map, as shown in the figure. At the end of the computation however, result matrix subblocks are situated on the individual processes, in conformance with their respective positions on the process grid, and consistent with a data partitioned map of the resulting matrix C. The foregoing discussion illustrates the basics of data decomposition. In a later chapter, example programs highlighting details of this approach will be presented.

3.2 Function Decomposition

Parallelism in distributed memory environments such as PVM may also be achieved by partitioning the overall workload in terms of different operations. The most obvious example of this form of decomposition is with respect to the three stages of typical program execution, namely input, processing, and result output. In function decomposition, such an application may consist of three separate and distinct programs, each one dedicated to one of the three phases. Parallelism is obtained by concurrently executing the three programs and by establishing a "pipeline" (continuous or quantized) between them. Note however, that in such a scenario, data parallelism may also exist within each phase. An example is shown in Figure 2.1 where distinct functions are realized as PVM components, with multiple instances within each component implementing portions of different data partitioned algorithms.

Although the concept of function decomposition is illustrated by the trivial example above, the term is generally used to signify partitioning and workload allocation by function *within* the computational phase. Typically, application computations contain several different subalgorithms — sometimes on the same data (the MPSD or multiple program/single data scenario), sometimes in a pipelined sequence of transformations, and sometimes exhibiting an unstructured pattern of exchanges. We illustrate the general functional decomposition paradigm by considering the hypothetical simulation of an aircraft consisting of multiple interrelated and interacting, functionally decomposed, subalgorithms. A diagram providing an overview of this example is shown in Figure 4, and will also be used in a later chapter dealing with graphical PVM programming.

In the figure, each node or circle in the "graph" represents a functionally decomposed piece of the application. The input function distributes the particular problem parameters to the different functions *2* through *6*, after spawning processes corresponding to distinct programs implementing each of the application subalgorithms. The same data may be sent to multiple functions (e.g. as in the case of the two *wing* functions, or data appropriate for the given function alone may be delivered. After performing some amount of computations these functions deliver intermediate or final results to functions *7, 8, 9* that may have been spawned at the beginning of the computation or as results become available. The diagram indicates the primary concept of decomposing applications by function, as well as control and data dependency relationships. Parallelism is achieved in two respects — by the concurrent and independent execution of modules as in functions *2* through *6*, and by the simultaneous, pipelined, execution of modules in a dependency chain, as for example, in functions *1, 6, 8, 9*.

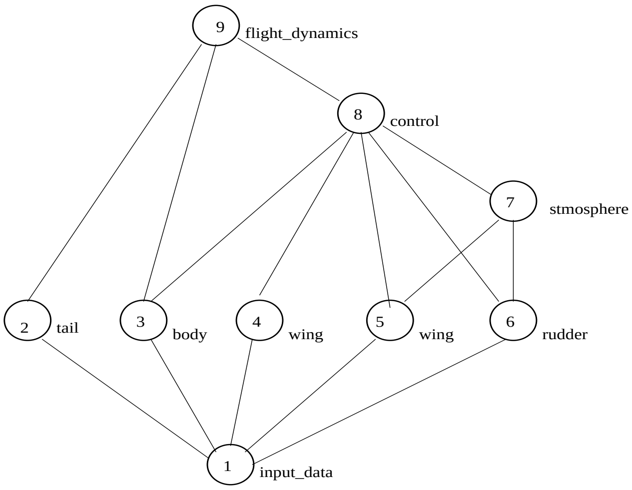


Figure 4: Function decomposition example

4 Porting Existing Applications to PVM

In order to utilize the PVM system, applications must evolve through two stages. The first concerns development of the distributed memory parallel version of the application algorithm(s); this phase is common to the PVM system as well as to other distributed memory multiprocessors. The actual parallelization decisions fall into two major categories — those related to structure, and those related to efficiency. For structural decisions in parallelizing applications, the major decisions to be made include the choice of model to be used i.e. crowd computation vs. tree computation and data decomposition vs. function decomposition. Decisions with respect to efficiency when parallelizing for distributed memory environments are generally oriented towards minimizing the frequency and volume of communications. It is typically in this latter respect that the parallelization process differs for PVM and hardware multiprocessors; for PVM environments based on networks, large granularity generally leads to better performance. With this qualification, the parallelization process is very similar for PVM and for other distributed memory environments, including hardware multiprocessors.

The parallelization of applications may be done either *ab initio* or from existing sequential versions or from existing parallel versions. In the first two cases, the stages involved are to select an appropriate algorithm for each of the subtasks in the application, usually from published descriptions — or by inventing a parallel algorithm. These algorithms are then coded in the language of choice (C, C++, or Fortran77 for PVM) and interfaced with each other as well as with process management and other constructs. Parallelization from existing sequential programs also follows certain general guidelines, primary among which are to decompose loops, beginning with outermost loops and working inward. In this process, the main concern is to detect dependencies and partition loops such that dependencies are preserved while allowing for concurrency. This parallelization process is described in numerous textbooks and papers on parallel computing, although few textbooks discuss the practical and specific aspects of transforming a sequential program to a parallel one.

Existing parallel programs may be based upon either the shared memory or distributed memory paradigms. Converting existing shared memory programs to PVM is similar to converting from sequential code, when the shared memory versions are based upon vector or loop-level parallelism. In the case of explicit shared memory programs, the primary task is to locate synchronization points and replace these with message passing. In order to convert existing distributed memory parallel code to PVM, the main task is to convert from one set of concurrency constructs to another. Typically, existing distributed memory parallel programs are written either for hardware multiprocessors or other networked environments such as P4 or Express. In both cases, the major changes required are with regard to process management. For example, in the Intel family of DMMP's, it is common for processes to be started from an interactive shell command line. Such a paradigm should be replaced for PVM by either a master program or a node program that takes responsibility for process spawning. With regard to interaction, there is, fortunately, a great deal of commonality between the message passing calls in various programming environments. The major differences between PVM and other systems in this context are with regard to (a) process management and process addressing schemes; (b) virtual machine configuration/reconfiguration and its impact on executing applications; (c) heterogeneity in messages as well as the aspect of heterogeneity that deals with different architectures and data representations; and (d) certain unique and specialized features such as signaling, task scheduling methods, etc.

5 PVM Performance in Cluster Environments

With the ever-increasing adoption of PVM (and systems like it) for high performance concurrent computing, the issue of performance assumes great significance. In order to experimentally evaluate the efficacy and viability of PVM, on typical networked environments i.e. clusters of workstations, we have undertaken a performance measurement and system enhancement exercise, which is described in this section. This involves the implementation and execution of a set of CFD benchmark applications on virtual parallel machines using the PVM software system. These applications are the five "kernel" benchmarks from the NAS Parallel Benchmark suite [25], an algorithmically specified collection of programs that are representative of real codes (or portions thereof) that are in production use in the aerospace community. Despite being termed kernels, the five applications rigorously exercise the processor, memory and, in the case of distributed-memory parallel machines, the communications capacity of any given system.

5.1 The NAS Parallel Benchmarks

The NAS Parallel Benchmarks refer to a suite of applications devised by the Numerical Aerodynamic Simulation (NAS) Program of the National Air and Space Administration (NASA) for the performance analysis of highly parallel computers. While the NPB suite is rooted in the problems of computational fluid dynamics and computational aerosciences, they are valuable in the evolution of parallel computing, since they are rigorous and as close to "real" applications as may be reasonably expected from a benchmarking suite. The NPB consist of five "kernels" and three simulated applications which "mimic the computation and data movement characteristics of large scale computational fluid dynamics (CFD) applications" [24]. These benchmarks are specified only algorithmically, thereby encouraging optimization and refinement, at the cost of necessitating substantial expertise and effort in porting to a new platform. Complete details of the NPB suite may be found in [24, 25]; for the sake of completeness, we outline the five kernels that were ported to PVM and used in the experiments reported in this paper.

- Kernel EP is to execute 2^{28} iterations of a loop in which a pair of random numbers are generated and tested for whether Gaussian random deviates can be made from them according to a specific scheme. This kernel falls into the category of applications termed "embarrassingly parallel".
- Kernel MG is to execute four iterations of the V-cycle multigrid algorithm to obtain an approximate solution to the discrete Poisson problem $\nabla^2 u = v$ on a $256 \times 256 \times 256$ grid with periodic boundary conditions. This application rigorously exercises both short- and long-distance communication.
- Kernel CG is to use the power and conjugate gradient methods to approximate the smallest eigenvalue of a symmetric, positive definite, sparse matrix of order 14000 with a random

pattern of nonzeros. The communication patterns in this kernel are long-distance and unstructured.

- Kernel FT uses FFT's on a $256 \times 256 \times 128$ complex array to solve a 3-dimensional partial differential equation. Communication patterns in this kernel are structured and long distance.

- Kernel IS is to perform 10 rankings of 2^{23} (8388608) integer keys in the range [0, 2^{19} (524288)). Communication in this benchmark is frequent and relatively low-volume.

5.2 PVM Implementation of the NPB Kernels

The parallelization of the five NPB kernels for the PVM system was based on a set of codes originally written for the Intel iPSC hypercube. These codes provided the basic partitioning and parallelization algorithms; substantial modifications to these code were required to convert them for the PVM system. Details of the parallelization as well as algorithm specifics pertaining to the applications may be found in [26]. In this section we present preliminary results of our performance experiments with these five kernels in three different PVM environments. The release version (3.2) of the PVM software, was used, and the default "daemon-based" communication scheme was selected. In the next section we describe a significant performance enhancement to the PVM communications scheme[1], and present updated results. In order to obtain an understanding of the effect of different hardware platforms while utilizing the same software system, we conducted our benchmark experiments on three environments, each with its unique characteristics.
Low-end workstations on low-speed shared medium networks: This platform is made up of 16 diskless Sun SS1+ workstations, each with 16 MB of memory and 32 MB of virtual memory(swap space), at the Emory University Parallel Processing Lab.
Medium-power workstations on high-speed shared networks: This environment is made up of seven IBM RS/6000 model 560 computers, and one RS/6000 model 320 computer, each with 32 MB of memory, and 64 MB of swap space, connected by FDDI.
Medium-power workstations on high-speed switched medium networks: This platform consisted of SGI R4000 workstations interconnected by FDDI Gigaswitch.

5.3 Preliminary Performance Results

In this subsection we present consolidated performance results for the five kernel benchmarks. Table 1 shows the the total *elapsed time* in seconds for each platform, the total communication volume, and the time for communication related activities alone for each benchmark on the three testbed platforms, using an unmodified version of PVM version 3.2.

A few observations are in order, regarding the measurements in table 1. The first is that generally, clusters of about ten workstations perform within an order of magnitude of a Cray YMP-1. The second is that communication times account for a large fraction of the overall execution time in many cases. Finally, communication efficiency, i.e. the ratio of obtained throughput to that theoretically possible in each network, appears low.

5.4 Enhanced Performance in PVM

To address some of the performance shortcomings described above, we devised an alternative communication scheme for PVM, and re-implemented the NAS benchmarks. The enhanced message passing scheme, accessible via the `pvm_fsend()` and `pvm_frecv()` calls, permit the direct transfer

[1] Since the time of this exercise, the release version of PVM (3.3.3) has incorporated the fast communication scheme.

Bench-mark	16 SS1+ Enet			8 RS6000 FDDI			8 SGI Gswitch			Cray Y/MP-1	i860 128
	Time (secs)	Com. (MB)	Vol. time	Time (secs)	Vol. (MB)	Com. time	Time (secs)	Vol. (MB)	Com. time	Time (secs)	Time (secs)
EP	1603	NA	NA	342	NA	NA	446	NA	NA	126	26
MG	198[1]	96	154	229	192	162	264	192	112	22	8.6
CG	701	370	480	285*	130	192	130+	250	101	12	8.6
IS	607[2]	150	595	674	560	610	770	560	720	17	14
FT	717[1]	420	502	645	1500	395	1070	1500	516	29	10

[1] Reduced problem size ($128 \times 128 \times 128$); [2] Reduced problem size (2^{21} keys, range 0 to 2^{19})

* 4 RS 6000 processors; + 9 SGI R4000 processors

Table 1: NPB Kernels on unmodified PVM

of user program data without requiring buffer initialization and packing. The `pvm_fsend()` and `pvm_frecv()` mechanisms are built on standard TCP internet protocols and are manifested as a separate and non-intrusive library in the PVM system. The `pvm_fsend()` and `pvm_frecv()` library was implemented and tested on a variety of environments and networks. Table 2 indicates the performance of this communication scheme for simple point-to-point data transfer, for a variety of message sizes, for each environment discussed in this paper. Also shown for reference, are the corresponding values for daemon-based PVM communication, and for a standalone benchmarking program, *viz.* TTCP.

Platform	Throughput (kB/sec)			
Msg. size→	1 byte	100 bytes	10kB	1MB
SS1+ Ethernet				
Daemon	0.06	12.88	263.41	358.48
Fsend	0.49	81.79	902.42	1003.87
TTCP	0.65	130.45	965.04	1125.24
RS6000 FDDI				
Daemon	0.09	20.67	374.04	711.58
Fsend	0.82	112.79	1568.40	2285.89
TTCP	1.85	325.40	2573.00	2918.20
SGI Gigaswitch				
Daemon	0.21	38.92	406.04	483.58
Fsend	1.11	102.79	3400.42	9550.87
TTCP	2.15	500.40	9203.00	9624.20

Table 2: Point-to-point communication bandwidth in PVM

These improvements were also carried over to the applications, as indicated by the representative tables shown below — 3 and 4. In the tables, we also include the total time and communication time for the previous experiments (in parentheses) for convenient comparison. We also include the number of messages, and an additional measure, *viz* the "maximum idle time".

These revised results demonstrate that performance levels close to theoretical capacities can be obtained in cluster environments while using PVM. However, certain factors in the results also indicate that dynamic load balancing schemes and better partitioning methods are probably required to exploit the full potential of network computing.

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs	Idle Time(secs)
16 SS1+ Enet	138* (198*)	96 MB	85 (154)	2704	48
8 RS6000 FDDI	110 (229)	192 MB	52 (162)	1808	30
8 SGI Gswitch	168 (264)	192 MB	81 (112)	1808	50
Cray Y-MP/1: 22 secs; i860/128 : 8.6 secs					

* Reduced problem size ($128 \times 128 \times 128$)

Table 3: Kernel MG on enhanced PVM

Platform	Time (secs)	Comm. Volume	Comm. Time(secs)	Number of msgs	Idle Time(secs)
16 SS1+ Enet	605 (701)	370 MB	404 (480)	37920	390
4 RS6000 FDDI	203 (285)	130 MB	101 (192)	7116	88
9 SGI Gswitch	108 (130)	250 MB	46 (101)	19756	42
Cray Y-MP/1: 12 secs; i860/128: 8.6 secs					

Table 4: Kernel CG on enhanced PVM

6 Generalized Distributed Computing and Parallel I/O with PVM

In the evolution of the PVM system for heterogeneous distributed computing, high-performance scientific applications have thus far been the main technical drivers. The computing model, as well as specific software features have been influenced by the requirements of scientific algorithms and their parallel implementations. We believe that by extending this infrastructure along certain important dimensions, systems such as PVM will be able to cater to a much larger class of application categories. The goal therefore is to enable generalized distributed computing within distributed and networked environments, i.e. to evolve both a conceptual model and a software infrastructure that integrally support high performance applications as well as other general purpose applications, including, but not limited to, distributed teleconferencing and groupware systems, heterogeneous and multi-databases, high speed on-line transaction processing and geographically distributed information systems.

The basic infrastructural requirements for supporting general purpose distributed computing in cluster environments include:

- facilities for parallel input and output, including the ability for multiple processes to simultaneously access files as well as for individual files to be distributed.
- concurrency control mechanisms at various levels, including the ability for multiple processes to synchronize and perform distributed mutual exclusion, for atomic broadcast and multicast, and preservation of delivery order.
- fault tolerance and data replication facilities that permit continued (but degraded) execution and protect against computation or data loss.
- support for a client/server model of computing, including mechanisms for service naming and lookup.
- transaction processing support.

6.1 The GDC Architecture

The proposed model for generalized distributed computing in PVM (termed the GDC layer) is an extension of, and consistent with, the existing model. The concept of "sessions" is central to this architecture — the basic notion being that processes from inter-related computations are dynamically created and destroyed as necessary, join sessions in order to contribute their portion of the computation, and primarily interact with other session members although cross-session interaction is also possible. Sessions are also the computational units in which computation and interaction semantics are established; e.g. a parallel program session might require process placement on the fastest CPU's and reliable communication, whereas a database access session must locate servers on specific machines and provide transaction oriented data exchange semantics. In the PVM GDC layer, library primitives are provided for "checking-into" (and "out of") sessions, at which time access control functions are also established. Processes enrolled in specific sessions may cooperate via normal PVM message passing mechanisms, but in addition, may achieve concurrency control by invoking special primitives for locking resources, entering transaction modes of operation, and can avail of failure resilience facilities based either on shadowing or on checkpoints. The GDC architecture also supports a parallel input-output framework as well as the client-server mode of distributed computing, details of which are presented in the next subsection. This enhancement layer to PVM is currently under alpha test and will be incorporated into the release version of the software in the near future. An overview of the proposed GDC architectural model is shown in figure 5.

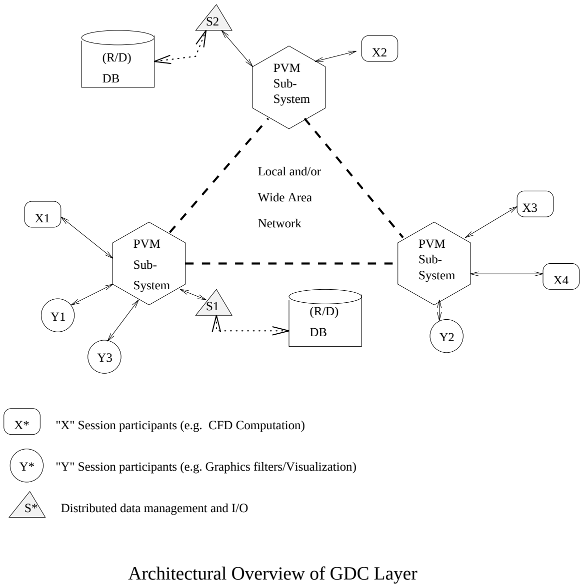


Figure 5: GDC Architectural Overview

6.2 The PIOUS Parallel I/O Subsystem

Central to the GDC layer of PVM is the parallel I/O subsystem, termed PIOUS. The PIOUS module or layer is motivated by the fact that most production level applications require infrastructural support for high performance input output, with mechanisms for concurrency control and failure resilience. PIOUS is an input/output system that provides process groups access to permanent storage within a heterogeneous network computing environment. PIOUS is a parallel distributed file server that achieves a high-level of performance by exploiting the combined file I/O and buffer cache capacities of multiple PVM-interconnected computer systems. Fault tolerance is achieved by exploiting the redundancy of storage media.

To better support parallel applications, PIOUS implements a parallel access file object called a *parafile* and provides varying levels of concurrency control for process group members. For portability, PIOUS allows parafile objects to be accessed with standard Unix semantics i.e. processes open, close, read and write parafiles as with sequential files. PIOUS is itself implemented as a group of cooperating processes within the GDC distributed computing framework. Parafiles are logically single files composed of one or more disjoint segments. Each segment is composed of zero or more records; a record is the base unit of information stored in a parafile. The PIOUS interface provides a process group with three *views* of a parafile object: global, independent, and segmented. Parafiles are two-dimensional file objects. Thus mapping arrays of data is simplified. A parallel computation can easily access rows or columns of a matrix by accessing individual segments of the parafile or by accessing the parafile as a linear sequence, as appropriate; block access is also possible via higher-level libraries. A schematic of the PIOUS subsystem is shown in Figure 6.

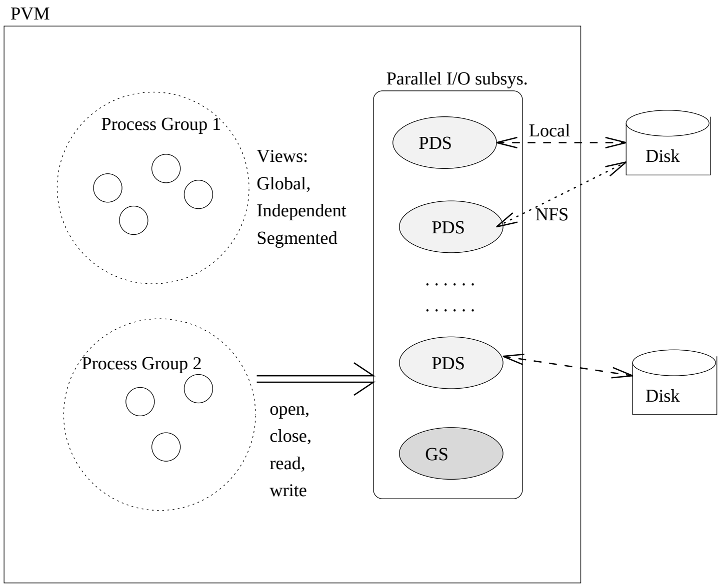


Figure 6: PIOUS Architectural Overview

PIOUS consists of a set of data servers, a service coordinator, and library routines linked with client processes. An underlying transport mechanism (i.e. PVM) is assumed to carry messages between client processes and components of the PIOUS architecture. PIOUS data servers are assumed to access permanent storage via a native file system, and reside on on each machine over which a file to be accessed is declustered. A preliminary prototype of the PIOUS system has been implemented, and has undergone initial performance and functionality testing. Our experiences indicate that the facilities provided by PIOUS are extremely valuable for concurrent applications requiring high performance parallel file access and a distributed database system is presently being built to verify the usefulness of the parallel I/O system for real applications. Performance results from the prototype implementation indicate that, for modest size data transfers, the overhead introduced by the PIOUS software is minimal, and is comparable to levels attained by other (non parallel) network file systems. A more detailed description of the PIOUS parallel I/O system may be found in [27].

7 A Threads-Based Concurrent Computing Model

In this research initiative, an alternative concurrent computing paradigm, based on "services", supporting data driven computation, and built on a lightweight process infrastructure, is proposed to enhance the functional capabilities and the operational efficiency of heterogeneous network-based concurrent computing. This approach, which is derived by combining variants of principles from multithreading systems, data flow computing, and remote procedure call, is believed to have the potential to enhance performance and functionality in heterogeneous systems, without too drastic a departure from the prevalent parallel programming methodologies.

7.1 The TPVM Framework

TPVM is a collection of extensions and enhancements to the PVM computing model and system. In TPVM, computational entities are threads. TPVM threads are runtime manifestations of imperative subroutines or collections thereof that cooperate via a few simple extensions to the PVM programming interface. In the interest of straightforward transition and to avoid a large paradigm shift, one of the modes of use in TPVM is identical to the process-based model in PVM, except that threads are the computational units. TPVM also offers two other programming models — one based on data driven execution, and the other supporting remote memory access. The individual models supported by TPVM are discussed in later sections; a general architectural overview of TPVM is depicted in Figure 7, and a brief description follows.

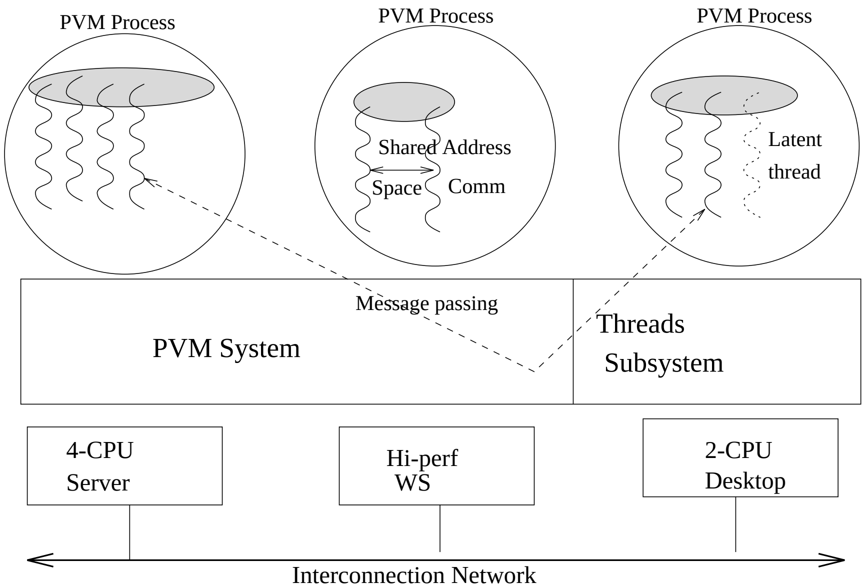


Figure 7: TPVM Architectural Overview

Architecturally, TPVM is a natural extension of the PVM model. In terms of the resource platform, TPVM also emulates a general-purpose heterogeneous concurrent computer on an interconnected collection of independent machines. However, since TPVM supports a threads-based model, it is potentially capable of exploiting the benefits of multithreaded operating systems as well as small-scale SMMP's — both of which are becoming increasingly prevalent in general purpose computing environments. Further, multiple computational entities may now be manifested within a single process. In combination, these aspects enable increased potential for optimizing interaction between computational units in a user-transparent manner. In other words, inter-machine communication may continue to use message passing, while intra-machine communication, including that within SMMP's, may be implemented using the available global address space. In addition, "latent" computational entities in the form of dormant threads may be instantiated either asynchronously or during initialization, at negligible or low cost. In many cases, this helps reduce the overhead of spawning new computational entities during application execution. Further, the concept of latent threads extends naturally to service-based computing paradigms that are more appropriate in general purpose, non-scientific, distributed and concurrent processing.

TPVM is designed to be layered over the PVM system, and does not require any modifications or changes to PVM. User level primitives are supplied as a library against which application programs link; operational mechanisms are provided in the form of standalone PVM programs. Central to the TPVM implementation is the concept of a scheduling interface that is responsible for controlling thread spawning as well as other facilities that are available in TPVM. Thus interface is defined in functional terms, thereby enabling implementations to evolve from a centralized mechanism in preliminary implementations to one based on distributed algorithms. A schematic of this the scheduling interface and the principal facets of a TPVM implementation are shown in Figure 8.

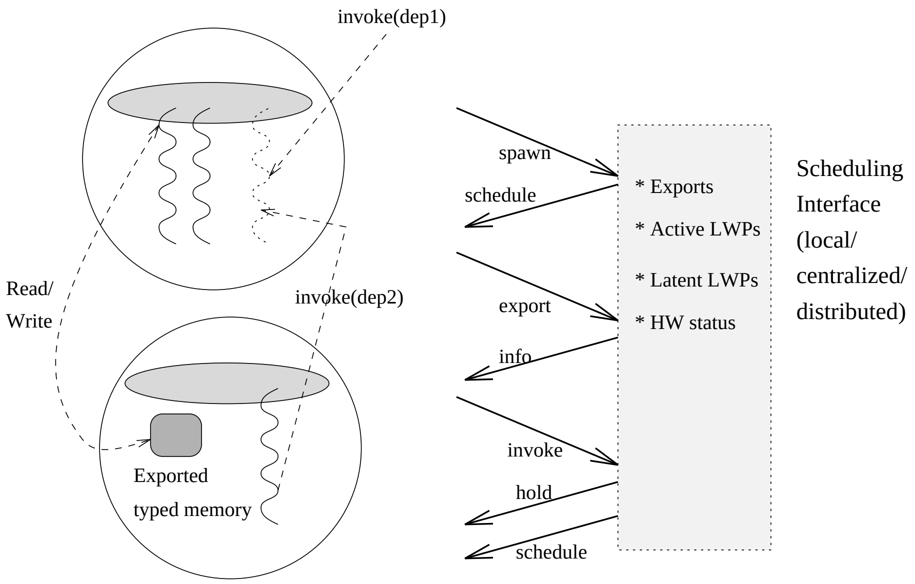


Figure 8: Schematic of TPVM Implementation

A thread in TPVM is essentially a subroutine/procedure (or a code segment, including nested procedure invocations, identified by one entry point). TPVM requires "strong encapsulation" of threads, i.e. shared variables are not permitted, although the system cannot enforce this restriction. Thus, a TPVM thread is a (sequence or collection of) subroutine(s) that, when initiated, possesses a thread of control and executes on its own stack, with its own data segment. Threads in TPVM exist within the context of PVM processes. However, processes in TPVM do not indulge in computation and/or communication; they only serve as "shells" or environments for threads to exist in, and play active roles only at certain points during execution e.g. for thread creation. A process may play host to (be the "pod" of) one or more threads, each of which may be an instantiation of the same or of different entry points. Application programmers access facilities in TPVM by invoking model-dependent functions; a detailed description may be found in [28].

7.2 Implementation and Experiences

Two different models are provided in the TPVM system and a preliminary version of each has been implemented and tested. The first is a process based model — TPVM supports a "traditional" concurrent computing model based on multiple interacting threads cooperating via explicit message passing. In this model, multiple threads, each with its unique thread id, are spawned and subsequently exchange messages using `send` and `receive` calls in a manner analogous to process spawning and interaction in PVM. However, to define threads that may be spawned, PVM pod (host) processes utilize functions such as `tpvm_export(. . .)` which declares a potential thread identified by a symbolic string valued name, and associated with a function entry point; additional parameters specify options, and limits to the number of threads allowed within this pod. An already existing thread or a pod process may subsequently spawn one or more instantiations of an exported thread which then communicate and synchronize via TPVM variants of the usual PVM routines.

Our experiences with a preliminary implementation of this process-oriented model for TPVM have been encouraging, both from the viewpoints of functionality as well as performance. Two textbook applications, namely matrix multiplication and sorting, were written to conform to this model and tested on our experimental implementation; results are shown in Table 5. As can be seen from this table, the TPVM versions perform better, by a factor of upto 24% — with the largest gains occurring when the granularity and the number of threads are "ideal". As some entries show, TPVM performance is worse when the overheads of thread management offset any gains due to the increased asynchrony.

Problem (size)	PVM: CPUs/Processes		TPVM: CPU's/Threads		
	4/4	16/16	4/16	16/64	16/36
Matmul (500x500)	201	78	182	94	60
Matmul (1000x1000)	1618	756	1610	710	663
Sort (2M integers)	1423	366	1201	351	309

Table 5: PVM vs. TPVM times in seconds (SS1+ workstations on Ethernet)

Motivated by the well known advantages of data driven computing, as well as by observations that scheduling is of critical importance for high performance, the TPVM system also supports a computing model that is different from the process-based paradigm. In this scheme, thread entry points are exported as before, but contain a list of "firing rules" that are required to be satisfied before a thread can be instantiated. The last two arguments of `tpvm_export` contain the size of, and a pointer to, an array containing a list of message tags — implying that the specified thread may be instantiated when one message of each tag type is available. Typically, as threads complete some portion of their allocated work, they are able to (partially) satisfy such a dependency. In order to indicate that a thread is able to satisfy a firing rule of an exported service, and to deliver a message containing the required data, special invocation functions are provided. Table 6 indicates the measured performance for the matrix multiplication and sorting examples mentioned earlier; results from the process-based model are also included for convenient comparison. As can be seen, the data driven model performs at approximately the same levels in some cases, and significantly better in others.

Problem (size)	ProcModel: CPU's/Threads			DflowModel: CPUs/Threads	
	4/16	16/64	16/36	4/?	16/?
Matmul (500x500)	182	94	60	194	62
Matmul (1000x1000)	1610	710	663	1582	640
Sort (2M integers)	1201	351	309	1167	309

Table 6: TPVM ProcModel vs. DflowModel times in seconds (SS1+ workstations on Ethernet)

8 Discussion

In this paper, we have presented the paradigm of concurrent computing on heterogeneous clusters using PVM, both from the operational perspective and in terms of future trends and research initiatives. As is evident from the material presented, PVM is both a robust heterogeneous computing system and an ongoing experimental research project, and continually evolving new ideas are investigated both by the project team and at external institutions; successful experimental enhancements or subsystems eventually become part of the software distribution. One example of a relatively concise enhancement that is undergoing investigation concerns system level optimizations for operating in shared memory environments. Small-scale SMM's are re-emerging, and a version of PVM that utilizes physical shared memory for interaction between the daemon and all user processes on such machines is being developed. Another project is aimed at providing fail-safe capabilities in PVM [20]. This enhanced version uses checkpointing and rollback to recover from single-node failures in an application-transparent manner, provided the application is not dependent on real-time events. Several other enhancements are also in progress, including load balancing extensions, integrating debugging support, and task queue management [21].

Apart from the experimental and research-oriented work described above, a number of other projects related to PVM are in various stages of progress; a few representative ones are:

- HeNCE [23] is a graphical programming system for PVM; this toolkit generates PVM programs, from depictions of parallelism dependencies as directed graphs, and provides an interactive administrative interface for virtual machine configuration, application execution, and animated visualization.

- XPVM is a graphical tool for PVM administration and profiling of PVM programs. It gathers monitoring events from applications, and displays this information, which can be useful for profiling, error detection, and optimization. This system is complemented by a buffered tracing scheme that permits monitoring data to be gathered without perturbing the application or overloading the network.

- The notion of "groups" and "contexts" are being introduced into the PVM system. Communications contexts are needed in situations where embedded applications exchange messages and message tags intended for an embedded library can potentially conflict with another, or with the host application. Such situations are avoided by specifying that message exchanges occur in identifiable sessions or contexts; grouping of processes to define process boundaries and establish contexts complements this methodology.

- The next version of the system, PVM 3.4, is scheduled to use a *Receiver makes Right* (RMR) data encoding instead than the XDR and RAW encodings now available. While some data conversion is needed to ensure that machines with differing data representations inter-operate correctly, the traditional methods of unconditionally converting to a standard form at the sender and reconverting at the receiver is probably wasteful. With fewer different data representations in emerging computer architectures, an optimistic scheme where the receiving end converts the data when necessary, is possible and more efficient.

- The DoPVM subsystem [22] is aimed at supporting the "shared object" paradigm in PVM. By writing C++ programs in which objects derived from built-in classes can be declared, this PVM extension permits a shared address space concurrent computing model, thereby alleviating the inherent complexity of explicit message passing programming.

References

[1] R. F. Freund and H. J. Siegel (eds.), IEEE Computer Special Issue on Heterogeneous Processing, 26(6), June 1993.

[2] V. S. Sunderam and R. F. Freund, J. Parallel and Distributed Computing, Special Issue on Heterogeneous Processing, (to appear) January 1994.

[3] R. F. Freund, SuperC or Distributed Heterogeneous HPC, 2(4):349-355, 1991.

[4] J. Potter, Associative Computing, Plenum Publishing, New York, 1992.

[5] L. H. Turcotte, A Survey of Software Environments for Exploiting Networked Computing Resources, Technical Report, ERCCFS, Mississippi State University, June 1993.

[6] D. Y. Cheng, A Survey of Parallel Programming Languages and Tools, NAS Systems Division Technical Report RND-93-005, NASA Ames Research Center, March 1993.

[7] L. Patterson, et. al., Construction of a Fault-Tolerant Distributed Tuple-Space, Proc. 1993 Symposium on Applied Computing, Indianapolis, February 1993.

[8] D. Gelernter, Domesticating Parallelism, IEEE Computer, 19(8):12-16, August 1986.

[9] J. Boyle, et. al., Portable Programs for Parallel Processors, Holt, Rinehart, and Winston, 1987.

[10] R. Hempel, The ANL/GMD MAcros (Parmacs) in Fortran for Portable Parallel Programming Using Message Passing, GMD Technical Report, November 1991.

[11] V. S. Sunderam, PVM : A Framework for Parallel Distributed Computing, Journal of Concurrency: Practice and Experience, 2(4):315-339, December 1990.

[12] V. Rego and V. S. Sunderam, Experiments in Concurrent Stochastic Simulation: The ECLIPSE Paradigm, Journal of Parallel and Distributed Computing, 14(1):66-84, January 1992.

[13] H. Nakanishi, V. Rego, and V. S. Sunderam, Superconcurrent Simulation of Polymer Chains on Heterogeneous Networks, Proceedings – Fifth IEEE Supercomputing Conference, Minneapolis, November 1992.

[14] D. H. Bailey, et. al., The NAS Parallel Benchmarks, International Journal of Supercomputer Applications, 5(3):63-73, Fall 1991.

[15] S. M. White, Implementing the NAS Benchmarks on Virtual Parallel Machines, Emory University M. S. Thesis, April 1993.

[16] S. M. White, A. Anders, and V. S. Sunderam, Performance Optimization of the NAS NPB Kernels under PVM, Proc. Distributed Computing for Aeroscience Applications, Moffett Field, October 1993.

[17] Flory, P. J., Statistical Mechanics of Chain Molecules, Interscience, New York, 1969.

[18] Stauffer, D., Introduction to Percolation Theory, Taylor and Francis, London, 1985.

[19] S. Moyer and V. S. Sunderam, "Parallel I/O as a Parallel Application", *International Journal of Supercomputer Applications*, Vol. 9, No. 2, summer 1995.

[20] J. Leon, et. al., "Fail Safe PVM: A Portable Package for Distributed Programming with Transparent Recovery", School of Computer Science Technical Report, Carnegie-Mellon University, CMU-CS-93-124, February 1993.

[21] J. Dongarra, et. al., *Abstracts: PVM User's Group Meeting*, University of Tennessee, Knoxville, May 1993.

[22] C. Hartley and V. S. Sunderam, "Concurrent Programming with Shared Objects in Networked Environments", *Proceedings – 7th Intl. Parallel Processing Symposium*, pp. 471-478, Los Angeles, April 1993.

[23] A. Beguelin, et. al., "HeNCE Users Guide", University of Tennessee Technical Report, May 1992.

[24] D. Bailey, J. Barton, T. Lasinski, and H. Simon, eds. "The NAS Parallel Benchmarks". Report RNR-91-002, Moffett Field, CA: NASA Ames Research Center, 1991.

[25] D. Bailey, E. Barszcz, et al. "The NAS Parallel Benchmarks." *International Journal of Supercomputer Applications*, Vol. 5, No. 3, pp.63-73, Fall 1991.

[26] A. Alund, S. White, and V. S. Sunderam, "Performance of the NAS Parallel Benchmarks on PVM Based Networks", *Journal of Parallel and Distributed Computing*, Vol. 26, No. 1, pp. 61-71, April 1995.

[27] S. Moyer and V. S. Sunderam, "PIOUS: A Scalable Parallel I/O System for Distributed Computing Environments", *Proceedings — 1994 Scalable High Performance Computing Conference*, May 1994.

[28] V. S. Sunderam, "Heterogeneous Concurrent Computing with Exportable Services", *Proceedings — Workshop on Environments and Tools for Parallel Processing*, SIAM Press, May 1994.

The Tool-set for PVM [†]

Thomas Ludwig, Roland Wismüller
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik der Technischen Universität München
D-80290 München, Germany
Tel.: +49-89-2105-2042 or -8164 or -8240, Fax: +49-89-2105-8232
E-mail: {ludwig,wismuell,bode}@informatik.tu-muenchen.de

Abstract

The Tool-set for PVM will comprise a set of integrated tools which can either be used individually or in concert. Tools will be subdivided into interactive tools for investigation and manipulation of the program state and automatic tools. Both types of tools will use online data, trace data, and checkpoint information to perform their task. The Tool-set will be composed of a debugger, a performance analyzer, a visualizer, a deterministic execution controller, a load balancer including a checkpoint generator, and a parallel file system. All tools will be available under the GNU General Public License Agreement. First versions of the tool environment will be released in spring 1996; some of its components will be available earlier.

1 Introduction

The PVM programming library has become an increasingly popular de-facto standard for writing explicitly parallel programs based on the message passing paradigm. A large variety of parallel applications, e.g. multigrid solvers, image processing or gene sequence analysis has been implemented on top of PVM. PVM is also being used as a platform for developing tools for parallel programming. Although some of them are rather sophisticated, currently only few tools are available and usable for the application programmers' community. One reason for this fact is that many tools are pure research prototypes providing only rudimentary, clumsy interfaces difficult to use for non-experts. A second reason is that existing tools are not integrated and cover only a single aspect of parallel program development. Here the term "integrated" does not primarily mean an integrated implementation of these tools, but the more important aspect of integrated usage, that is whether or not the tools can be used in combination and also provide support for each other. Currently, different tools can usually not be used for the same program, since they require the program to be compiled in different ways or to be linked with instrumented libraries not compatible with each other.

A new project of the Lehrstuhl für Rechnertechnik und Rechnerorganisation, Technische Universität München (LRR-TUM) will change this situation by providing an integrated tool-set for PVM including both run-time support and programming tools. The project is based on the previous work of our group that is performing research in the field of parallel processing tools for more than eight years[1]. It will integrate and adapt our existing tools and experience to form a PVM tool environment supporting parallel I/O [8], load balancing [5], resource management and checkpointing [9], performance analysis [4, 3], debugging [7], deterministic execution [6], and visualization [2] (the citations refer to already existing work). Since we can rely on considerable

[†] This work is partly funded by the German Science Foundation, Contract: SFB 342, TP A1
[1] If you are interested in the ancestors of The Tool-set, please refer to [1].

preparatory work that always aimed towards industrial quality, easy-to-use interfaces, and integration of programming tools, we expect a first version of THE TOOL-SET to be available for the user community in spring 1996.

2 THE TOOL-SET

This section will give an overview over the concepts of THE TOOL-SET. Many of them are already fixed and are waiting just to be implemented, others are still research topics and require more conceptual work. As the complete tool-set has a high internal complexity, not all features will be available at the same time with full functionality. The end of this paper will present a preliminary time schedule and some first deadlines.

The tools can be divided into classes according to the following criteria. First, we will offer interactive tools and automatic tools. Interactive tools (like THE DEBUGGER, THE LOAD BALANCER) will support implementation and maintenance phases whereas automatic tools (THE LOAD BALANCER) mainly concentrate on the production phase of the software. Tools will use different sources of information. A monitoring system will give direct access to the running program on the workstation cluster, thus supporting online debugging and performance measurement. All measured characteristics can be recorded in traces which describe the individual behavior of a single program execution. Traces can be used e.g. for statistical program analysis. In addition, checkpoints will be generated which represent the state of a program at a given point of time. Checkpoint data is mainly used for load balancing purposes. Traces and checkpoints can be entered in a database system for comparison, version management etc. All tools are based upon these few information types, however, none of them uses the complete set. Interactive tools will be integrated considering their graphical user interface. Implementation of the GUIs will be based on OSF/Motif and therefore give the users a standardized look and feel.

Finally everything is grouped around PVM version 3.3.x. We expect THE TOOL-SET to be easily adaptable to future versions of PVM as only the monitoring system is closely interconnected with internal PVM mechanisms. However, any change of programming model (e.g. new communication types) would also require an adaption of those interactive tools which should be able to handle the new constructs.

Figure 1 shows the module structure of THE TOOL-SET. Due to the limited space of this paper we can only give a list of some selected highlights and a short description of two of the tools: of THE DEBUGGER and of THE PERFORMANCE ANALYZER.

- THE DEBUGGER can use checkpoint information to resume execution of a program starting from a specific point during the program run. Debugging cycles will become shorter as the program has not to be restarted from the beginning.
- THE PERFORMANCE ANALYZER will have extended functionality to try to automatically detect a focus of interest, e.g. workstations with high idle-time or communication frequency. It will also offer statistical data and comparison of different program runs by using traces.
- THE VISUALIZER will show the behavior of PVM programs at the level of tasks and communications thus unveiling e.g. deadlock situations.
- THE DETERMINISTIC EXECUTION CONTROLLER (also called THE DETERMINIZER) supports deterministic runtime behavior of a parallel program thus ensuring reproducible program runs and results during test phases.
- THE LOAD BALANCER can migrate running processes from loaded workstations (e.g. owner wants to work at his console) to free workstations, using the mechanisms of the checkpoint generator CoCheck. Furthermore, the heuristics for controlling migration decisions can be improved by learning from trace material from previous program runs.
- THE PARALLEL FILE SYSTEM will support various modes for accessing data files. Moreover, all activities can be monitored and evaluated by THE PERFORMANCE ANALYZER.

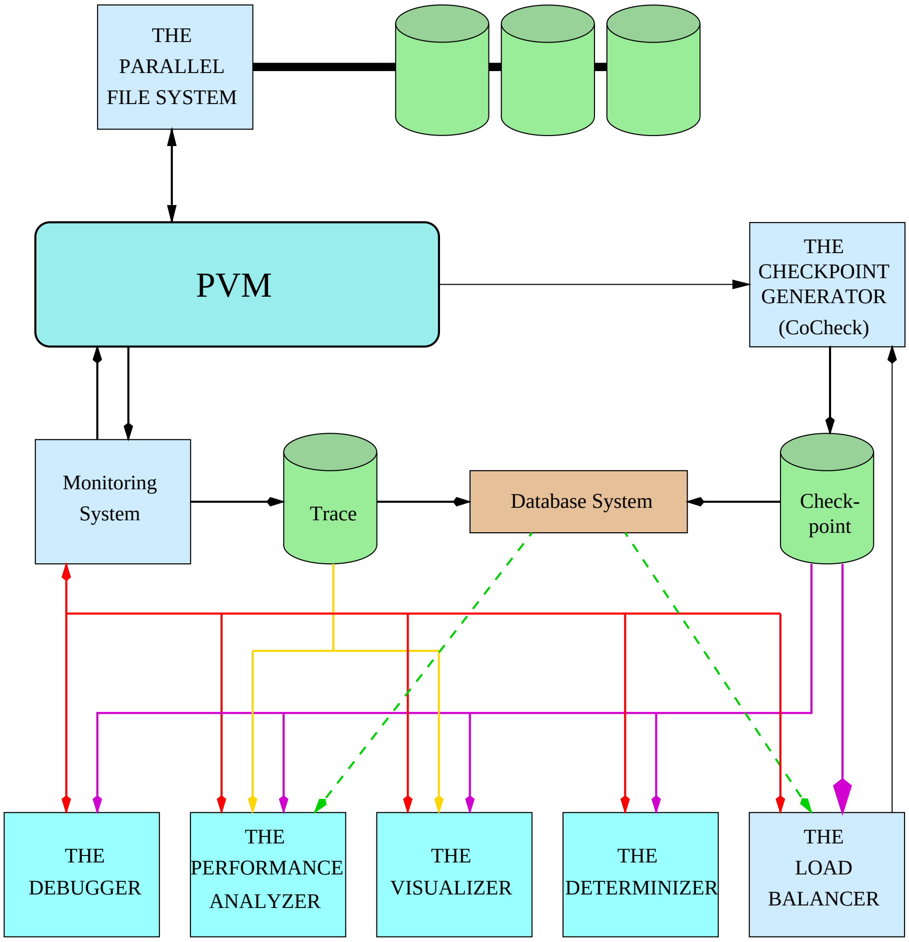


Figure 1: Module structure of the complete tool environment

2.1 The Debugger

The Debugger will provide support for the efficient source level debugging of distributed programs written in C or Fortran 77. In contrast to other projects it is not a simple interface built around a sequential debugger, but a real parallel debugger with special support for PVM. The Debugger offers a comfortable graphical interface which is based on a debugger window showing the source code and command output. In addition, visualizers for regular and also irregular data structures will be provided. The debugger window can be associated with an arbitrary set of tasks that is displayed in a list. All debugger commands, e.g. single-step, set breakpoint or print, are applied to all these tasks or a selectable subset. So data parallel programs where all tasks execute roughly the same code can be examined using a single window. Multiple windows can be used to debug groups of tasks executing different codes.

Besides the flexible user interface, The Debugger provides a variety of other features essential for parallel programming. We will discuss only the most important ones here. First of all, PVM is fully supported. The Debugger can be used for heterogeneous environments and allows to inspect PVM objects like a task's incoming message queue or a barrier's task queue. Tasks can be identified not only by their ID, but also by additional information, such as host, file and group names or parent task. Based on patterns using this information, dynamically spawned tasks can be automatically stopped at the beginning and may be added to any debugger window.

Second, The Debugger uses the event-action paradigm instead of a simple breakpoint scheme. Events which may be parameterized, represent special conditions in the program, e.g. 'task reaches a source line' or 'message is received'. Each event defined can be associated with a list of actions that are executed when the event occurs. Actions include stopping any set of tasks, tracing the event, printing variables or defining new events and actions. Since actions are evaluated autonomously by the distributed monitoring system without interaction with the debugger's front end or the user, intrusion is kept very low. Furthermore, actions may also be triggered by events on remote hosts, so global halting and distributed breakpoints are possible. We will also provide special breakpoints allowing to follow message transfers: when a task sends a message, the breakpoint will stop the receiver immediately after it has received that message, so data processing can be watched across task boundaries.

Finally, THE DEBUGGER will be integrated with THE DETERMINIZER and CoCheck, making cyclic debugging practical. Parallel programs usually run for a very long time, so re-running them becomes a tedious job. In addition, program behavior may not be reproducible due to race conditions. Therefore, a form of backtracking will be provided by generating a checkpoint and saving the debugger's configuration upon user request. The user may then return relatively quickly to that point and re-execute the last part of the program. THE DETERMINIZER will then either ensure reproducible behavior or may enforce a different message ordering, so the effects of race conditions can be examined.

2.2 THE PERFORMANCE ANALYZER

THE PERFORMANCE ANALYZER will we able to handle two different kinds of measurements types: online analysis and trace evaluation. Both types offer slightly different functionality and are distinguished by the fact that a trace reflects a fixed program run of the past which includes only a limited amount of measured data. Measures which have not been evaluated during runtime are lost for trace analysis.

Let us have a closer look at the functionality of THE PERFORMANCE ANALYZER. It provides analysis on four levels of abstraction: at the level of the virtual machine, the node level, the task level, and the function level. Three types of load characteristics can be evaluated: the amount of computation (CPU-time etc.), communication characteristics like number of bytes sent/received, and I/O characteristics (also volume and time based). An investigation of these measures should give the programmer some idea on possibly inefficient constructs in his program. However, with an increasing number of tasks it becomes difficult to detect the nodes of interest. Therefore, THE PERFORMANCE ANALYZER offers special attributed measurements. The nodes which shall contribute to a value are specified via a predicate. If the predicate is not true, their current value is disregarded. Currently, specification of predicates has to be done manually. In future, we will think about methods of how to add some automatism, e.g. by stepwise strengthening user-defined predicates in order to minimize the set of interesting nodes. By having identified nodes with unusual behavior it would be possible to automatically start a refinement of performance analysis.

Attributed measurements not only make it easier to find bottlenecks but also improve the scalability of the tool. Especially with a higher number of nodes or tasks this feature can reduce the amount of information that has to be displayed on the screen.

Apart from offering current values, mean values, and a graphical representation of values changing during runtime THE PERFORMANCE ANALYZER will support enhanced statistical functionality like minimum and maximum values, statistical distributions of values over the time-axis, and maybe user definable combinations of primitive performance measures. Finally, THE PERFORMANCE ANALYZER will support the program dynamics of PVM. Not only will it be possible to handle a dynamically changing set of active workstations, but also to evaluate the positive or negative influence of load balancing. This will be achieved by a monitoring system which informs the tool about every change of the current process graph and its mapping.

3 Project Status and Availability

Currently we are implementing The Visualizer, The Parallel File System, and CoCheck. The latter will be available in 9/95. As the monitoring system is not yet implemented, The Visualizer will preliminaryly be based upon traces generated by PVM and XPVM. We have not yet decided whether we will also provide a version of The Performance Analyzer for this trace type.

The next important step to be made is the implementation of the monitoring system which is a prerequisite to all interactive tools and to The Load Balancer. With the monitor being finished we will adapt our existing tools to the instrumented PVM environment. First products should be available for the users in spring 1996. They will be fully functional but will not yet have integrated enhanced features which are current research topics of our group.

References

[1] H. Beier, T. Bemmerl, A. Bode, et al. TOPSYS - Tools for Parallel Systems. Research report SFB 342/9/90 A, Technische Universität München, Jan. 1990.

[2] A. Bode and P. Braun. Monitoring and Visualization in TOPSYS. In G. Kotsis and G. Haring, editors, *Proc. Workshop on Monitoring and Visualization of Parallel Processing Systems*, pages 97 – 118, Moravany nad Váhom, CSFR, Oct. 1992. Elsevier, Amsterdam, 1993.

[3] R. Borgeest. Cyclic Performance Debugging. Submitted for the 2nd European PVM Users' Group Meeting, Lyon, France, Sept. 1995.

[4] O. Hansen. A Tool for Optimizing Programs on Massively Parallel Computer Architectures. In *High-Performance Computing and Networking, Volume II*, volume 797 of *Lecture Notes in Computer Science*, pages 350 – 356, München, Apr. 1994. Springer, Berlin.

[5] T. Ludwig. Aspects of Load Management on Parallel Computers. In V. Malyshkin, editor, *Proc. International Conference Parallel Computing Technologies, PaCT-93, Obninsk, Russia*, pages 301–313, Vol. II, Moscow, Russia, Sept. 1993. Recursive Super Computers (ReSCo).

[6] M. Oberhuber. Elimination of Nondeterminacy for Testing and Debugging Parallel Programs. In *Proc. of the 2nd Int. Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, St. Malo, France, May 1995.

[7] M. Oberhuber and R. Wismüller. DETOP - An Interactive Debugger for PowerPC Based Multicomputers. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications*, pages 170–183. IOS Press, May 1995.

[8] C. Röder, S. Lamberts, and T. Ludwig. PFSLib - An I/O Interface for Parallel Programming Environments on Coupled Workstations. Submitted for the 2nd European PVM Users' Group Meeting, Lyon, France, Sept. 1995.

[9] G. Stellner and J. Pruyne. Resource Management and Checkpointing for PVM. Submitted for the 2nd European PVM Users' Group Meeting, Lyon, France, Sept. 1995.

Visualization of Parallel Program Execution †

Peter Braun[1] and Roland Wismüller[2]

[1] Siemens AG, ZFE T SN6,
Otto-Hahn-Ring 6, 81739 München

[2] Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik, TU München, 80290 München

Abstract

The paper describes a tool for debugging parallel programs by visualization and animation of their execution behavior. The visualization and animation tool VISTOP (VISualization TOol for Parallel Systems) has originally been developed for a programming library called MMK in a PhD-thesis [Bra94] as part of a tool environment for programming distributed memory multiprocessors. VISTOP supports the interactive on-line visualization of message passing programs based on various views, in particular, a process graph based concurrency view for detecting synchronization and communication bugs.

The paper presents the features of this original version of VISTOP, the process of porting the tool to the PVM programming model, the new features of the PVM version and some of the problems that occurred during the port.

1 Introduction

While the development of parallel hardware advances very rapidly with systems consisting of hundreds or thousands of processing elements, the development and improvement of techniques and tools for debugging and performance analysis are lagging behind modern hardware technology. This is one of the main obstacles of a widespread usage of these systems.

New methods to program high performance computer systems such as HPF (High Performance Fortran) try to facilitate their use, however, the majority of parallel systems with fully distributed resources are still programmed with an explicit parallel programming model. Programming libraries such as PVM provide a model of communicating sequential processes which exchange information solely by messages. The interaction of many independent processes causes additional difficulties and bugs in parallel programs such as communication errors or performance bottlenecks which occur in addition to problems within the sequential parts of the program. The understanding of the execution behavior of many concurrent processes is very difficult and requires adequate software tools for debugging and performance analysis. However, current parallel debuggers operate at a very low level of abstraction and cannot provide enough insight into the overall execution of parallel programs. This is one of the main reasons why current tools are very often not accepted by the programmers of parallel systems [Pan93].

The following section gives a brief description of the visualization tool VISTOP for the MMK programming model. This version is currently being ported in order to support PVM applications. The changes to be made and the problems that arise are described in Section 3.

†This work is partly funded by the German Science Foundation, Contract: SFB 342, TP A1

2 The Visualizer VISTOP for the MMK Programming Model

The current version of the visualization tool VISTOP (VISualization TOol for Parallel Programs) animates message passing programs developed with the parallel programming library MMK [BL90], which is similar to PVM or MPI. The basic idea of VISTOP is analogous to the classical approach to debugging sequential programs, which involves repeatedly stopping the program during execution, examining the state, and then continuing the execution. This is also the approach used in most commercially available debuggers for parallel and distributed systems.

VISTOP tries to overcome the limitations of these extensions of traditional low-level break-point debuggers for sequential programs by providing a global view of the program state at the level of abstraction of the programmer. A global perspective makes it much easier to detect the reason for communication errors such as deadlocks or races. Main features of VISTOP compared to other approaches are [Bra94]:

- Most visualization systems are post mortem systems which analyze the program execution off-line after the program has terminated. However, debugging is an interactive process where the user examines the state of the system in order to decide which information is needed next. This approach can only be supported effectively by on-line tools. For instance, VISTOP can be used in conjunction with our parallel debugger DETOP [BW94] in order to inspect values of variables or to define break-points. On-line use reduces the amount of information that has to be gathered from the application as only the information that is requested by the user must be collected.
- Many distributed systems are missing a global time base. Thus, most visualization tools that rely on time stamped trace events cannot guarantee to show a "correct" representation of the program execution. This is not acceptable for a tool that should serve to increase the understanding of the program execution. VISTOP is based on a formal time model which uses causal relationships to enforce that the visualization is consistent with causality [SM92].
- The presentation is based on the semantics of the programming model. It shows the communication protocols and communication errors, i.e. the tool can be used during the debugging phase of a program and does not require that the program must terminate correctly.
- VISTOP supports a dynamic process model which makes the tool applicable to a wide range of applications such as distributed database or traffic control systems.

The system provides three views that show different aspects of a parallel computation. The *concurrency view* is based on a hierarchical process graph and shows communication and synchronization events. The *object creation graph* presents dynamic object creation. The *system view* shows the distribution of the program objects on a parallel system.

2.1 Concurrency View

In the concurrency view, the program objects are arranged in a two-dimensional area. The basic presentation metaphor of the view is a dynamic, hierarchical object graph. The nodes of the graph form little icons which can represent either objects of the programming model or subgraphs of the program (see figure 1).

The MMK programming model provides three predefined object types. *Tasks* are the active components which perform the calculations of the program. They communicate via *mailboxes* and synchronize via *semaphores*. The communication protocol can be adjusted by specifying the buffer size of the mailboxes and by the selection of timeout intervals of the communication functions. For example, a synchronous communication between two task objects similar to the ADA rendezvous semantics can be specified by setting the buffer size to 0 and the timeout interval to unlimited.

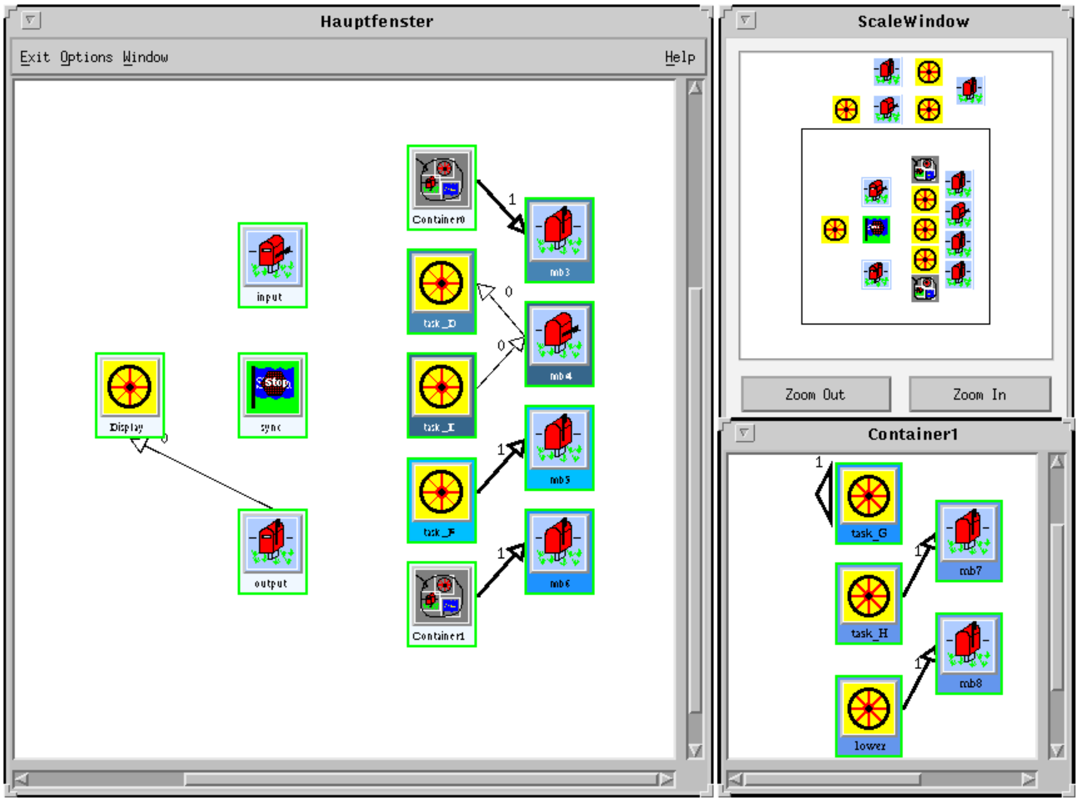


Figure 1: VISTOP Concurrency View.

Usually, programming objects are displayed by a small icon which shows a characteristic symbol for the object type and the identifier of the object. The size of the icon can be adjusted by zooming. In addition, the icons change their shape to show the current state of the object. This feature is used to visualize the communication protocol and error conditions. Additional information about objects can be displayed in a so-called "detail view" which can be obtained by selecting an object with the left mouse button. For instance, the detail view of a mailbox object shows the message buffer and the task objects which were sending the messages which allows to detect communication errors that are caused by messages from wrong senders.

The layout of the object graph should ideally reflect the communication structure of the parallel application in order to make the display most effective. The arrangement of the program object can be assigned by the user accordingly by selecting and moving the objects. Communication and synchronization events are indicated by arrows. A thin arrow characterizes a non-blocking communication or synchronization. Blocking library calls are visualized by thick arrows (see figure 1).

2.2 Hierarchical Abstraction

Hierarchical abstraction can be used at every level of computation to control complexity and make the construction of larger systems more manageable. While the method is widely applied in textual programming (e.g. functions, procedures, modules) and in design tools such as CASE tools, currently no existing visualization tool exploits this technique.

In VISTOP hierarchical abstraction is realized by "container objects". Each container consists of two parts, an icon and an associated concurrency window. The icon represents the group of program objects located in its concurrency window or one of its descendants. Program and container objects can be moved between different windows by drag and drop. Thus, container can be used to form a hierarchical program graph by distributing the program objects in an appropriate way.

In order to control presentation complexity of the display during visualization all concurrency windows can be closed and opened by selecting the respective container. Figure 1 shows the main window of the concurrency view that contains two container objects `Container0` and `Container1`. Only the concurrency window of `Container1` is visible whereas `Container0` is closed.

Communications between objects outside and objects inside a container are displayed by an arrow from the outside object to the container (see communication between `Container0` and mailbox `mb3`). Communication events within a container are only displayed in the concurrency window associated with the container, i.e. if the window is closed, the event is invisible (see communication between `task_H` and `mb7`). Communication between members of a container with communication partners outside are displayed by an arrow head (see `task_G`). The window that contains representations of both communicating objects shows the communication partner which is `mb6` in the example.

2.3 Object Creation View

This view displays the dynamic object creation chain of programming objects created and/or deleted frequently as it happens for example in database systems or tree search algorithms. The objects are represented in form of a graph consisting of at least one tree. An example for an object tree can be seen in figure 2. Each of the tasks that are declared statically form the root of a separate tree. The parent entry shows the task that created the child object dynamically. By selecting a node of the graph, additional information about the object is displayed (state of the object, module where the object was created, ...).

When a new object is created, this is animated in the object creation view as follows. First, the position of the object in the tree is determined. All objects below this position move down one line. A "turtle" appears below the task which creates the new task and draws a line from the parent to the new object. Now the turtle grows and finally the icon for the new object appears. The deletion of an object is animated in a way analogously to the creation.

2.4 System View

This visualization component shows the distribution of the objects of an application onto different processing nodes. Different processing nodes may be located on different machines, when a program is distributed in a network of UNIX workstations. The display is organized as a matrix. Each column contains all objects of one node. An object may be displayed as an icon that shows its type, its name or its object identifier. By selecting an object with the mouse, additional information about the object can be obtained. The VISTOP system view is displayed in figure 3.

This view can visualize migration of objects in applications which perform migration with the automatic load balancing facility of the MMK. When objects are created or deleted dynamically, this view shows the location of the objects and their overall distribution in the system. Again, the events displayed in this view are animated to show the dynamic behavior of the underlying application.

3 Porting VISTOP to PVM

The application of the MMK version of VISTOP proved that the visualizer is a valuable tool for understanding and debugging parallel programs. For PVM, however, no tool of comparable functionality existed. Therefore, we decided to port VISTOP to the PVM programming model.

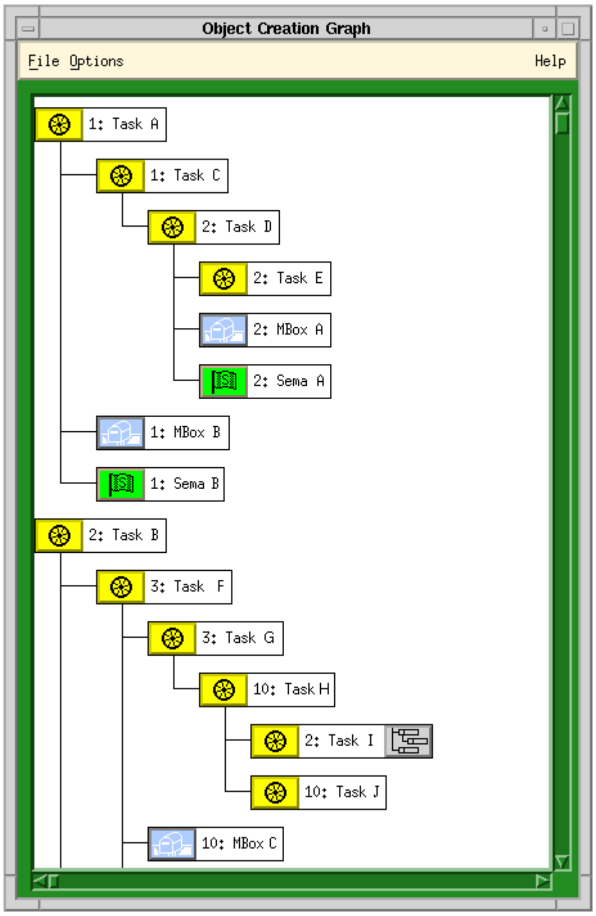


Figure 2: VISTOP Object Creation View

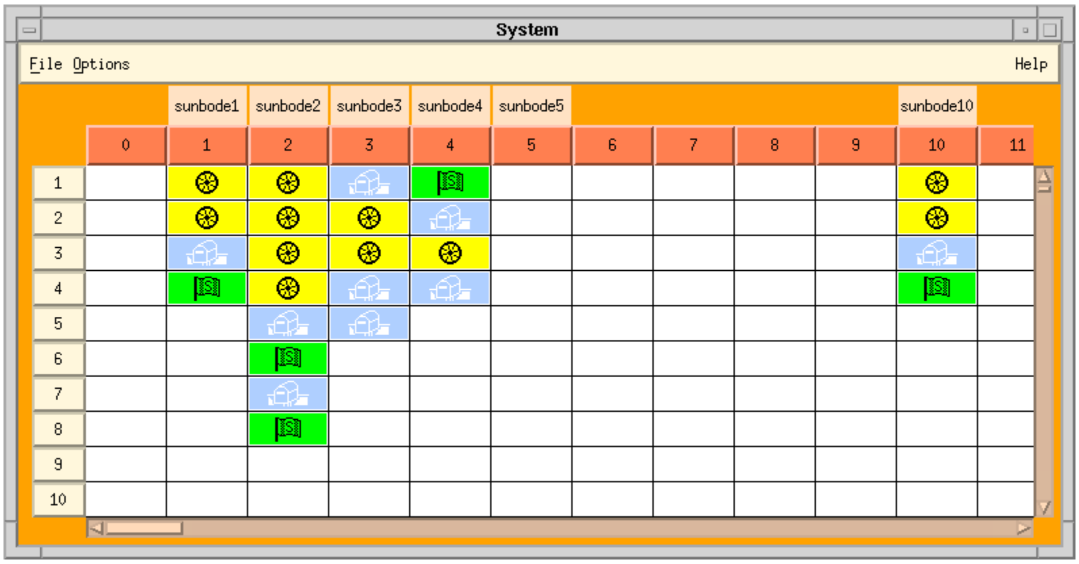


Figure 3: VISTOP System View

This port is one part of a much larger project called "The Tool-set for PVM" [LWB+95], where a set of integrated tools for the development and execution of PVM programs is being developed.

3.1 Structure of VISTOP

In order to show, which parts of the visualizer are affected by the port to another programming model, we will now briefly present the structure of VISTOP. It consists of three major parts (see Fig. 4):

1. The **data acquisition** is responsible for detecting relevant events in the execution of a program and for feeding the events to the modeling layer in a correct order. Events are either read on-line from a monitoring system, or off-line from a pre-recorded trace file. In any case, since the target systems usually don't provide an accurate global time, events from different processes must be sorted according to the causality ("happened before") relation [Lam78].

2. The **modeling layer** computes consistent global states from the event stream generated by the acquisition layer. This computation also takes into account the semantics of the programming model, so the computed state can provide information that is not directly contained in the events. For instance, the model can decide whether or not a receive call blocks its caller, since it computes the queue of receivable messages from the event stream. If the queue is empty, the task must be blocked. In the same manner, error conditions, e.g. receiving a message with wrong data type or length can be detected by the model.

3. The **visualization layer** finally visualizes different views of the global state computed by the model. It also controls the two other layers.

From this description, it is clear that most of the work has to be done in the two lower layers. Since we have a new programming library and also new target hardware, we need a different monitoring system. Also the event processing has to be adapted, since events and parameters have changed, and there are new interactions (e.g. barrier synchronization) that influence the computation of a causal event ordering. The modeling layer requires major modifications, since the semantics of PVM is very different from the semantics of the MMK programming model. Therefore, the way how global states are computed from events is also different. Finally, visualization patterns for the new object and interaction types in PVM have to be incorporated into the visualization layer. However, most of its features can be reused.

3.2 Visualization Patterns

In contrast to the MMK, there are no mailboxes and no semaphores in PVM; communication is performed directly between tasks. For synchronization, there is a new object, the barrier. In addition, PVM offers process groups and new communication patterns: broadcast in a group, multicast to a list of tasks and receiving messages without specifying the sender.

These differences must be reflected in the patterns that are used to visualize communication and synchronization in the concurrency view. The new patterns are depicted in Fig. 5: The task icons are identical to that of the MMK version. In addition, there is a new symbol for barriers. It is created when the first task in a group calls `pvm_barrier()` and is destroyed when the last task leaves the synchronization call.

There are no explicit objects visualizing groups in the concurrency view. A first idea has been to visualize each group by a container object that contains all the tasks in that group. However, containers in VISTOP form a strict hierarchy of tasks, i.e. a task can only be in one container, while in PVM, a task can belong to more than one group. Thus we decided to provide an additional view, called group manager. It displays the list of all groups that exist at the current point in the

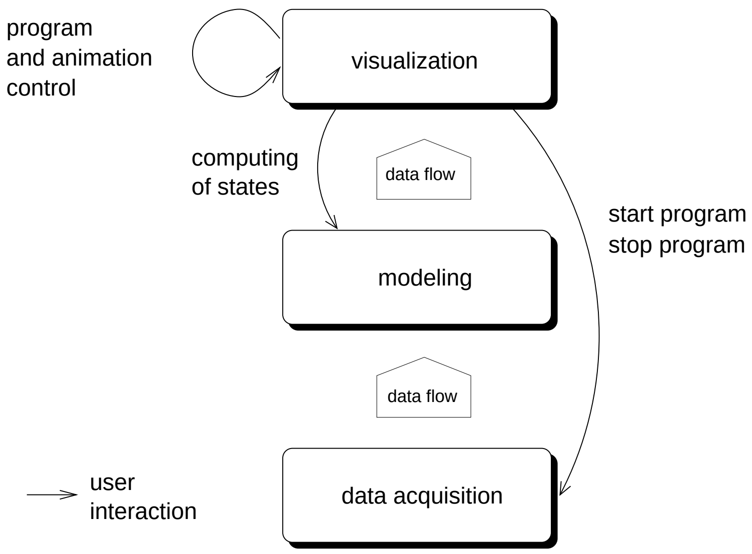


Figure 4: Structure of VISTOP

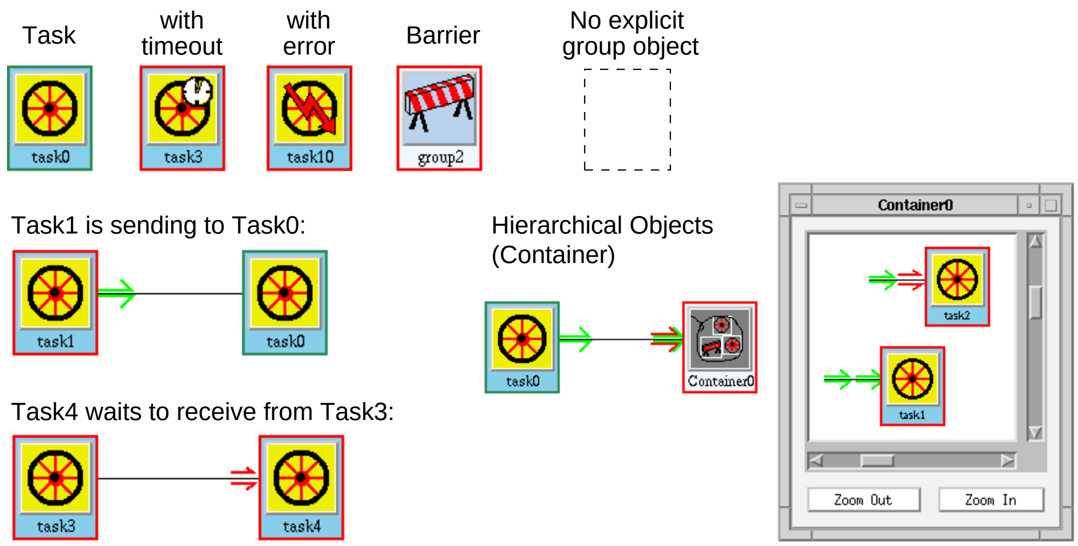


Figure 5: New visualization patterns

animation, and allows to select a single group, a union, or an intersection of groups. When this is done, all visualized objects that do not belong to the specified group (or union/intersection) loose parts of their coloring, as shown in Fig. 6. In this way, users can quickly determine the group membership of visualized objects.

Since PVM uses direct communication between tasks and offers some new communication structures, also the way of visualizing interaction between objects has been modified, as shown in Fig. 5. Instead of drawing thin or thick arrows between objects, now a simple line is drawn, denoting that there is some interaction. Additional arrow heads in different colors on either side of the line specify the kind of interaction: A green (filled) arrow head denotes an interaction that does not block the active partner, while a red (hollow) arrow head denotes a blocking interaction. The direction of the arrow head indicates the direction of the (requested) information transfer, while its position marks the active partner of an interaction. For example, in the middle left of Fig. 5, `task1` is sending to `task0`, since there is a filled arrow head attached to `task1` that is directed towards `task0`. The lower left of the figure shows the visualization pattern when `task4` waits for a message from `task3`. The new visualization strategy can also show multiple interactions, e.g. between a task and a container object. In the lower right of Fig. 5, `task0` sends a broadcast to `task1` and `task2`, both in the container, while `task1` receives a message from `task0`. `task2` didn't start receiving yet and is still waiting for a message from `task0`. Broadcasts and multicasts are visualized in an intuitive way by showing that a task sends to multiple tasks at the same time (see `task0B` in Fig. 6).

In addition to these changes in the concurrency view, the detail view of task objects has been adapted to the task information available with the PVM programming model (see Fig. 7).

3.3 Data Acquisition and Modeling

Although the long term goal for our work is to provide the visualizer VISTOP for PVM based on an on-line monitoring system, the first version uses XPVM for data acquisition. XPVM records a trace of events in a PVM application and is able to store this trace in an SDDF file [GKP94, BDG+95]. Using XPVM for data acquisition allows VISTOP can be used for PVM before our own monitoring system is ported. However, we currently loose the interactive control over the application's execution. Therefore in a next step, the visualizer will make use of the monitoring system developed in the OMIS project [LWSB95] and will then also become part of The Tool-set.

The modeling of global states from a sequence of events turned out to be much more complicated for PVM than it was for the MMK programming model. The main reason for this fact is the absence of an event that indicates the insertion of a message into a task's message queue. Such an event is not provided by the built-in trace facility of XPVM. For the modeling of global states, this has three consequences:

1. The message queue of a task cannot be computed accurately. All the model can provide is a *set* of receivable messages. It cannot determine the order in which messages are stored in the queue.

2. The VISTOP animation is designed to show when a task blocks in a receive call due to an empty message queue. Since the point in time, when a message is put into the message queue of a task is unknown, it is not possible to decide whether the task blocks in a receive call or not. As the current solution, we assume that a message is inserted into the receiver's queue when the sender finishes its send operation.

3. It is impossible to easily visualize receive operations in the usual way, if no sender is specified in the receive call, i.e. a wild-card is used. Since the order of messages in the task's message queue cannot be modeled properly, it is not known in this situation, which task is the sender of the message that is being received.

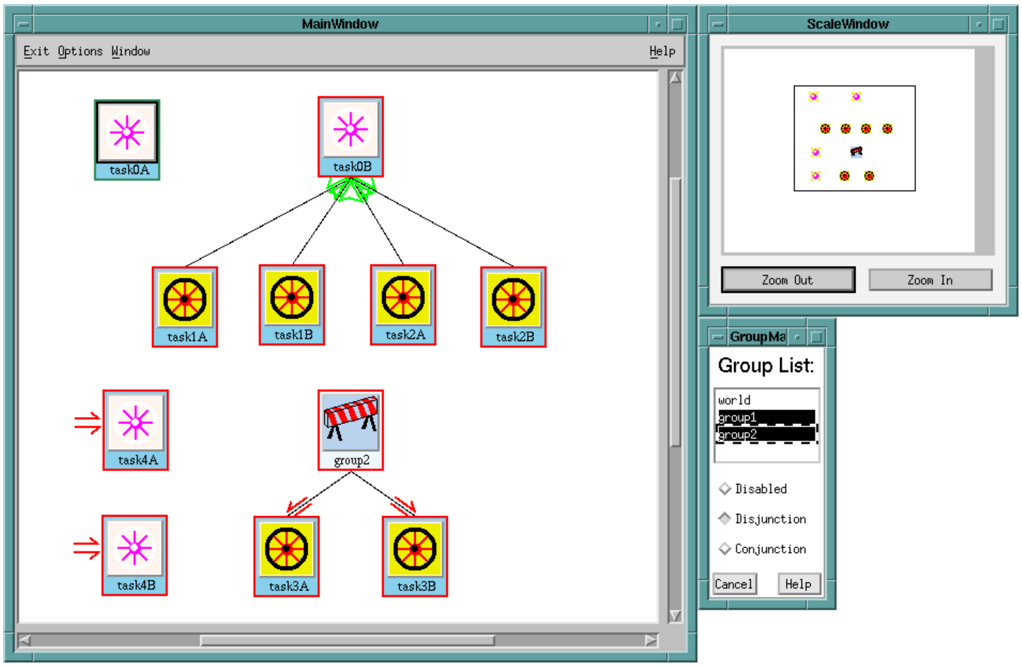


Figure 6: New concurrency view

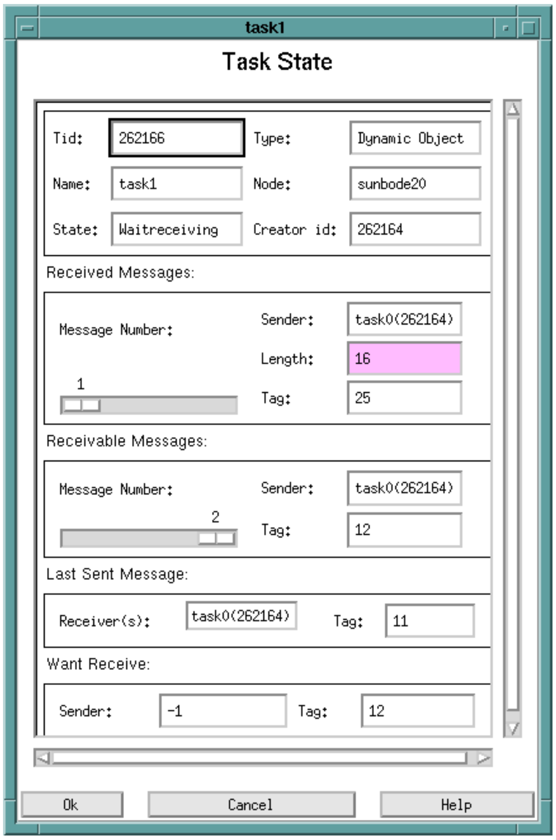


Figure 7: Detail view of a task

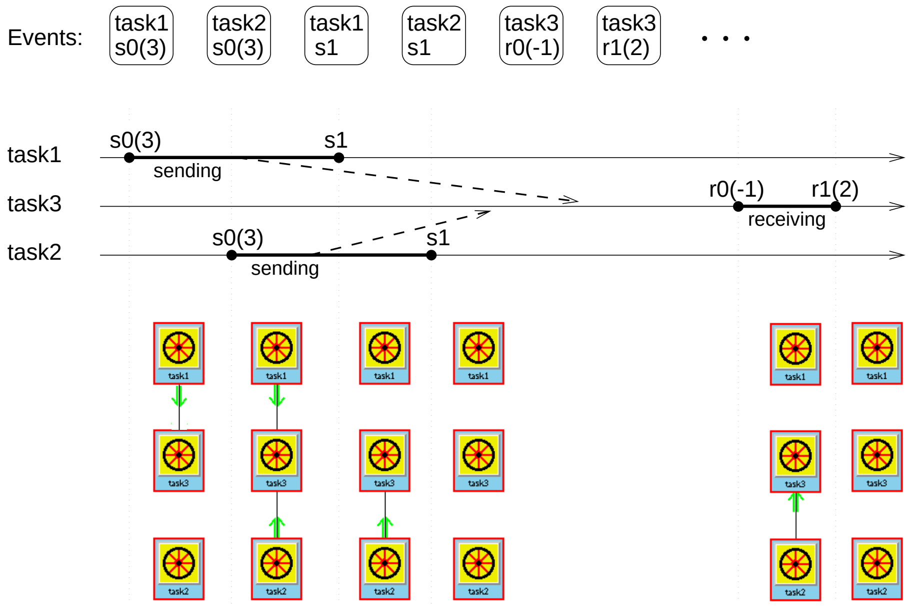


Figure 8: Processing of communication operations

Fig. 8 illustrates the modeling process and the last problem just mentioned. The first line shows a simplified event stream that is the input for the model. `s0/s1` denote the start/end of a send call, `r0/r1` the start/end of a receive operation. The numbers in parentheses indicate the destination or source task, -1 denotes a wild-card. The space-time diagram below shall indicate the result of the modeling process: a sequence of global states is computed. Finally, as shown in the bottom half of the figure, each global state is visualized by a graphical pattern.

For the first four states, the modeling and visualization process is quite straightforward. However, a problem arises when event `r0(-1)` is beefing processed. The global state after this event is the following: "`task3` is receiving a message from `task2`"[1] However, according to the description above, the model cannot determine the sender of the message being received until it sees the event `r1(2)` that always contains the actual sender. So in order to visualize correct global states, the receive operation is shown not before the modeling layer has processed the end-of-receipt event. Then an artificial event is introduced that causes the model and the visualization to return to a state where the task finished receiving. So in effect, receive operations are shown one step too late, but a simple analysis shown that at least the causality relation cannot be violated by this delay.

As a longer term solution, we will try to provide the crucial event "a message is inserted into a tasks message queue" with the PVM on-line monitoring system developed in the OMIS project. However, this will most probably require some modifications in the PVM daemons. We will therefore try to further cooperate with the PVM development team in order to avoid the necessity of a special PVM version for the usage of our tools.

4 Conclusion

The application of VISTOP for the MMK programming model has shown that visualization can increase the understanding of parallel programs. Hierarchical abstraction is quite helpful to in-

[1] Even if the messages have the same tag and arrive at `task3` in the order suggested by the dashed arrows, the task can still receive `task2`'s message first, if there is a direct connection between `task1` and `task3`. In this case, PVM will always first ask the daemon for a receivable message before it checks the direct connections.

crease the scalability of visualization tools for functional debugging of parallel programs. The concept is easy to understand and can be applied to any parallel program.

The visualizer is now also available for the PVM programming environment in a first version. It currently animates SDDF traces generated by the XPVM tool. Future work includes the development of a PVM on-line monitor that will acquire the data needed for VISTOP, and an integration of the visualizer into THE TOOL-SET.

References

[BDG+95] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Recent Enhancements to PVM. *International Journal of Supercomputing Applications and High Performance Computing*, 1995.

[BL90] T. Bemmerl and T. Ludwig. MMK - A Distributed Operating System Kernel with Integrated Dynamic Loadbalancing. In H. Burkhart, editor, *Proc. of CONPAR90 VAPP IV*, pages 744–755, Zürich, Schweiz, 1990. Springer-Verlag.

[Bra94] Peter Braun. *Visualisierung des Ablaufverhaltens paralleler Programme*. PhD thesis, Technische Universität München, 1994.

[BW94] T. Bemmerl and R. Wismüller. On-line Distributed Debugging on Scalable Multiprocess or Architectures. In *High-Performance Computing and Networking*, volume II, pages 394–400, München, April 1994. Springer.

[GKP94] A. Geist, J. Kohl, and P. Papadopoulos. Visualization, Debugging, and Performance in PVM. In *Proc. Visualization and Debugging Workshop*, October 1994.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LWB+95] T. Ludwig, R. Wismüller, R. Borgeest, S. Lamberts, C. Röder, G. Stellner, and A. Bode. THE TOOL-SET – An Integrated Tool Environment for PVM. In *Proceedings of EuorPVM'95 Short Papers*, Lyon, France, September 1995. Ecole Normale Supérieure de Lyon. Technical Report 95-02.

[LWSB95] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS – On-line Monitoring Interface Specification. Internal report, Technische Universität München, 1995.

[Pan93] Cherri M. Pancake. Customizable portrayals of program structure. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 64 – 74, San Diego, Ca, 16. – 18. Mai 1993.

[SM92] Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Technical Report SFB124-15/92, Universität Kaiserslautern, 1992.

Checkpointing and Process Migration for PVM

Georg Stellner
Institut für Informatik der Technischen Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation
D-80290 München
stellner@informatik.tu-muenchen.de

Abstract

Checkpoints cannot only be used to increase fault tolerance, but also to migrate processes. The migration is particularly useful in workstation environments where machines become dynamically available and unavailable. We introduce the CoCheck environment which allows the creation of checkpoints. Particularly, we focus on the basic protocol and the improvements made to efficiently support process migration.

1 Introduction

Researchers from many different areas have needs for computational power to solve their specific problems. Today, many are successfully using PVM[4] on Networks of Workstations (NOW) [1] to satisfy their requirements. A PVM application uses the aggregate computational power of several workstations to speed-up the computation of a single problem. PVM has been used to parallelize a great variety of applications like genetic sequence analysis, semiconductor simulation or air quality modeling.

Usually, in a NOW a single workstation is assigned to a single *primary user (owner)*: the workstation can typically be found on this user's desk. Other users (*secondary users (guests)*) may access this machine via the network. Psychologically, the owner of the workstation expects that his workstation is at his disposal all the time. Whenever he wants to work interactively, he expects quick and immediate responses of his machine.

Running a PVM application on a NOW requires executing processes on many machines as a secondary user. Hence, a PVM application borrows resources such as computational power or main memory from other primary users. A single process of a PVM application is typically bound to the machine where it has been started for its lifetime, primary users may find their system in a state where interactive work is slowed by secondary users. As a consequence, they refrain from making the resources of their machines available to other users. For users, it becomes more and more difficult to find workstations on which to run their PVM applications. The situation becomes worse with an increasing number of users parallelizing their application using PVM. Hence, a facility is needed that maintains the ownership of the machines: as soon as interactive work begins on a machine, all processes which are borrowing resources must vacate. The mechanism for doing this are consistent checkpoints and process migration.

2 CoCheck: Checkpoints and Process Migration

Our protocol to create consistent checkpoints described in [3] proved to be viable and stable but was lacking good performance. This was due to several reasons. First, the checkpoint files which

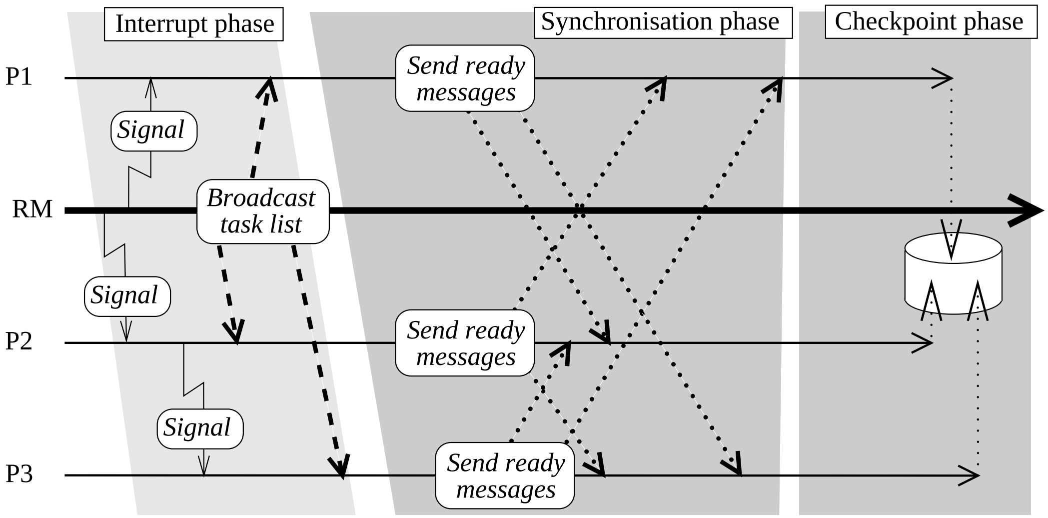


Figure 1: Steps to create a consistent checkpoint in CoCheck

were produced by the single process checkpointer were too large. Then the migration of a single process required that all processes had to write a checkpoint file. Finally, all checkpoints were written to a network file system producing burst network traffic.

Hence, in the latest version of the CoCheck checkpointer for PVM applications we have addressed those problems to improve the performance. First, we have incorporated the latest version of the Condor single process checkpointer. It reduces the size of the checkpoint file of each process [5]. Then, the protocol has been enhanced to allow that checkpoint files can also be written to local discs and that only migrating processes write a checkpoint file. Finally, we now use the resource manager interface of PVM 3.3.x to register an additional system task: the resource manager (RM). The RM task intercepts certain PVM calls such as adding or removing hosts or spawning tasks. It is responsible to carry out the requested services instead of the PVM daemons. Hence, it is possible to introduce a modified behavior of the intercepted PVM calls.

Both the creation of a consistent checkpoint of an application and the migration of a set of processes is initiated by the RM of the virtual machine by delivering a combination of a signal and a message to each process of the PVM application (cf. Figure 1). As the wrappers for the PVM functions guarantee that the PVM calls cannot be interrupted by a signal, it is possible to use PVM functions within the signal handler. PVM calls such as `pvm_recv` present a problem because they cannot safely be interrupted by a signal. Here, the message which is sent together with the signal can be used to force the interception of the signal. The wrapper functions for the blocking PVM calls have been implemented in that way. In addition this message contains information about all tasks of the application, which tasks need to create a checkpoint file, the location of the checkpoint file and if the process should exit. The location can either be a file on a local or network file system or a network address. The network address can be used in case of process migration, where the checkpoint can be directly transfered to the new process on another host. Exiting is only necessary if a process must vacate a machine.

After the signal and the task list have been intercepted all the tasks send specially tagged “ready” messages to each other. As PVM guarantees the delivery of the messages in sending sequence each task can draw the following conclusion: after all ready messages from all other tasks have been received there are no more outstanding messages on the network for this task. Hence, it can be seen as a normal stand alone process without any network state. While each process waits for the reception of the ready messages, it collects all other incoming messages in its address space, so that they become part of the checkpoint file when it is produced and can be retrieved after restart from the wrapper functions.

After each process has collected all the ready messages it disconnects itself from the PVM

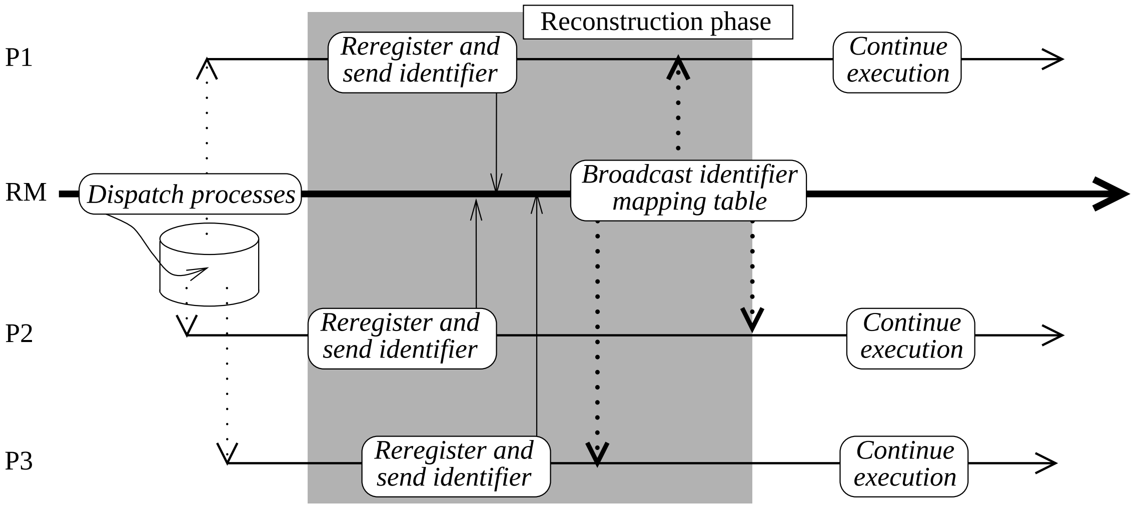


Figure 2: Restarting from a checkpoint.

system and uses the Condor single process checkpoint library to create a checkpoint. Depending on what was specified in the task list message the file is either written to disk (local or via a remote file system) or its state is transferred directly to a remote process which will become the migrated process. If the checkpoint has to vacate the machine it exits after the creation of the checkpoint is complete. Otherwise it directly begins the restart protocol (cf. Figure 2).

At the beginning of the restart phase all processes are stand alone processes. As the first step to continue the parallel PVM application, each process rejoins PVM and gets a new task identifier (tid). This tid is then sent to the RM which can set up a tid mapping table from original tids to current tids. As the application should not be aware of the fact that a process has migrated or a checkpoint was taken and the application has been restarted from that checkpoint, the tids which each task got, when it enrolled for the first time, are used throughout the lifetime of the whole application. To achieve this transparency of the tids the wrappers use a mapping table each time a tid is referenced in a PVM call. This mapping table is set up and distributed to all tasks by the RM. After each task has received its copy of the mapping table it can continue normal execution. The wrapper functions take care of retrieving messages that have been stored in the restored user address space before other incoming messages.

As already mentioned one requirement is to have a signal safe extension of PVM. CoCheck achieves this by providing wrapper functions for all PVM calls which block signals. This introduces an additional overhead of about 2 percent during normal operation for sends and receives as shown in figure 3. The ping-pong measurements were done on two Sparc 10 machines connected by a nearly unloaded Ethernet during nighttime.

3 Conclusion and Future Work

The latest version of CoCheck allows for creating checkpoints of PVM applications. Due to the enhancements discussed in this paper CoCheck now also provides a means to efficiently migrate single processes. This has been achieved by restricting the creation of checkpoint files to the migrating processes. Local discs can now be used to store those files or the checkpoint can be directly transfered over a TCP connection to the new host of the process.

The CoCheck environment comprises a set of libraries, a simple RM for PVM and a program to use CoCheck's functionality. Applications which should benefit from CoCheck's features simply need to be relinked against the CoCheck libraries. The user can then use the supplied command to either create a checkpoint, restart an application or migrate certain processes. The CoCheck

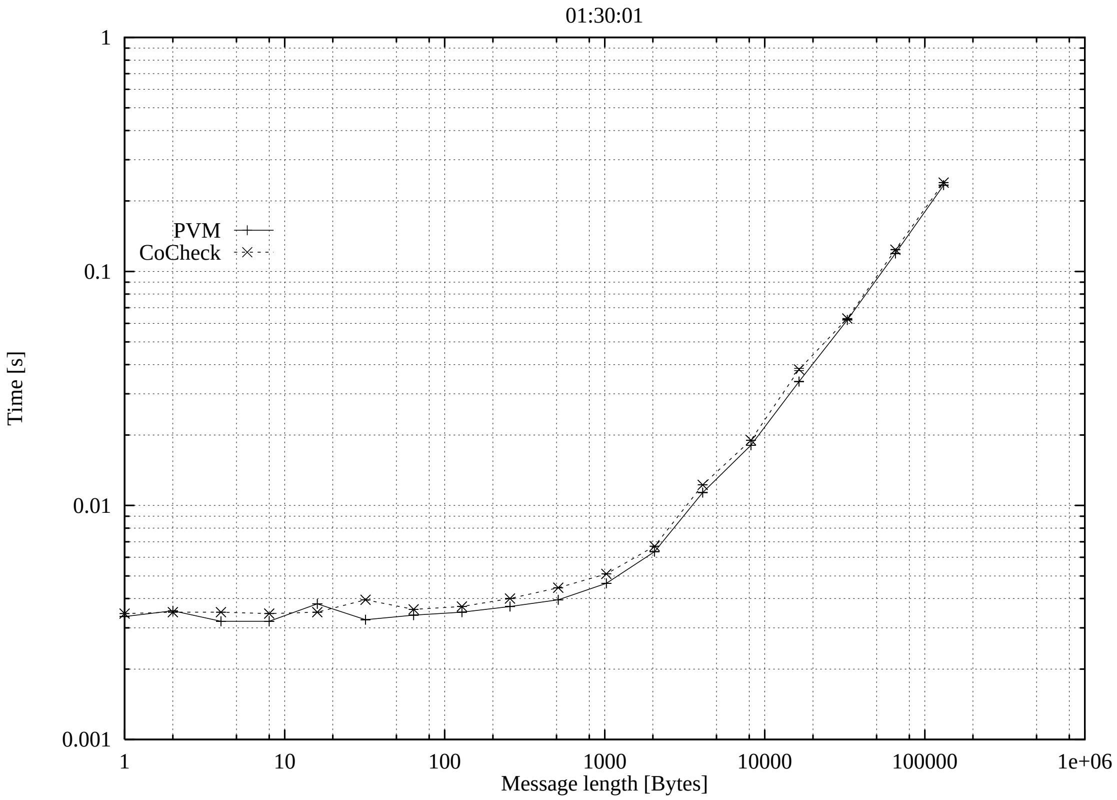


Figure 3: Performance of the CoCheck wrappers compared to PVM 3.3.7

system is publicly available under the terms of the GNU public license. To obtain the software refer to the CoCheck homepage: http://wwwbode.informatik.tu-muenchen.de/~stellner/CoCheck/.

In addition to the supplied control command CoCheck offers an API with which the users can request CoCheck functionality within their applications. This includes the creation of checkpoints or the migration of processes. Hence, the user is capable of integrating dynamic load balancing into his applications, where the application can request to migrate a set of processes to new hosts, which are more powerful or less loaded. Together with the PVM feature of notifications, CoCheck can be used to detect failures of machines and restart the application from a formerly created checkpoint.

CoCheck provides a preemptive global scheduler to RM processes. Hence, in the future we will focus on using this service in the RM. In addition to the use for RM, CoCheck will be incorporated in THE TOOL-SET[2]. Apart from providing basic checkpointing, CoCheck will be used within THE TOOL-SET to provide the core migration facility for system integrated dynamic load balancing. The debugger of THE TOOL-SET will use CoCheck's checkpointer to provide cyclic debugging, i.e. during a debugger session the user can create a checkpoint and restart the program from that state all over again to examine it.

References

[1] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] Thomas Ludwig, Roland Wismüller, Rolf Borgeest, Stefan Lamberts, Christian Röder, Georg Stellner, and Arndt Bode. THE TOOL-SET — an integrated tool environemnt for PVM. In

Proceedings of the 2nd European PVM Users' Group Meeting (Short Papers), Lyon, September 1995.

[3] Georg Stellner. Consistent Checkpoints of PVM Applications. In *Proceedings of the 1st European PVM Users Group Meeting*, http://www.labri.u-bordeaux.fr/~desprez/CONFS/PAPERS/abs010.ps.gz, 1994.

[4] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, 20(4):531–545, April 1994.

[5] Todd Tannenbaum and Michael Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, (2):40–48, February 1995.

NXLib – NX Message Passing on Workstations*

Stefan Lamberts
LRR-TUM
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik
Technische Universität München

Abstract

NXLib is a library and programming environment for workstations which resembles the parallel programming environment of Intel's Paragon Supercomputer. This article will give a short overview of our motivation for NXLib, its implementation, and show some performance figures.

1 Motivation

State-of-the-art multicomputers like the Paragon offer a proprietary message passing environment. An implementation of that on coupled workstations allows the use of interconnected workstations as a development platform for applications. In addition to that, it is also attractive to use interconnected workstations as additional computational resource. The performance constraints of coupled workstations restricts this to applications with limited demands concerning message latency and bandwidth, which usually are applications with coarse grain or medium grain parallelism.

In contrast to other parallel programming environments like PVM [5] and P4 [1], NXLib resembles a native message passing environment of a real supercomputer, the Paragon [2]. Thus, NXLib applications running on Intel's Paragon will take advantage of better communication performance of the native message passing calls. NXLib applications on workstation clusters will achieve about the same performance as applications which use other parallel programming environments [4, 3].

2 Implementation

2.1 Virtual Paragon Node

In the following discussion the term *Paragon Node* will be referred to as the collection of a hardware Paragon node, the operating system kernel and a set of application processes running on top of that.

The basic means to model Paragon nodes on coupled workstations is virtualization. Consequently, a *virtual Paragon node* (VPN) resembles a Paragon node on a workstation. The hardware and software properties of a Paragon node which are not available on a workstation are offered in the following way. We introduce a daemon process which takes over part of the functionality of the Paragon hardware and operating system. The calls of the application processes to NX routines are transformed into requests to the daemon.

*This project was partially funded by a research grant from Intel Corporation

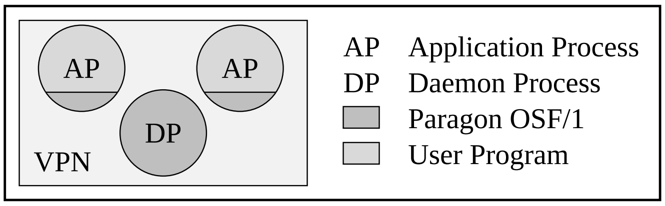


Figure 1: NXLIb's Virtual Paragon Node

In such an implementation every system call would require an interprocess communication between daemon and application process. To reduce the amount of interprocess communication, parts of the operating system's tasks have been moved into the application processes and will be linked to the application process. Figure 1 illustrates the structure of a virtual Paragon node.

2.2 Communication Layers

An important issue for a message passing library for coupled workstations is portability and flexibility. A layering of the message passing library has been designed to cover both aspects. Figure 2 shows the layers of the NXLib environment.

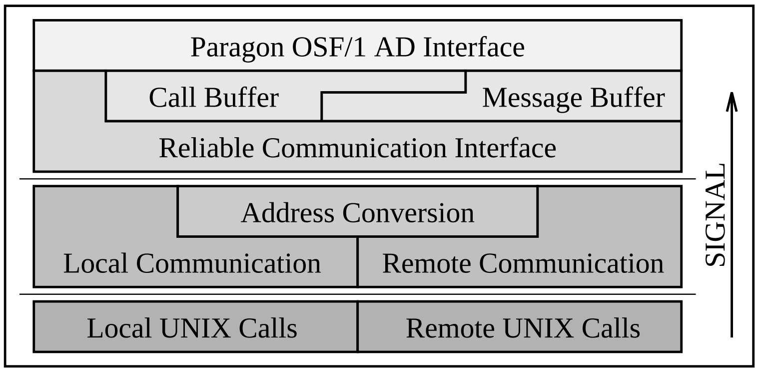


Figure 2: Communication Layers of NXLib

The basis consists of standard UNIX system calls. To achieve a great flexibility concerning the communication protocol which is used for the implementation, NXLib distinguishes between local and remote communication. Within the local and remote communication layer a protocol specific addressing scheme is used. The reliable communication layer provides reliable point-to-point communication calls disregarding the location of the communication partners. The reliable communication interface still uses the Paragon addressing scheme. The address conversion layer has been introduced to map Paragon addresses to corresponding protocol specific addresses. In addition to its address conversion task, this layer also distinguishes whether a communication is local or remote. Provided with that information, the reliable communication layer can invoke the appropriate local or remote communication calls.

We introduced a buffer layer which consists of the call buffer for pending send and receive calls and the message buffer for incoming messages. The latter is based on the simplifying assumption that on each machine unlimited buffer space is available. Unlike the Paragon, where incoming messages are placed in a reserved memory area, in NXLib the memory is dynamically allocated when a message arrives on a node.

The Paragon OSF/1 communication interface finally provides the user calls which are available on a Paragon system. Send and receive calls access the buffer layer by inserting the call in the call buffer or checking the message buffer for matching messages. NX calls which imply interprocess communication (e.g. global operations), access the reliable communication layer directly. Some calls of the Paragon interface, e.g. `dclock()`, will be handled immediately by the interface layer.

In order to provide Paragon's interrupt driven message passing, on arrival of an incoming message a signal is delivered to the receiving process. Within the signal handler NXLib receives and handles the message. Currently, local and remote communication uses the TCP/IP socket interface.

3 Performance

Figure 3 shows the message latencies for local communication (both application processes on the same workstation) and remote communication (application processes on different workstations) in a log-log-scale. The results were obtained by dividing the time of synchronous ping-pong operations between two processes using `csend()` and `crecv()` calls by two. The plots show maximum, average and minimum values obtained in 1000 iterations during normal working time. The values for remote communication with message sizes above 100 K-bytes were measured with only 100 iterations due to the very long execution times. We used an IBM RS/6000 3BT and an IBM RS/6000 390 both with 64 Mbytes of memory coupled via our local Ethernet.

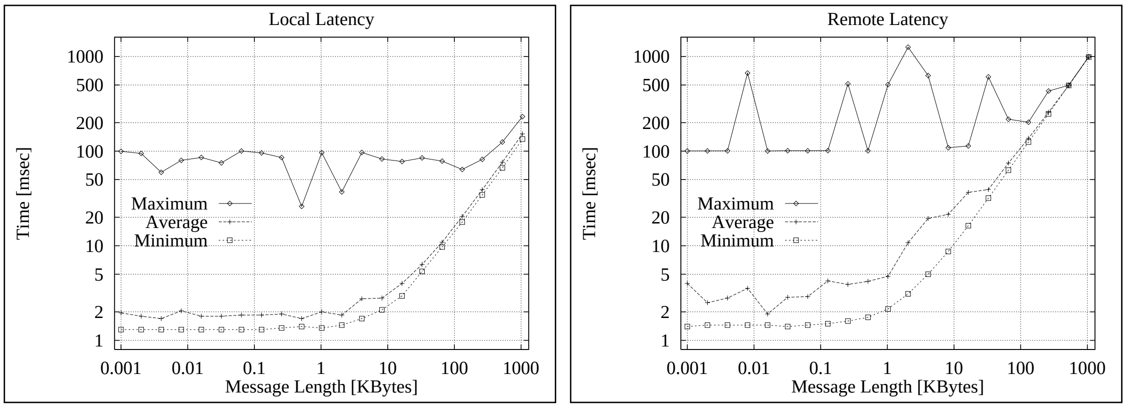


Figure 3: Local and Remote Message Latency of NXLib

The average 1 byte message latency is 1.95 m-sec for local communication and 4 m-sec for remote communication. Best 1 byte message latency is 1.3 m-sec and 1.4 m-sec respectively. The highest average bandwidth is achieved with the largest messages. In this measurement this was 6.9 M-bytes/sec locally and 1.06 M-bytes/sec remotely with a message size of 1 M-byte. For remote communication the best bandwidth value of all iterations increases only slightly if the message size is above 16 K-bytes where 1 M-bytes/sec is reached.

4 Conclusions

The NXLib environment allows the use of a network of workstations for mainly two purposes. First, the network of workstations can be used to develop software which should finally run on a Paragon system. Workload can therefore be withdrawn from the multicomputer system. The CPU time which is gained by shifting the development of applications to workstations can be used for production runs of computational intensive problems. Second, instead of using the workstations merely as a development platform they can also be used as a production environment for certain applications. Especially coarse grain applications can achieve good speed-ups in a workstation environment.

NXLib was made available to the public under the Gnu library license and the Gnu general public license end of 1993 for SunOS 4.1.x. In the course of 1994, NXLib was ported for HP 9000 HP UX 09.01, SGI IRIX 5.1.1.1, IBM RS 6000 AIX 2.3, DEC Alpha AXP OSF 1.2, and Sun Solaris 2.3. Thus, NXLib is now available for all major workstation platforms via anonymous ftp from `ftp://ftpbode.informatik.tu-muenchen.de/NXLIB/` and other ftp servers. We released four new versions of NXLib up to the current release V1_1_4 which included several bug fixes and ports to other platforms.

Work on an implementation which uses System V IPC shared memory facilities for local communication is almost finished. Additionally, we are developing an environment which offers the functionality of Paragon's parallel file system PFS on workstations.

References

[1] Ralph M. Butler and Ewing L. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.

[2] Intel Corporation, Supercomputer Systems Division, Beaverton, Oregon. *Paragon User's Guide*, 312 489-003 edition, June 1994.

[3] Stefan Lamberts, Georg Stellner, and Thomas Ludwig. *NXLib Users' Guide*. Technische Universität München, D-80290 München, v1_1_4 edition, February 1995.

[4] Georg Stellner, Arndt Bode, Stefan Lamberts, and Thomas Ludwig. Developing application for multicomputer systems on workstations. In Wolfgang Gentzsch and Uwe Harms, editors, *High-Performance Computing and Networking, Vol. II*, number 797 in Lecture Notes in Computer Science, pages 286–292. Springer-Verlag, April 1994.

[5] V. S. Sunderam, G.A Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, 20(4):531–545, April 1994.

PFSLib – Performing Parallel I/O in PVM Applications*

Christian Röder

Technischen Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
D–80290 München
e–mail: {roeder|lamberts|ludwig}@informatik.tu-muenchen.de

Abstract

In this paper we discuss the design and implementation of an I/O interface for parallel programming environments on clusters of workstations. The PFSLib library offers the functionality of Intel's parallel file system PFS and is intended to be used for parallel applications using PVM or other message passing libraries for workstation clusters. An example of a simple parallel matrix-vector-multiplication shows how PFSLib function calls are used together with PVM.

1 Introduction

During the last years parallel programming environments (PPEs) like PVM [8], P4 [1] or NXLib [7] became more and more popular in the parallel computing community. Those PPEs offer sophisticated mechanisms for managing processes of parallel applications and communication via message passing. If the interdependency of the processes and thus their need of synchronization and communication is low compared to the applications overall runtime a workstation cluster is a good alternative to supercomputers. But the development of an appropriate I/O interface follows a similar history as it was for real supercomputers where the primary design goals focused on increasing processor speed and fast interconnection networks. The I/O traffic was handled by a host computer producing a severe I/O bottleneck. Increasing the speed of I/O traffic to the attached hard disks was neglected for a long time. To solve the I/O requirements of scientific applications vendors like Intel and nCUBE offered dedicated I/O subsystems for their parallel computers, which are able to perform parallel I/O operations concurrently on a set of disks. A similar situation arised when PPEs for coupled workstations were under development. There, two major paradigmas were used for I/O in parallel applications. Firstly, one dedicated process is responsible for all read and write accesses to files, thus distributing and gathering all data to or from the other processes via messages. Secondly, every process within in the application accesses local files confirming its data. Those files have to be post-processed for further usage. Current research activities in the field of parallel computing show that scalable I/O and parallel file systems are one of the major topics of interest.

In 1993 the parallel processing group at the LRR-TUM developed an emulation of the Intel Paragon Supercomputer. The latest release of this parallel programming model for coupled workstations called NXLib[1] is available since February 1995. The NXLib library comprises message passing and process management routines. It allows to run early software development phases on

*This project was partially funded by a research grant from Intel Corporation

[1] Release V_1_1_4 can be accessed via URL: ftp://ftpbode.informatik.tu-muenchen.de/NXLIB

a workstation cluster and to switch to the Paragon for final production runs only. Like in most others PPEs the lack of a sophisticated I/O facility restricts programmers in writing applications which operate on large amounts of data. The new library PFSLib adds the functionality of Intel's parallel file system. One of the main goals in designing and implementing PFSLib was to be independend from the native message passing library and thus having a universal I/O interface for all PPEs.

2 Related Work

Available results taken under consideration when designing and implementing the PFSLib can be grouped into three different categories:

- Operating system integrated approaches for remote handling of I/O on clusters of coupled workstations (e. g. NFS, AFS, DFS).
- Existing approaches for parallel I/O for workstation clusters (e. g. PIOUS [6]).
- Definitions of user interfaces for parallel I/O (e. g. MPI-I/O [2], Intel PFS [4], IBM Vesta [3])

PIOUS (Parallel Input OUtput System) is an existing approach for workstation clusters and was presented in 1993 by Moyer and Sunderam. In PIOUS process groups on heterogenous clusters access common data on peripheral storage devices. The following issues are covered in the first version of PIOUS:

- Portability to other systems is achieved by the independence of the underlying transport mechanism. PIOUS only requires reliable data transfer mechanisms between cooperating workstations.
- Parallelism of the system results from asynchronous execution of the operations.
- Scalability is guaranteed by storing files in a distributed manner.
- Integrated control of parallelism and mechanisms for fault tolerance through the replication of data.
- An optimal structure of a parallel file system is currently unknown. Therefore, PIOUS has a flexible design that allows modification to the user interface and the internal data structures.

For a more detailed description of the implementation and a performance analysis of PIOUS see [6].

Although it is still not yet clear which concepts have to be integrated, available interfaces offer different I/O modes, such as how processes participating in parallel I/O access parts of a file. With MPI-I/O it was suggested to treat I/O as a special kind of message passing. However, by adhering to the syntax of message passing calls an I/O library can only be used with one individual message passing library. PFSLib provides much more flexibility in that point.

3 The User Interface of PFSLib

The main concepts of parallel I/O concern file access patterns and the semantics of read and write calls. As mentioned above the user interface of PFSLib is in most parts identical to that of Intel's parallel file system PFS. Due to the different architectural concepts of clusters a few restrictions have to be made.

File access patterns are selected by specifying one of five I/O modes, which can be set, queried, and changed by using library calls. To gurantee correct semantics obviously all processes involved in accesses to a specific file must use identical I/O modes. The modes are just attributes indicating how the file accesses are ordered but they are not stored within the file. Every new `open()` call is independent from previous operations on the file.

M_UNIX provides each process with its own file pointer and it is the programmer's responsibility to perform reasonable read and write calls. This mode is best used when processes operate on large disjoint segments of the file. All read/write requests are served immediately.

M_LOG provides a single file pointer for all processes. Each I/O operation modifies the file pointer of all processes. This mode is preferable to write log-files.

M_SYNC provides a single file pointer for all processes. The operations are ordered by the logical number of the calling process. Thus, all processes have to perform the same sequence of calls. This mode is especialy useful when writing ordered lists or checkpoint information to disk.

M_RECORD is similar to M_SYNC but gives each process its own virtual file pointer. The data is read or written in records of equal size. Still the processes stick to the same sequence of calls but in contrast to the above mode they are no longer synchronized when actually performing them.

M_GLOBAL provides read access in a broadcast manner where all processes read the same data, e.g. in the startup phase of the application. Write access will write the data of only one arbitrary process to the file. If the contents of written data is validated by some other means this gives a much higher performance than every process writing the same data.

The concepts of reading and writing to parallel files follow the principle of message passing between processes. However, there is no syntactical or semantical dependency with the NX message passing calls. This is a prerequisite for PFSLib to be usable with other programming environments.

Two exisiting variations for read and write calls are blocking and nonblocking calls. With blocking calls the data is already read or written when the call returns whereas with nonblocking calls the user has to check for completion of the call. In addition to reading and writing sequences of bytes the user interface supports operations for scattered data structures.

Additionally, the user interface comprises various calls for opening, closing and advanced management of files.

4 Design and Implementation Aspects

Following one of our main design goals PFSLib works together with major environments like PVM, MPI [9], and NXLib. Neither do we rely on any semantical information from the environment nor is PFSLib based on a specific message passing subsystem. To achieve this we consider that processes within the application implicitely form a group. This is neccessary to ensure correct semantics for M_SYNC and M_RECORD mode where access to the same file is ordered by the identifiers of the processes within the application. Additionally the independence from the message passing system is achieved by using the remote procedure call mechanisms provided by the UNIX operating system. Finally, the complete functionality can be accessed via C and Fortran user interfaces.

The current implementation of PFSLib has a centralized server for managing file access and does not yet incorporate mechanisms to distribute single files to several disks. However, for large amounts of data being accessed the data transfer is executed by an extra server. This concept is flexible enough to be extended to mechanisms for data distribution. The design aspects of this version cover the following issues:

Internal states of server and clients: As opposed to NFS the PFSLib server is not stateless. Its activities depend for several modes on the sequence of read and write requests transferred by the processes (clients) to the server. In the current implementation the server keeps information on both connected clients as well as open files.

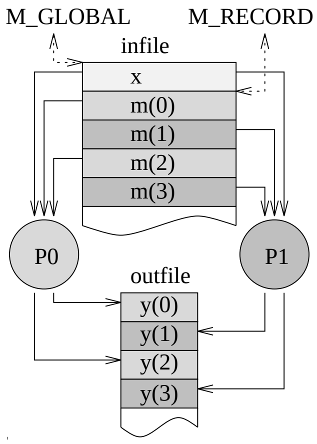
File access strategy: All read and write operations are currently managed by the server. Hence, filesystems with and without NFS can be treated identically. If the amount of data exceeds a configurable threshold the file access is handled by a so-called IO-Server. In case of an asynchronous operation the client process forks and the child process carries out the file access. Unix IPC shared memory is used for the data transfer between client and its child process.

Course of an operation: 1.) The client checks whether the operation accesses a file controlled by the PFS-Server or a Unix file. If the file is a Unix file the appropriate standard library call will be executed and the call returns. 2.) In case of a PFSLib file the client increases the RPC timeout if the operation is synchronizing. This is necessary because the answer of the server might be delayed until all other processes issued the same operation. 3.) The client sends a RPC request to the server. 4.) The server carries out several security and parameter checks. 5.) The server handles the request and sends the result back to the client. 6.) The client analyzes the response and the call returns.

Client synchronization: In the case of synchronizing calls (e. g. in I/O-mode M_SYNC) the call can not be completed unless all participating processes have performed that call. Hence, the server delays the completion of the corresponding RPCs.

5 Using PFSLib for PVM Applications

In this section we want to show the ease of using PFSLib in a parallel application based on PVM performing a matrix-vector-multiplication $M \times \vec{x} = \vec{y}$.




```
client()
{
 int fd, cnt;
 int mynum = pvm_mytid();
 int total = pvm_numtids();
 pfs_init("pfsdhost", total, mynum);
 fd = gopen("infile",O_RDONLY,M_GLOBAL);
 cread(fd, x, n);
 setiomode(fd, M_RECORD);
 for (cnt=0; cnt <(n/total); cnt++) {
  cread(fd, m[cnt*total+mynum], n);
  /* calculate and store results */
 }
 close(fd);
}
```

Figure 1: Two processes reading parallel from a file and the corresponding code

Let k be the number of processes participating in that application and let n be the dimension of the components with $n \bmod k = 0$. The input file contains a vector $\vec{x}$ as the first entry and the matrix M row by row as the second entry. Setting the M_GLOBAL mode and reading the first

n numbers from the file every process will get the vector $\vec{x}$ as the result. Now the mode changes to M_RECORD. Subsequent read calls will access row $(i-1)*k+j$ where i is the number of calls and j is the identifier of the process within the application. Thus every process calculates the component $(i-1)*k+j$ of the output vector $\vec{y}$ with each step. The vector may be written to another file also using the mode M_RECORD. No synchranization of clients or their performed calls is needed. Figure 1 shows this scenario for two processes.

6 Future Work

In the future we will improve the efficiency of PFSLib by modifying some of the internal concepts of implementation. Our main goal is to achieve scalability. Hence, we will introduce a concept of distributed servers which will access files distributed over several workstations in parallel. Furthermore, we will enhance the user interface in order to study the usefulness of different access modes. In addition PFSLib will be incorporated within *The Tool-Set* [10].

References

[1] R.M. Butler and E.L. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20(4):547–564, April 1994.

[2] P. Corbett, D. Feitelson, Y. Hsu, et al. MPI-IO: A parallel i/o interface for MPI version 0.3. NAS Technical Report NAS-95-002, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA, January 1995.

[3] P. Corbett, D. Feitelson, J.-P. Prost, and S. Johnson Baylor. Parallel access to files in the Vesta file system. In *Proc. Supercomputing '93*, pages 472–481. IEEE Computer Society Press, November 1993.

[4] Intel Corporation, Supercomputer Systems Division, Beaverton, Oregon. *Paragon User's Guide*, 312 489-003 edition, June 1994.

[5] S. Lamberts, T. Ludwig, C. Röder, and A. Bode. PFSLib — A file system for parallel programming environments. SFB-Bericht, SFB 0342, Technische Universität München, 80290 München, Germany, 1995. To be published in summer.

[6] S.A. Moyer and V.S. Sunderam. PIOUS: An architecture for parallel i/o in distributed computing environments. In *Workshop on Cluster Computing*, Tallahassee, FL, USA, December 1993. Florida State University.

[7] G. Stellner, A. Bode, S. Lamberts, and T. Ludwig. Developing application for multicomputer systems on workstations. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking, International Conference and Exhibition, Volume II*, no. 797 in LNCS, pp. 286–292. Springer, April 1994.

[8] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences and trends. *Parallel Computing*, 20(4):531–545, April 1994.

[9] D.W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, April 1994.

[10] T. Ludwig, R. Wismüller, R. Borgeest, S. Lamberts, C. Röder, G. Stellner and A. Bode. *The Tool-Set* – an integrated tool environment for PVM. Submitted for the 2nd European PVM Users' Group Meeting, 1995

SPTHEO –
a PVM-based Parallel Theorem Prover

Christian B. Suttner

Institut für Informatik
TU München, Germany
Email: suttner@informatik.tu-muenchen.de

Abstract

SPTHEO is a parallelization of the sequential theorem proving system SETHEO, based on the SPS model for parallel search. The SPS model has been designed to allow efficient parallel search even in comparatively low bandwidth and high latency environments, such as common workstation networks. In order to obtain a portable and efficient implementation, the PVM message passing system has been used for implementing the communication part of the system. This report describes the basic outline of the system, and presents evaluation results for the communication aspects as well as the performance as a proof system.

1 Introduction

The goal in automated theorem proving (ATP) is, given a set of axioms and a sentence to be proven, to show that the sentence is a logical consequence of the axioms. This general paradigm can be used in many applications, such as circuit verification [BCMD90], program synthesis [SWL$^+$94], and software and protocol verification [CGH$^+$93, Sch95].

ATP is based on search: the solution to a problem is found by a systematical trial-and-error of possible combinations between inference rules, axioms, and deduced formulas. A common way to describe a search space is the OR-search tree. Figure 1 gives an example for an OR-search tree. The top node denotes the original problem (start situation), from which alternative search paths extend. In theorem proving, the OR-branches result from different clauses that can be used to solve a particular subgoal.

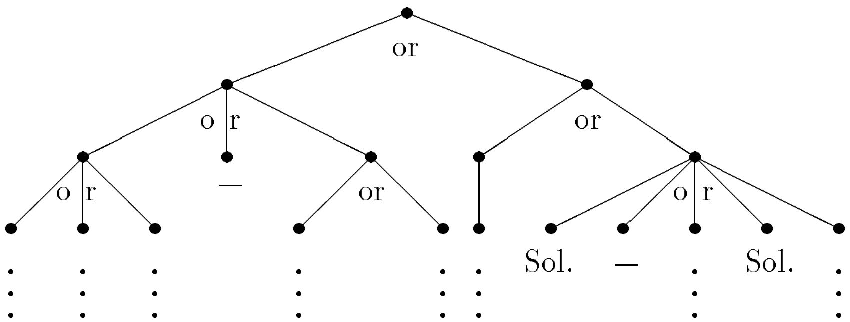


Figure 1: OR-search-tree.

The predominant problem in theorem proving (and search in general) is the performance limitation that arises due due to a combinatorial explosion of the search space: if a proof consists of many individual steps, the number of possible paths becomes intractable for exhaustive search. Even though advanced systems use various search pruning and shortcut techniques, many interesting problems require enormous computing time or are well beyond tractability.

In order to improve the usefulness of ATP systems in practical applications, a significant performance increase, esp. with respect to solving harder problems, is required. An approach to achieve such an improvement is the use of parallel instead of sequential search. Parallelism offers two advantages in that respect. First, the use of hundreds of processors significantly increases the hardware resources, both in computational power and in main memory capacity. This allows up to a linear reduction in the runtime of problems and makes problems tractable which previously were out of reach due to memory requirements. Second, and even more importantly, parallel search allows the avoidance of early bad search decisions. The exploration of several paths in parallel ensures that (viewed from the top of the search tree) at least one process is guaranteed to follow the best search path initially. This can lead to an exponential reduction in the amount of search required, compared to sequential depth-first exploration of the search tree. In sequential search, a similar reduction can only be achieved with breadth-first search, a technique that proved undesirable in practice due to memory requirements that are usually too large for a single processor and also the associated system complexity.

A thorough examination of previous work in parallel automated theorem proving [SS94] has lead to the establishment of guidelines which seem necessary to ensure the construction of a parallel search system which lasts for more than one hardware generation and can adopt improvements in sequential search technology adequately. These guidelines are:

- coarse-grain parallelization
  → minimize the synchronization overhead
- start with good sequential systems
  → employ their sophistication and functionality
- use powerful processors
  → to compare well against sequential system on the best workstation
- avoid specialized hardware
  → increases lifetime
  → increases portability
- simple model for parallelization (esp. wrt. communication)
  → good for implementation/maintenance
  → sequential search improvements easier to incorporate

The SPS (Static Partitioning with Slackness) model [Sut95] provides a parallelization model which follows these guidelines.

2 The SPTHEO System

The Underlying Sequential System. SPTHEO (Static Partitioning THEOrem prover) is based on the SETHEO (SEquential THEOrem prover) system [LSBB92, LMG94, GLMS94]. SETHEO is a sound and complete for first order predicate logic. It is based on the model elimination calculus (similar to PROLOG) and implemented as an extended Warren Abstract Machine [War83, Sch91].

The Computational Model of SPTHEO. The execution of SPTHEO is based on the SPS-model [Sut95, Sut96], and can be separated into three phases as shown in Figure 2. Briefly, tasks are generated in an initial, sequential search phase. These tasks are then distributed and executed in parallel. The motivation for this is to minimize the communication overhead, while avoiding load imbalance by proper task generation and by supplying more than one task to each processor. Below the individual phases are described in more detail.

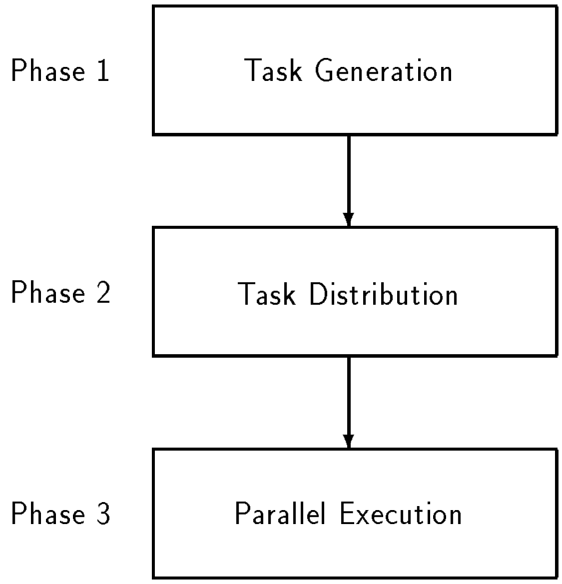


Figure 2: The computational phases of SPTHEO.

The Task Generation Phase. In the first phase (task generation), an initial, finite segment of the search space is explored. During this phase, tasks for parallel execution are generated. Task generation occurs only during this phase and independently of the processing of the tasks later (static partitioning). The number of generated tasks (denoted by the letter m) is equal to or larger than the number of processors (denoted by the letter n). The term *slackness* is used to express the relation between the number of tasks and the number of processors.

The task generation phase is defined by the rules used for partitioning the search space into tasks, a generation strategy, and the desired number of tasks. SPTHEO allows OR- and independent-AND partitioning (iAND-partitioning for short). With OR-partitioning, a separate task is created for each alternative clause that can be used to solve the current subgoal. With iAND-partitioning, a separate task is created for each group of independent subgoals that remains to be solved. The generation strategy used is iterative-deepening search over the depth of the proof tree. That means that for each particular deepening level, the search is pruned as soon as the resulting proofs would exceed a certain depth. The task generation thus operates as follows. The levels defined by iterative-deepening are successively explored, counting the number of search cut-offs due to reaching the depth limit. Each such point where the search has been artificially stopped can be used as a task for parallel search. As soon as a level is reached which allows to create enough tasks (m is specified by the user), task generation is enabled and the desired number of tasks is approximated by appropriately switching to a previous deepening level (where the number of tasks that remain to be produced is already known). For details about the generation control see [Sut95].

The Task Distribution Phase. In the second phase (task distribution), the tasks generated in phase one are distributed among the available processors. SPTHEO makes beneficial use of the availability of all tasks prior to their distribution: in the iAND-processing option, redundant tasks may occur, which are detected and eliminated. The task distribution affects the potential for load imbalance: if tasks with very long run time accumulate at one processor, while small tasks accumulate on another, imbalance results. Experiments showed that the cumulation of similar sized tasks is heuristically minimized, if the distance in the OR-search tree between the tasks mapped to the same processor is as large as possible. Further experiments [Hub93] showed that a very good approximation of the optimal solution is obtained by employing a simple modulo mapping: $\mathrm{Task}_i \rightarrow \mathrm{Prozessor}_{i \bmod n}$, where Task_i is the i-th task generated.

In order to keep the communication overhead low, an efficient task encoding is used. Since each task represents a location in the OR-search tree, simply the choices made during the search that lead to the task-defining position in the search tree are stored. This is simply a list of numbers, for example 3 1 ... for saying: take the third choice among the first set of alternatives, take the first choice among the second set, etc. From these numbers, each processor then recomputes deterministically the state that has been saved by the task, and proceeds searching for a solution from there. For iAND-partitioning, the tasks do not comprise full OR-search tree nodes, but are further restricted to solving a particular set of open subgoals within such an OR-search tree node. Therefore, the task encoding for an iAND-task consists of the previously described list of numbers plus a list of numbers that denotes which subgoals (e.g., the second and the fifth) have to be processed.

The Parallel Task Execution Phase. Finally, in the third phase (task execution), the tasks are executed independently on their assigned processors. Note that no interaction with other tasks is necessary in order to carry out a task. Since possibly more than one task has to be processed per processor, a service strategy is required. For this, quasi-parallel processing is preferable over serial processing. The reason for this is that a task may not terminate within a given runtime limit. With serial processing, this would lead to an infinite delay of all tasks remaining on that processor, which causes search incompleteness. Timesharing of tasks at a processor is realized by starting for each task a modified version of the SETHEO system (extended by the ability to receive and process a task) as a PVM process.

The Process Structure of SPTHEO. SPTHEO employs a simple master/slave process structure. A modified SETHEO process (master) performs the initial sequential task generation, including redundancy elimination for iAND-partitioning. The generated tasks are then one after the other spawned on the available processors. For this, a load-dependent performance index for all processors is established at the time the virtual machine is built. Tasks are then distributed in a round-robin manner, with processors ordered according to their performance index.

Each terminating task returns a result status to the master. For OR-partitioning, each task simply returns a status flag denoting success (a proof has been found) or failure (the search space has been completely exhausted with no proof found). As soon as the first success message is received, the master terminates all still active processes. Each task process terminates itself after its runtime limit is exhausted, and so does the master.

For iAND-partitioning, the same basic structure is used. However, for each success message received, the master checks if the solution of the respective task already provides a solution to the full problem, or if further open subgoals remain. In case further work remains, a message is printed summarizing the current proof advance. Regarding control, iAND-partitioning allows two options. Either all iAND-tasks are distributed in the beginning, or only a selected subset. Based on the membership of particular iAND-subgoals in different OR-nodes (the same iAND-subgoal can be part of several OR-nodes), partial orderings can be constructed which enforce as much parallelism as necessary, but as little as possible. This has been shown to be able to improve the efficiency of parallel search [Sut95]. Therefore, initially only a small set of tasks are started, and further tasks are spawned by the master whenever a success message is received and work remains to be done, depending on the partial task ordering.

3 Evaluation of SPTHEO

The SPTHEO system has been extensively evaluated using the TPTP library [SSY94, SS95]. Altogether, 2571 TPTP v1.1.3 problems were evaluated, thereby covering a broad application spectrum of 25 scientific domains, with problem difficulties ranging from very easy to very difficult (including open) problems.

In order to determine the performance improvement compared to the underlying SETHEO system and a previous parallelization, called RCTHEO (for Random Competition parallelization of SETHEO [Ert93]), both these systems have been tested as well. Since all three systems share most of their code (all are versions of the same program), the comparison is highly accurate in the sense that differences are mainly due to differences in the computational models, and not due to implementation differences.

All experiments have been performed on a network of 121 HP workstations connected by Ethernet. The network consists of 110 HP-720 workstations (58 Drystone MIPS, 38.5 SpecInt'92) with 32 MByte main memory each (200 MByte disk for swap space), and 11 HP-715/50 workstations (62 Drystone MIPS, 36 SpecInt'92) with 80 MByte main memory each (2GByte storage disk and two 500MByte disks for system and NFS data). Each HP-715/50 workstation operates as file server for 10 HP-720 clients. The network is partitioned with bridges into 5 segments with two servers each, and one segment with one server. In order to ensure that equivalent hardware is used in all experiments, no processes were run on server machines.

Runtime Assessment in a Distributed Environment. A major difficulty for the evaluation of a network-based parallel system is the influence of other users on the available processing capacity, both in terms of processor load and available communication bandwidth. As a result, the wall-clock runtime of a parallel job can vary significantly. Since an important purpose of the SPTHEO evaluation is an analysis of the performance of the SPS-model, such variations are undesirable. Ideally, exclusive usage of the network would be possible, allowing to obtain the true runtimes on such a network. However, measurements show that the employed network is highly utilized, even during the night and on weekends. Therefore, extensive experiments based on exclusive usage are not possible. Fortunately, both the SPS-model and the random competition model (RCTHEO) nevertheless allow an analysis which reliably approximates the results that would be obtained under exclusive usage. For SPTHEO, this is achieved by the following.

Instead of the standard SPTHEO operation where termination occurs as soon as a proof has been found, each task is independently processed until it fails, succeeds, or is terminated due to the runtime limit. The performance statistics for all tasks are then collected in a file. Also, at the beginning of this file the performance statistics for the task generation phase are included. These data provide a precise assessment of the number of search steps and runtime for each task and for the task generation.

Given these data, the runtime on some hardware platform can be estimated. Assume a particular task σ is the first task that leads to a proof in a standard SPTHEO run under exclusive hardware usage. The runtime of SPTHEO using OR-partitioning then consists of the runtime T_{gen} for task generation, the time $T_{dist}(\sigma)$ until σ is distributed, the runtime T_σ for processing task σ, the time T_{spp_delay} for the processing delay of σ due to time sharing, and the time $T_{termination}$ until the success message from σ is received and processed by the master[2] (runtime $= T_{gen} + T_{dist}(\sigma) + T_\sigma + T_{spp\_delay} + T_{termination}$). T_{gen} and T_σ are already contained explicitly in the logging file. T_{dist} can be upper bounded by the worst-case assumption that σ is always the last task distributed. In practice, for many problems a proof is reported to the master even before all tasks have been distributed. An assessment of the "location" of the successful task with the shortest runtime is shown in Table 7. It shows that in the case of many generated tasks ($m_{desired} = 256$), σ in (geometric) average appears after approximately $\frac{1}{4}$ of all tasks have been distributed. T_{spp_delay} depends on the number of tasks, the times of processing start, and runtimes of the tasks that are processed on the same processor as σ. Given a particular distribution scheme for tasks, the number and runtimes of these tasks can be extracted from the logging file. For an upper bound on T_{spp_delay}, it can be assumed that all these tasks start at the same time as σ. In case the task switching overhead due to quasi-parallelism is not negligible, measurements need to be performed. It is then straightforward to compute T_{spp_delay} from the logging data and measure-

[2] One may also add the time until the master terminated all tasks still being processed.

ments. Finally, $T_{termination}$ can be obtained by appropriate communication time measurements. In case no proof is found within the runtime limit, the runtime can be computed as above, based on the particular task σ which terminates last.

Averages	$m_{desired}$ = 16	256
Harmonic	0.17	0.04
Geometric	0.35	0.26
Arithmetic	0.53	0.50

Table 7: Average location of the task with the shortest runtime, given in terms of a fraction of the number of tasks. $m_{desired}$ specifies the number of tasks for task generation (specified by the user).

Figure 3 shows the wall-clock time required for distributing a single task (including process startup) using the PVM message passing library on the HP network under non-exclusive usage. The measurements have been obtained for SPTHEO, but equivalent values can be assumed for RCTHEO. It shows arithmetical average values for the average and maximal distribution time over 20 repetitions, for 20 different problems with different numbers of tasks to distribute. The average time for up to 20 tasks is approximately 0.1 seconds, with maximal values from 0.2 to 0.7 seconds. The figure shows a slight increase of the average value as the number of tasks increases, and a significant increase of the maximal value that is observed (here up to 2.1 seconds). The reason for this is that as the number of tasks increases, the probability increases that a processor with a high load is used. Such a processor responds slowly, leading to significantly increased maximal values for some tasks, which in turn leads to an increase in the average distribution time. The figure suggests that for exclusive network usage an average distribution time of approximately 0.1 wall-clock seconds can be expected, for at least up to 60 tasks.

Below the output regarding task distribution of an actual SPTHEO run is shown. The run has been performed during daytime with a high load on the network.

```
Entering PVM! Start Parallel Processing...
Spawned tasks:  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
Spawned tasks: 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
Spawned tasks: 45 46 47 48 49 50 51 52 53 54 55
Task  37 (iand-task) successful ... solves part of OR-node  31 (1 tasks left)
Spawned tasks: 56 57 58 59 60
Task  39 (iand-task) successful ... solves part of OR-node  32 (1 tasks left)
Spawned tasks: 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81
Spawned tasks: 82 83 84
Now waiting for a proof ...

Task distribution time (wall-clock): 67.40 seconds
Task spawning times    (wall-clock): min= 0.04 max= 5.27 average= 0.79 seconds
```

The example shows that two iAND-tasks (37 and 39) are solved before the task distribution is finished. The time required to distribute all tasks is quite high: 67 seconds. The reason for this is that some processors respond slowly (the maximal time required for a single spawn has been over 5 seconds) due to other load. The maximal time for a spawn observed in a run has been 191 seconds. High spawning times can significantly decrease the performance: while the particular slow-spawning task may not even be necessary for a proof, the processing of all tasks remaining to be spawned is blocked. This can be counteracted either by a modified (but more complicated) system design using hierarchical task distribution scheme, or preferably by a different spawning primitive which is non-blocking.

Proof-finding Performance of SPTHEO. The most relevant criteria in automated theorem proving is the number of problems that can be solved with given resources. For the comparison,

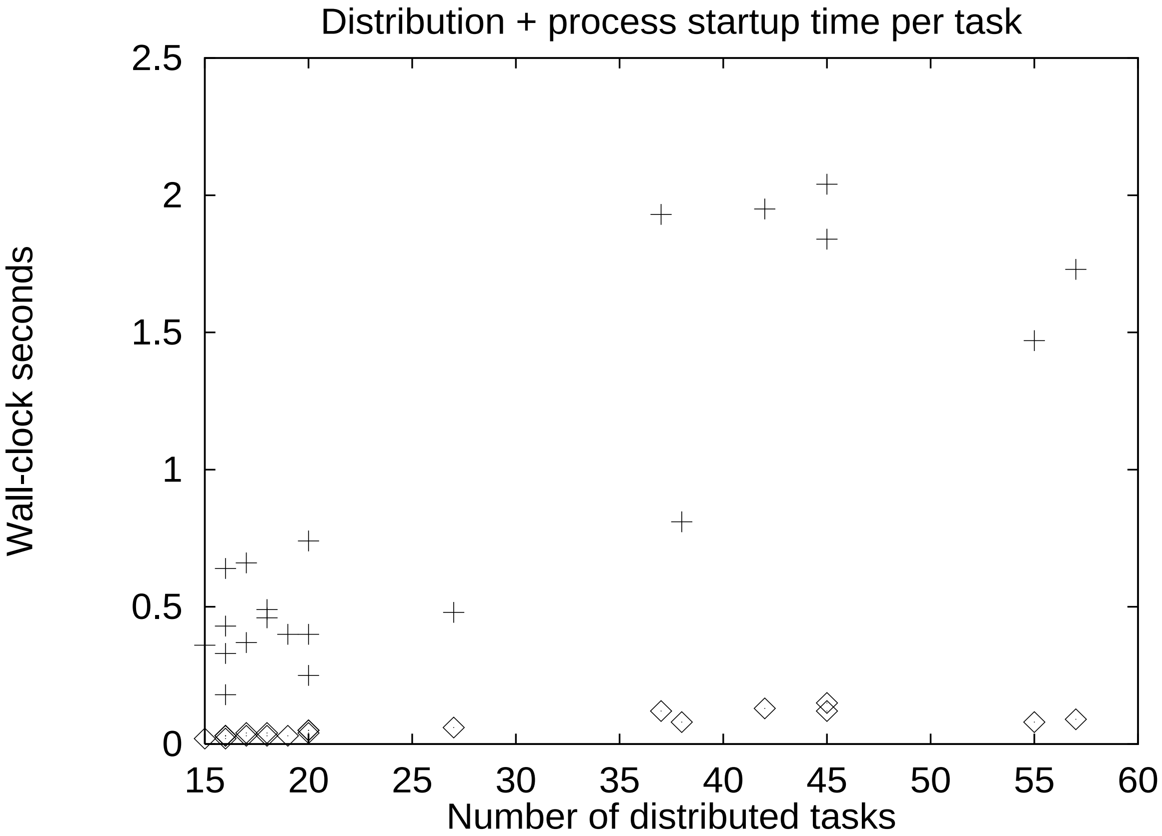


Figure 3: Distribution and process startup time per task, averaged over 20 repetitions for 20 different problems (one vertical pair of ⋄, + for each problem).
"⋄" denotes the arithmetical average, "+" denotes the maximal value observed.

a runtime limit of 1000 seconds ($\approx$ 17 min) for each problem has been used for SETHEO. For SPTHEO and RCTHEO, a runtime limit of 20 seconds per task has been used.

It is found that SPTHEO achieves a substantial performance improvement compared to both systems. As an indication of the achieved advance in ATP, with 256 processors (simulated) SPTHEO solves 167 more problems than SETHEO, and 150 more problems than RCTHEO. Neglecting all trivial problems in the TPTP (i.e., problems solved by the sequential system in less than one second), this represents an increase of 83% of solved problems. Even compared to the parallel RCTHEO systems (with the same number of processors), 69% more non-trivial problems are solved.

Speedup Performance of SPTHEO. Figure 4 shows the relative speedup for SPTHEO based on the number of search steps, for $m_{desired} = 16$ and $m_{desired} = 256$. It shows nearly linear speedup up to 16 processors, and an increasing deviation from linear speedup for larger n. It should be noted that the plots display the geometric averages; the arithmetic averages are much closer to being linear for large numbers of processors. There are several reasons why the relative speedup does not remain linear for large n. First, for $n = m_{desired}$, the slackness is close to one. In this case, the largest load imbalance is encountered, and some processors may become idle due to failing tasks. However, since failures occur rare in average, this has comparatively little influence on the average speedup. Second, since usually slightly more tasks are generated than specified by $m_{desired}$, there are cases where the number of tasks at the processor which first terminates successfully for some n is not decreased by using twice as many processors, due to an unfortunate task distribution. Inspection of some individual problems with low relative speedup indeed showed that adjusting the number of processors such that $spp = \frac{m}{n}$ are whole numbers (or as close as possible), better speedup is obtained. Third, and most importantly, the influence of the serial fraction given by the task generation overhead increases with decreasing spp. For large n, the number of search steps required to solve a task is frequently smaller than the number of search steps required for task generation, which limits the achievable speedup considerably. As discussed before, the relative speedup metric does not provide a useful judgment in these cases, because it ignores the absolute values. Reducing the task generation overhead would improve the relative speedup for large n significantly. However, since the task generation overhead is typically quite small in absolute terms, the actual performance of the system in terms of theorems provable under typical constraints or in terms of runtime (in the order of seconds) would not improve noticeably. This shows the danger of using a relative speedup for the evaluation of parallel systems: an improvement of the relative speedup for large n would be possible, but ineffective for the absolute system performance.

More interesting, in particular for potential users of the system, is a comparison of the actual runtime improvement that is achieved by SPTHEO. Figure 5 shows lower bounds for the geometric average of the absolute speedup of SPTHEO compared to SETHEO. The runtime measurements for this include the input overhead. Furthermore, the task distribution overhead is included based on the assumption that the task which terminates the computation is the $m/2$-th task distributed (where m is the number of generated tasks), and assuming a distribution time of 0.1 seconds per task. According to Table 7, the use of $m/2$ in this calculation provides a pessimistic average case assumption.

Figure 5 gives lower bounds on the average speedup because there is a large number of problems for which SETHEO did not find a solution (28%). The true speedup for these problems is therefore unknown, and the runtime limit of 1000 seconds is used as a lower bound of the SETHEO runtime. Therefore, the decreasing speedup slope noticeable for large n is not necessarily an indication of a performance saturation, but is significantly influenced by the runtime limit. Again, it may be noted that the arithmetic average speedup values are significantly larger in all cases. Moving from the left to the middle to the right plot shows a strong dependency of the speedup on the problem difficulty for SETHEO. Including many of the simple problems, low absolute speedup is achieved. This is little surprising, since the overhead of generating and distributing tasks cannot pay off for problems which require only a few seconds sequentially (the estimate of the distribution

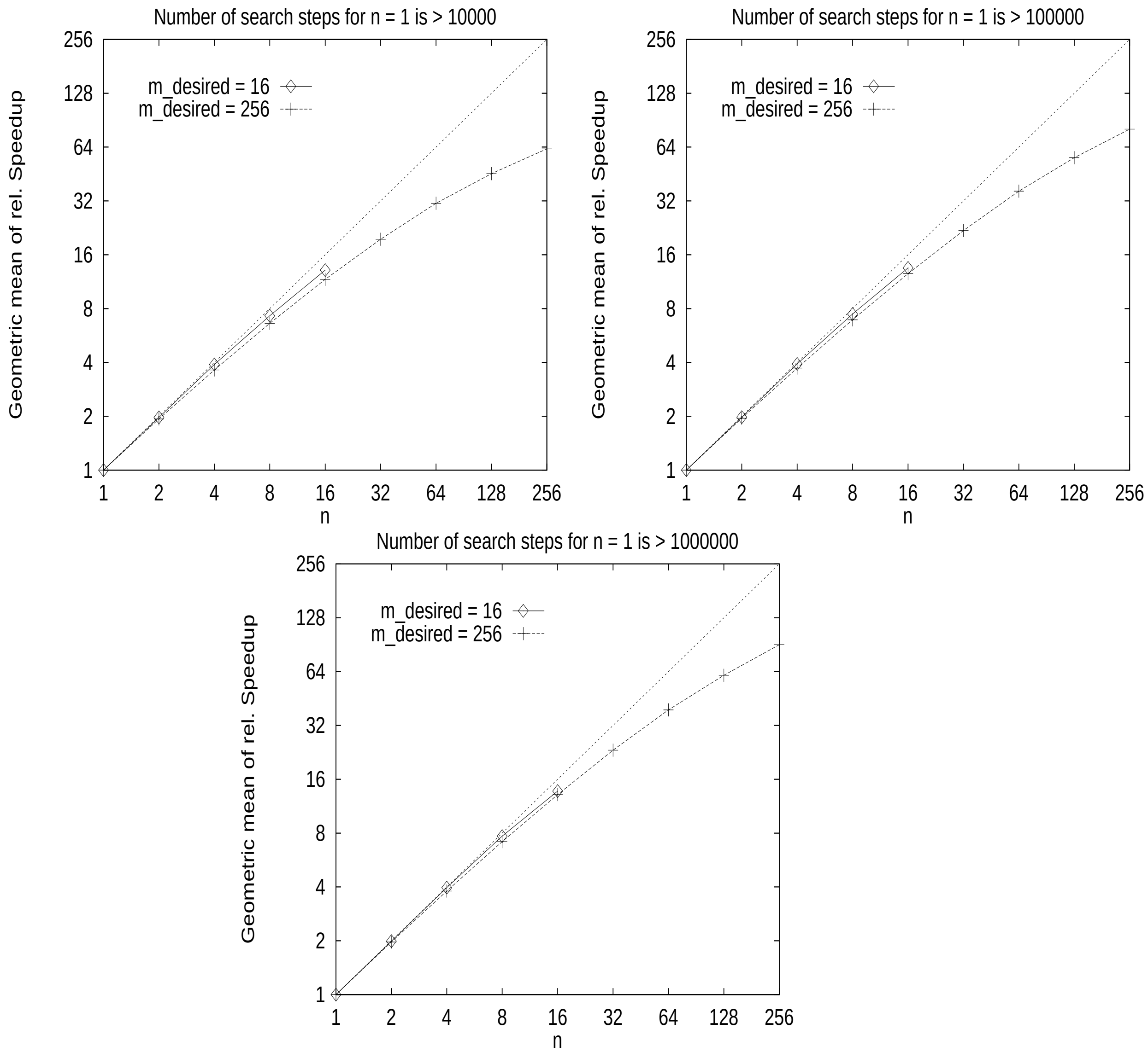


Figure 4: Geometric average of relative speedup of SPTHEO based on the number of search steps performed for different numbers of processors, for three problem collections based on the number of search steps required by one processor.

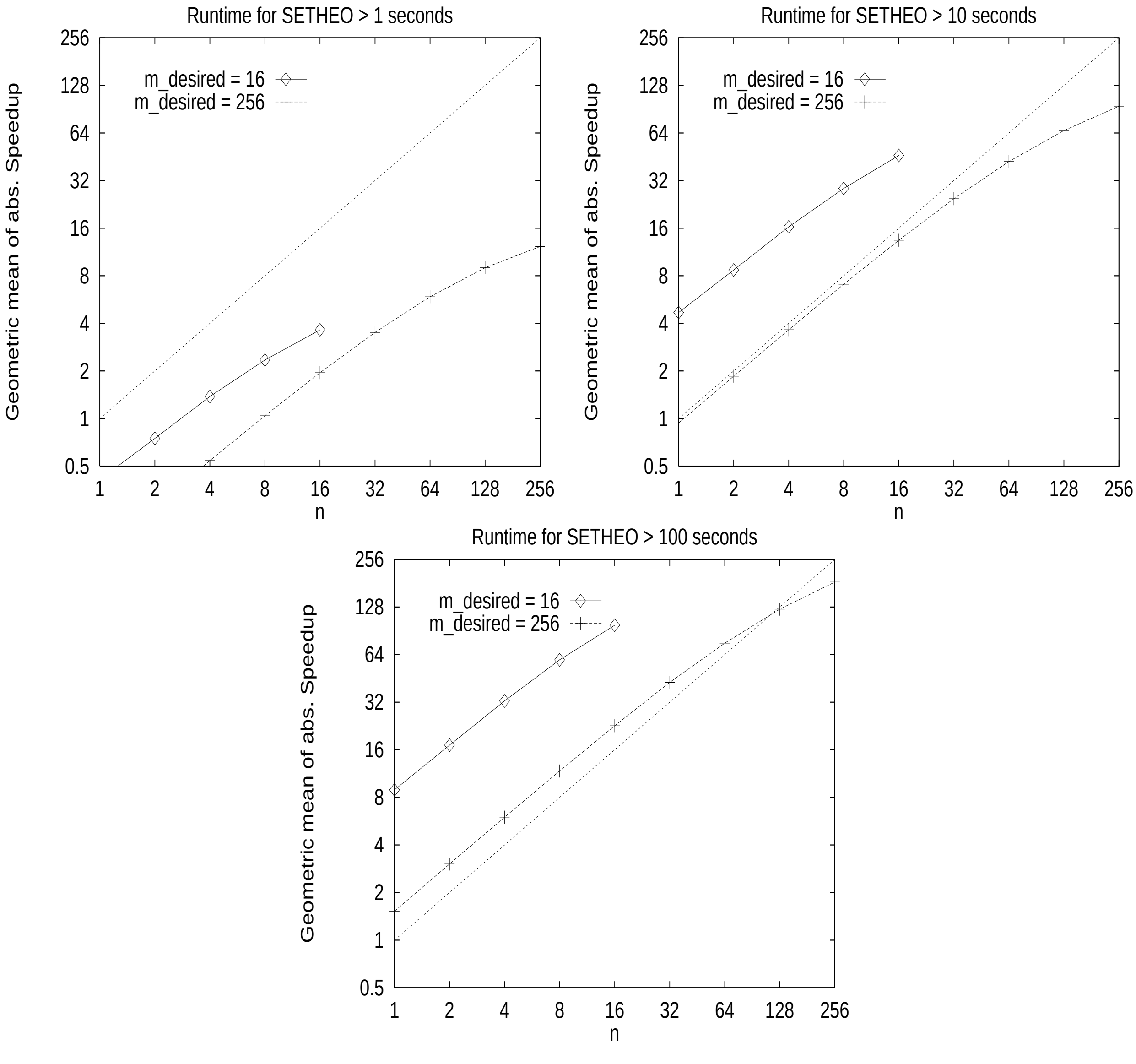


Figure 5: Lower bounds on the geometric average of the runtime speedup of SPTHEO compared to SETHEO including input and task distribution times, for three problem collections based on the runtime required by SETHEO.

overhead alone leads to a runtime of $128 \times 0.1 = 12.8$ seconds for 256 processors). For increasingly difficult problems (middle and right plots), the absolute speedup becomes superlinear in average (up to some n for $m_{desired} = 256$). The plots also reveal a significantly better performance for $m_{desired} = 16$ than for $m_{desired} = 256$ for an equivalent number of processors. This is mainly due to the task distribution overhead. For the applied estimate, the geometric averages are $T_{dist} = 1.1$ seconds for $m_{desired} = 16$ and $T_{dist} = 13.3$ seconds for $m_{desired} = 256$, regardless of n. These times are part of the serial fraction of the computation, and therefore reduce the speedup noticeably. Assuming faster communication hardware would therefore lead to a significant shift upwards of the curves for $m_{desired} = 256$, and a small shift upwards for $m_{desired} = 16$. This shows how the net improvement for some number of processors depends on the communication performance of the hardware. For good absolute speedup on systems with low communication performance, it is sufficient to use a comparatively small number of processors. But even for low communication performance large numbers of processors can be used successfully if sufficiently hard problems are treated.

4 Conclusion

In this paper, the SPTHEO system has been presented. SPTHEO is the first ATP system utilizing OR– and independent-AND search space partitioning. It is implemented in C and PVM and thereby provides a portable system running on most parallel hardware as well as networks of workstations. An evaluation has been presented which is based on extensive experiments performed on a network of 121 HP-Workstations. At the time of its construction, this represented the largest PVM application (in terms of workstation nodes) in operation.

The availability of a message-passing library significantly simplified the implementation of the parallel system. With respect to the further development of such libraries, three issues were noticed that would improve their use in SPTHEO-like systems:

- best processor spawn
  Spawn the next process on the most powerful processor available (this issue is also mentioned in the PVM FAQ 2.19).
- non-blocking spawn
  Do not wait for a process spawn to finish, but continue working immediately. For SPTHEO, the blocking of individual spawns provides a significant performance limitation, as it delays the start of all following tasks.
- environment evaluation suite
  A program which produces a table of performance statistics for the current virtual machine, e.g.
  - min/max/average time to spawn a process
  - min/max/average time to send messages of various sizes
  Such a tool would be very helpful for developers whenever tradeoffs depending on the communication performance have to be made as well as for everyday use for obtaining information on the currently available performance.

References

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proc. 27th ACM/IEEE Design Autom. Conf.* IEEE Comp. Soc. Press, 1990.

[CGH+93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, and K. L. McMillad L. A. Nessn an. Verification of the Futurebus+ Cache Coherence Protocol. In *Proc. 11th Intl. Symp. on Comp. Hardware Description Lang. and their Applications*, 1993.

[Ert93] W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen*, volume 25 of *DISKI*. Infix-Verlag, 1993.

[GLMS94] C. Goller, R. Letz, K. Mayr, and J. Schumann. SETHEO V3.2: Recent Developments (System Abstract) . In *Proceedings of CADE-12*, pages 778–782. Springer LNAI 814, 1994.

[Hub93] M. Huber. Parallele Simulation des Theorembeweiser SETHEO unter Verwendung des Static Partitioning Konzepts. Diplomarbeit, Institut für Informatik, Technische Universität München, 1993.

[LMG94] R. Letz, K. Mayr, and C. Goller. Controlled Integrations of the Cut Rule into Connection Tableau Calculi. Technical Report AR-94-02, Technische Universität München, 1994. submitted to JAR.

[LSBB92] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[Sch91] J. Schumann. Efficient Theorem Provers based on an Abstract Machine. Dissertation, Institut für Informatik, Technische Universität München, Germany, 1991.

[Sch95] J. Schumann. Using SETHEO for Verifying the Development of a Communication Pro tocol in Focus – A Case Study –. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Proc. of Workshop Analytic Tableaux and Related Methods, Koblenz*, volume 918 of *LNAI*, pages 338–352. Springer, 1995.

[SS94] C.B. Suttner and J. Schumann. Parallel Automated Theorem Proving. In *Parallel Processing for Artificial Intelligence 1*, Machine Intelligence and Pattern Recognition 14, pages 209–257. Elsevier, 1994.

[SS95] C.B. Suttner and G. Sutcliffe. The TPTP Problem Library (TPTP v1.2.0 - TR Date 19.5.95), 1995. Technical Report AR-95-03, Institut für Informatik, Technische Universität München, Munich, Germany; Technical Report 95/6, Department of Computer Science, James Cook University, Townsville, Australia.

[SSY94] G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In *Proceedings of the 12. International Conference on Automated Deduction (CADE)*, pages 252–266. Springer LNAI 814, 1994.

[Sut95] C.B. Suttner. *Parallelization of Search-based Systems by Static Partitioning with Slackness*, 1995. Dissertation, Institut für Informatik, Technische Universität München. Published as volume 101 of DISKI, Infix-Verlag, Germany.

[Sut96] C.B. Suttner. Static Partitioning with Slackness. In *Parallel Processing for Artificial Intelligence 3*. Elsevier, 1996. to appear.

[SWL+94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In *CADE 12*, pages 341–355, 1994.

[War83] D.H.D Warren. An Abstract PROLOG Instruction Set. Technical report, SRI, AI Center, Menlo Park, CA, USA, 1983.

The Requirements of a Database System for a Parallel Programming Environment‡

Giannis Bozas, Richard Lehn, Andreas Listl, Markus Pawlowski, Angelika Reiser
Institut fur Informatik, Technische Universität München
Arcisstr. 21, D-80290 München, Germany
e-mail: {bozas, lehn, listl, pawlowsk, reiser}@informatik.tu-muenchen.de

Abstract

MIDAS (MunIch Parallel DAtabase System) is a parallel relational database system [4]. It is well suited to serve as a platform for the exploration of various strategies in order to parallelize a relational database system. This paper depicts how the abstract architecture of MIDAS looks like. A detailed description shows the implementation of a MIDAS prototype on a network of UNIX workstations. The core of the MIDAS prototype is implemented using PVM. But pure PVM does not fulfil all demands in order to implement a parallel database system. We demonstrate how our implementation combines the PVM programming model with standard UNIX mechanisms. We use UNIX shared memory to enhance the efficiency of our implementation and we use access privileges to UNIX files to secure the database against unauthorized access. Furthermore, we describe the requirements of MIDAS for a parallel programming environment like PVM.

1 The MIDAS Architecture

MIDAS has a client/server architecture (see figure 1). The *MIDAS server* provides a high level interface (SQL) to retrieve and manipulate efficiently the contents of a relational database. The database itself is stored in the file system provided by the implementation platform. *MIDAS clients* are database applications. They are sequential programs issuing transactions to the MIDAS server. Each transaction consists of retrieval and update queries. A MIDAS client can run on any computer having access to the MIDAS server over anetwork.

The MIDAS server is composed of two layers: The *MIDAS access system* and the *MIDAS execution system* .

- The MIDAS access system supports parallelism between different MIDAS clients. This kind of parallelism is usually called *inter-transaction parallelism* . For that purpose, the access systeem provides a mechanism so that an arbitrary and varying number of clients can issue their queries to the MIDAS server in parallel.

  The second task of the access system is to compile, optimize and parallelize the queries issued by the clients. The queries are purely descriptive (SQL). They do not contain any parallel constructs. The access system generates query execution plans (QuEP) which can be performed by the execution system in parallel in order to answer the queries of the clients.

- The *MIDAS execution system* is responsible for the parallel execution of queries in accordance with their query execution plans. This kind of parallelism is usually called *intra-query parallelism* . It is achieved by the capability of the execution server to work on different parts of one QuEP simultaneously. Interim results can be exchanged between the workers involved in one QuEP.

‡This research is directed by Prof. R. Bayer Ph.D. and is part of the more extensive project "Tools and Methods for Using Parallel Computer Architectures" executed at the dept. of computer science of the Technische Universität München since 1990 (sponsored by DFG contract SFB 342).

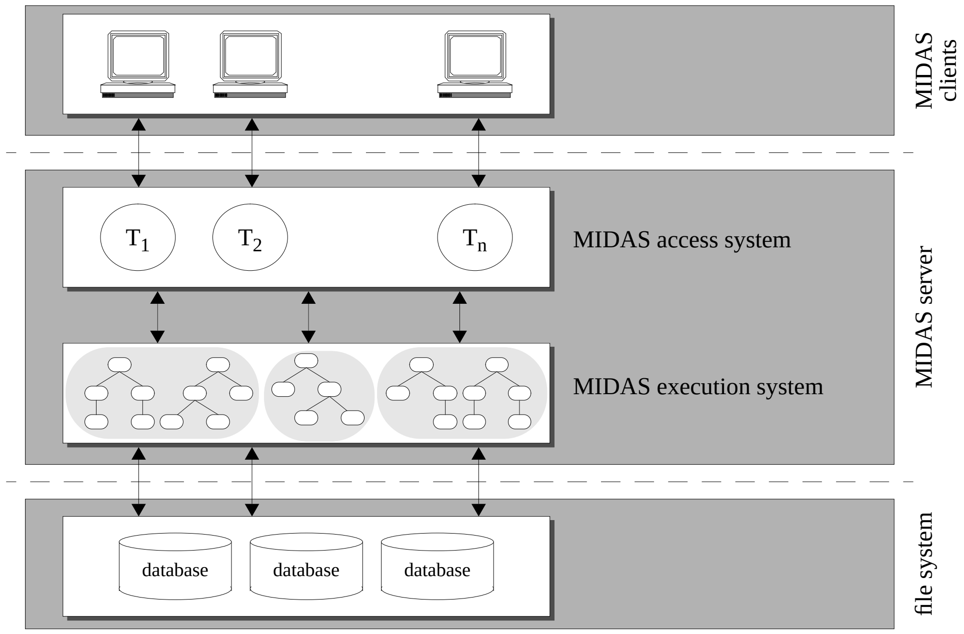


Figure 1: The MIDAS architecture

Furthermore, the MIDAS execution system has to provide an efficient access to the physical database stored on non volatile memory like magnetic disks.

2 The MIDAS Prototype

Flexibility and scalability are the two main goals which must be accomplished by the implementation of the MIDAS server. PVM seems to be well suited because it provides much flexibility. The possibility to start dynamically new PVM tasks helps to implement a scalable system. Whenever a new MIDAS client arrives, the MIDAS server can increase its degree of parallelism by starting new PVM tasks within the MIDAS server. These new PVM tasks are dedicated to the additional work caused by new MIDAS clients.

All components of the MIDAS server are implemented as a set of PVM tasks. All MIDAS clients are conventional UNIX processes. It is not possible to implement the clients as PVM tasks, because we must protect the database against unauthorized access and PVM is not capable of dealing with different user-IDs in one PVM program (see section 3).

The detailed description of the MIDAS prototype follows figure 2. A set of UNIX workstations builds the set of hosts composing the virtual machine on top of which the MIDAS server is running. The MIDAS server consists of a static and a *dynamic part* . Its static part is made of a fixed number of PVM tasks which does not change during the lifetime of the MIDAS server. Its dynamic part grows and shrinks during the lifetime of the MIDAS server.

2.1 MIDAS access system

The implementation of the MIDAS access system consists of one *connection server* and a varying number of application servers. The numbers within the following description refer to figure.

The *connection server* is a PVM task belonging to the static part of MIDAS. It builds the entry point to the MIDAS server for the MIDAS clients. The connection server can be accessed by remote procedure calls (RPC) and is globally known to all clients (1). Whenever a new client wants to connect to a database maintained by the MIDAS server, it applies for an application server. This is done by calling the appropriate remote procedure of the connection server (2). Due to that call the connection server starts a new application server for this client (3). This application server is a PVM task belonging to the dynamic part of MIDAS. The connection server takes the actual load of the virtual machine into account in order to choose the host where the application server will be started. By that means, the connection server is able to distribute the load among the hosts of the virtual machine evenly. Once that the application server is started, it is exclusively at the MIDAS client's disposal. The client and its application server are fellows and belong together until the client disappears. Furthermore, the application server provides a private RPC interface to its client. The client uses this interface to communicate with the application server by remote procedure calls (4). In this manner, we achieve a scalable system. The number of application servers grows with the number of clients actually connected to the MIDAS server. No single component of MIDAS becomes a bottleneck.

The *application server* receives queries from its client. The queries are transformed into QuEPs by compilation, optimization and parallelization. The application server initiates and controls the parallel execution of these QuEPs by scheduling. For that purpose, it starts and removes dynamically so-called interpreters (5). They belong to the MIDAS execution system (see below). Finally, the application server sends back the results produced by the execution of the QuEPs.

The application server disappears as soon as its client disconnects from the MIDAS server.

2.2 MIDAS execution system

The MIDAS execution system is designed as a set of *cache/lock servers* and a set of *interpreters* . The cache/lock servers belong to the static part of MIDAS. Exactly one cache/lock server is running on each host of the virtual machine. The interpreters belong to the dynamic part of MIDAS. Their actual number depends on the number of QuEPs actually performed by the MIDAS execution system and the chosen degree of parallelism for each QuEP.

The *cache/lock server* is a PVM task. It provides an efficient and transparent mechanism so that all interpreters can access all database pages. For that purpose, caching techniques are used [2]. Each cache/lock server installs a UNIX shared memory on its host. This UNIX shared memory makes up the local database cache (LDBC). It is divided into frames. Each frame can hold one database page. If an interpreter requires an access to a database page during the execution of a QuEP, it sends a request (6) to the cache/lock server running on its host. The cache/lock server puts the required page into a frame of the local database cache. The requesting interpreter is informed (6) about the address of the frame holding the required page. Efficiency is the reason why we use UNIX shared memory in order to implement the LDBC. Pages can be accessed by several interpreters sequentially or even simultaneously without copying them from one address space to another. Our MIDAS prototype uses UNIX shared memory, because PVM does not provide a mechanism to share parts of the address space of different PVM tasks running on the same host. But at least, PVM is flexible enough to combine the UNIX shared memory mechanism with the message passing library of PVM.

The usage of an own LDBC on each host of the virtual machine causes a cache coherency problem. It is solved by using a variation of the invalidation approach combined with dynamic page ownership to guarantee weak cache coherency [2, 3]. Two phase locking is used in order to synchronize different transactions. The communication between the cache/lock servers which is necessary to realize cache coherency and concurrency control is done by PVM messages (7). In this case, a thread mechanism would be very helpful to avoid the implementation of a complicated scheduling strategy in order to prevent deadlock situations.

The *interpreters* are PVM tasks as well. They are started and removed by the application servers (5). Their job is to execute one QuEP in parallel. Each interpreter works on one part of a QuEP which is sent to it by the application server (8). The lifetime of an interpreter coincides with the time necessary to execute its part of the QuEP assigned to it. The access to database

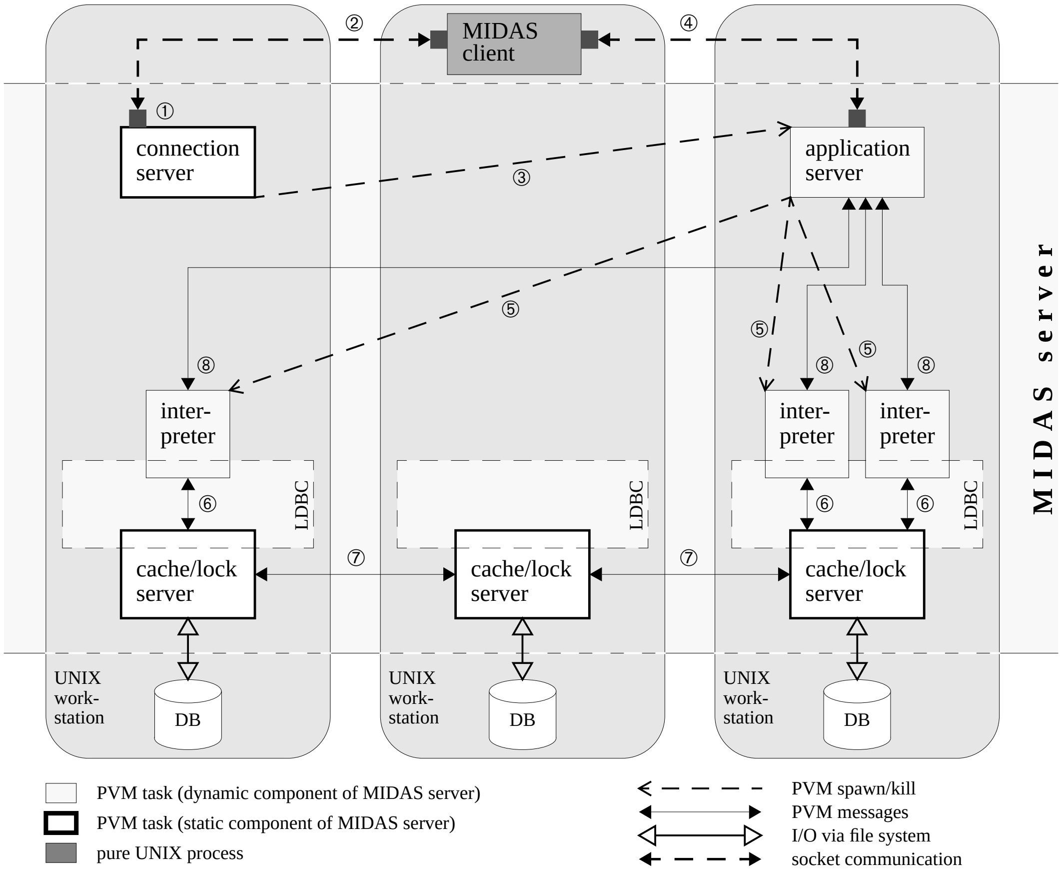


Figure 2: The MIDIAS Prototype

pages is performed locally by the mechanism described above. For that purpose, each interpreter must be attached to the UNIX shared memory building the LDBC.

3 Requirements

In order to achieve an efficient and robust implementation of the MIDAS prototype using a parallel programming environment like PVM we have the following mandatory requirements to PVM:

- **Protection and Authentication.** The physical database is stored in files of the UNIX file system. Protection bits grant access rights based on user-IDs. The PVM tasks making up the MIDAS server must be allowed to access the database files directly, whereas the MIDAS clients must be prevented from direct access because of security reasons. Therefore the clients and the MIDAS server must have different user-IDs. This is not possible within one PVM program [1].

- **Fault tolerance.** Typically database systems have to provide a high degree of fault tolerance. Especially in a parallel environment the failure of one computation node or disk should not break down the whole database system. Using PVM as parallel environment it is possible to design a highly fault tolerant database system. Because of its robustness PVM can easily be expanded to support fault tolerance [5].

- **Shared memory.** As mentioned in section 2.2 MIDAS uses shared memory to efficiently implement the local database cache on each node. Due to the lack of shared memory concepts in PVM MIDAS has to use UNIX shared memory, which makes it system dependent. Therefore our requirement to PVM is to provide a system independent mechanism to share parts of the address space of different PVM task running at least on the same host.

- **Threads.** The communication between the cache/lock servers is done by PVM messages. In this case, a thread mechanism would be very helpful to avoid the implementation of a complicated scheduling strategy in order to prevent deadlock situations. Furthermore threads would increase the performance of MIDAS: In the actual implementation the dynamically started PVM tasks (interpreter, application server) are realized as UNIX processes. But UNIX processes have high start-up costs. Using threads instead of processes would decrease these start-up costs dramatically, which leads to higher performance.

- **Dynamic behavior.** One main requirement of MIDAS to a parallel programming environment is the ability to dynamically start and stop tasks and to dynamically add and remove computing nodes to an from the set of nodes (virtual machine). Since PVM provides these abilities it is well suited to implement a parallel database system.

Additionally to these mandatory requirements a parallel programming environment should also be flexible and scalable, in order to implement a parallel database system. Furthermore it should provide a common file system and it should be available on many different platforms. Last but not least a parallel programming environment should be easy to use, since making parallel programs is difficult enough and the programer should not mess around with the difficulties of a not well implemented parallel programming environment.

4 Conclusion

PVM is well suited to implement a parallel database system. We have chosen PVM because it is convenient to use and flexible to scale the database system dynamically. Furthermore, PVM provides a high degree of fault tolerance which is important to database systems, also. One drawback of PVM is the absence of different access rights to the database stored in the UNIX file system within one PVM program. Another drawback is the lack of shared memory to implement an efficient database cache. Threads within PVM are desired to make the MIDAS implementation easier and more efficient.

References

[1] A. Geist et al.: PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994

[2] A. Listl, M. Pawlowski: Parallel Cache Management of a RDBMS. Technical Report SFB 342/18/92, Technische Universitat Munchen, August 1992

[3] A. Listl: Using Subpages for Coherency Control in Parallel Database Systems. In Proceedings of PARLE'94, Athens, Greece, July 1994

[4] G. Bozas et al.: Using PVM to implement a Parallel Database System. In Proc. of the 1st European PVM Users Group Meeting, October 1994.

[5] J. Leon, A.L. Fisher, P. Steenkiste: Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, 1993.

SEMPA:

Software Engineering Methods for Parallel Scientific Applications

Peter Luksch, Ursula Maier, S. Rathmayer, Matthias Weidmann
Lehrstuhl fr Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur
(LRR-TUM)
Institut fr Informatik
Technische Universitt Mnchen
D-80290 Mnchen
e-mail: {luksch, maier, maiers, weidmann}@informatik-tu.muenchen.de
WWW: http://wwwbode.informatik.tu-muenchen.de/
Tel.: (089)2105-8164; Fax: (089)2105-8232

Abstract

SEMPA is an interdisciplinary project that brings together researchers from computer science, mechanical engineering and numerical mathematics. Its central objective is to develop new software engineering (SWE) methods for (distributed memory) parallel scientific computing.

The parallelization of an industrial CFD software package (**TASCflow** from ASC GmbH) will serve as the main test case for defining and evaluating these methods. A major concern in parallelizing TASCflow is to achieve maximum portability, i.e. cover a large number of target systems ranging from (possibly heterogeneous) networks of workstations (NOWs) to massively parallel systems (MPPs).

In order to optimize utilization of NOWs, a resource management system is being designed, which runs parallel applications in batch mode. It keeps track of the resources that are not claimed for interactive use and assigns them to the parallel applications waiting in the queue. The individual tasks of the applications are dynamically reassigned as workstations are claimed for interactive use and other workstations become idle.

SEMPA is being funded by the BMBF (Federal Department of Education, Research and Technology). The project has started in April 1995, and is scheduled for three years, with a total staff of six researchers. LRR-TUM is in charge of project management.

1 Motivation

In many applications, fluid simulations are required because experiments are either impossible (such as in the case of climate modeling) or are too expensive. Today, the main factor that limits the use of simulation is run-time. Only parallel processing together with efficient numerical algorithms can achieve the performance that is necessary to enable more wide-spread use of simulation. Providing the necessary computational power will make simulations feasible in many areas where they would require unrealistic run-times today. In mechanical engineering, productivity can be considerably increased if flow simulations, which today have to be run as batch jobs overnight, could be run interactively from a CAD program.

Parallel processing has developed successfully in the research area over the last years. Now, as there are standardized message passing interfaces, such as PVM [GBD$^+$94], MPI [MPI94] etc., portable software can be developed for a wide range of hardware platforms – from (heterogeneous) networks of workstations (NOWs) to high-end massively parallel systems (MPPs). Since even small companies usually have a number of workstations connected by a local area network (LAN), developing parallel software on a commercial basis is becoming an attractive option.

However, experience has shown that software development for parallel systems still is much less productive than writing sequential programs. One reason for this is that there are no adequate

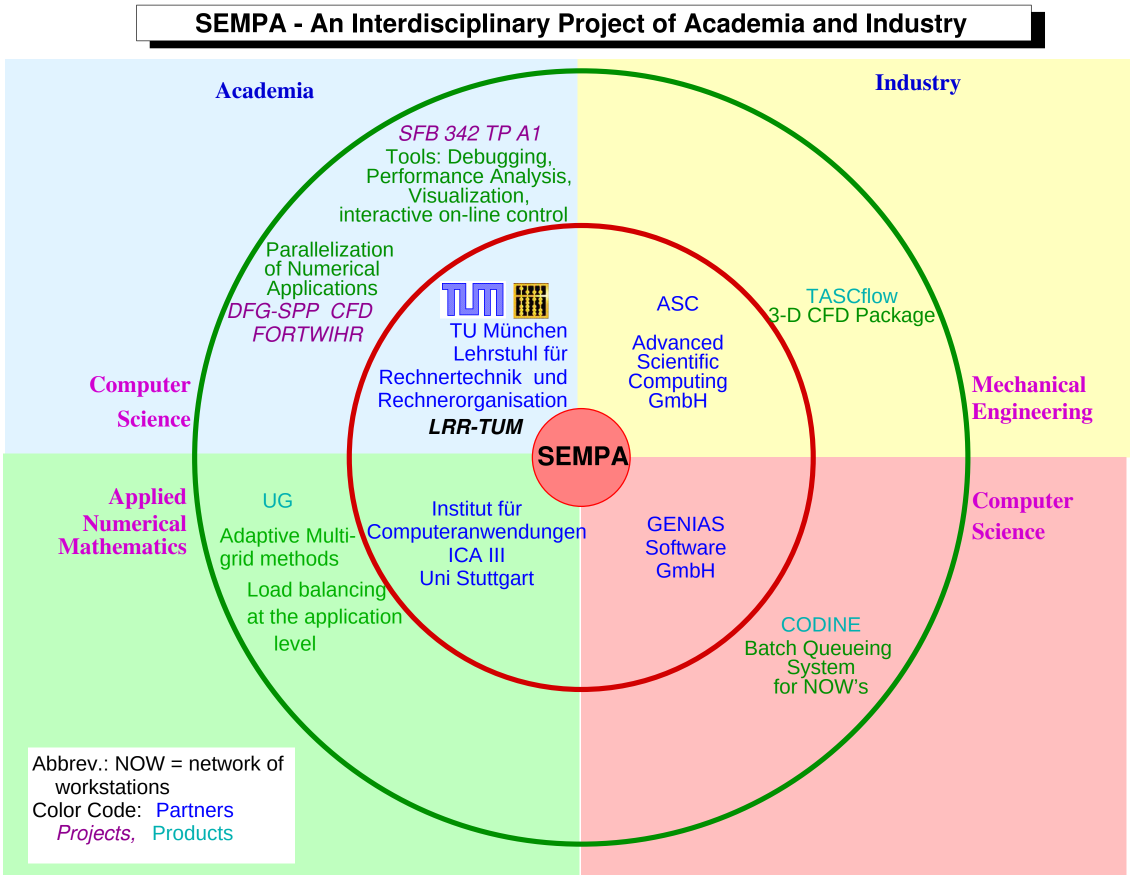


Figure 1: partners involved in the project

tools for designing and analyzing parallel software. In addition, there are no software engineering (SWE) methods that address the problems related to parallelism such as synchronization issues, deadlocks and non-determinism. Finally, there is only very little support for the software engineer who is faced with the problem of understanding and existing program in order to parallelize it for execution on a distributed memory multiprocessor.

2 Partners

SEMPA is a joint interdisciplinary project involving partners from industry and academia:

LRR-TUM (Lehrstuhl fr Rechnertechnik und Rechnerorganisation, Institut fr Informatik, Technische Universitt Mnchen). LRR-TUM is in charge of project management, and is doing research in the following areas of Computer Science:

- multiprocessor architectures
- tools for designing and analyzing parallel programs
- parallel and distributed applications
- distributed shared memory systems

For more detailed information, see LRR-TUM's WWW home page (URL http://wwwbode.informatik.tu-muenchen.de/).

Advanced Scientific Computing GmbH (ASC), Holzkirchen. ASC is developing and marketing the CFD simulation package **TASCflow** which solves the Navier-Stokes equations in 3d space. **TASCflow** is used in many companies and universities for simulating flows in a wide range of applications [TUG93, TUG94, TUG95].

GENIAS Software GmbH, Neutraubling near Regensburg. The company markets a number of software packages for NOWs and MPPs. They have developed the batch queuing system **CODINE** on NOWs, which will be the basis for the resource to be developed in SEMPA.

Institut fr Computeranwendungen (ICA III), Universitt Stuttgart. ICA's main areas of research in numerical mathematics are:

- robust multi-grid methods for a wide range of problems including computational fluid dynamics, flow in porous media and computational mechanics.
- the software tool-box UG, which simplifies the adaptive solution of partial differential equations on unstructured meshes in two and three dimensions.
- parallelization of the above-mentioned techniques which requires dynamic load migration and dynamic load balancing on the application level.

3 Objectives

In SEMPA, three major domains of research can be identified:

Software Engineering Methods. In parallel scientific computing, software engineers usually are faced with an existing program or with existing modules, typically written in FORTRAN77, which they are expected to parallelize for execution on a distributed memory multiprocessors (NOW or MPP). Therefore the focus of SWE is on the following topics:

Analysis of complex software systems. Before defining a concept for parallelizing an existing program, one must gain a basic understanding of its main data and control structures. Performance analysis is necessary to find out the computationally intensive modules.

General Approaches to Parallelizing Scientific Computations. A generally accepted approach to parallelizing scientific applications is the SPMD model. While the SPMD approach is very general, our objective is to define additional approaches which are specific to certain classes of scientific applications.

Documentation standards. Standards are needed for documenting the sequential software, the parallelization concept and its implementation.

Standards for program development. Guidelines have to be set up that define the the stages of the development process: functional specification, high level and detailed design, implementation, test and verification, documentation, and release. It is important that each stage of specification and design is carefully reviewed before proceeding to the next step.

Portability. Parallel software design on a commercial basis requires programs to be available for hardware platforms ranging from (low-end) NOWs to high performance MPPs. Portability does not only guarantee a large market to the software vendor, it also saves the users' investment into model design as they move to more powerful hardware platforms. Besides being independent from specific hardware platforms, the software should be designed to be as independent as possible from a specific message passing library, because it is not yet clear which interface will be *the* standard in the future and because for some MPPs considerable performance can be gained by porting the software to the vendor's native library.

Modularity and Re-usability. Maintenance of a large software package produced by multiple groups requires a modular structure, where modules interact via defined interfaces. Since FORTRAN 77 offers only minimal support for modular design, newer object oriented languages will be considered.

Concurrent Software Engineering. The design of parallel scientific software typically involves several groups of programmers from different disciplines and institutions working in parallel on different parts of the software system.

Parallelization of TASCflow. An enhanced version of the 3d CFD package marketed by ASC is being parallelized for execution on NOWs and MPPs. It is the major test case for developing and evaluating our SWE methods.

TASCflow solves the Navier-Stokes equation in 3d space on unstructured grids using a finite volume discretization and an algebraic multi-grid solver. The program is written in FORTRAN 77 and has about 113,000 lines of code.

Load Balancing and Resource Management. A resource manager is being developed, which basically is a batch queuing system for parallel applications running on NOWs. The individual processes of the application are dynamically assigned to available processors (i.e. workstations). The resource manager will support load balancing by providing appropriate resource usage information and a mechanism to migrate processes from one workstation to another.

Figure 2 illustrates the interactions between these domains. In this paper, we will focus on the area of SWE methods.

4 Software Engineering in SEMPA

Being an interdisciplinary project of partners from academia and industry, the emphasis of SEMPA is on putting into practice the methods that are defined in the project. The experience gained from applying them to the parallelization of **TASCflow** will help to improve our methods. **TASCflow** is particularly well suited as a test case, because it is a complex software system which implements state-of-the-art methods in CFD simulation and because the sequential program has been designed based on SWE methods used in industry.

SEMPA focuses on the task of designing parallel scientific software, either from existing (sequential) programs or by integrating existing software modules into a new parallel program. As mentioned in section 3, portability and modularity are central design objectives. Given this background, we have to consider FORTRAN 77, the language in which most scientific software is written, although its support for modularity is minimal. An important objective is to demonstrate practical ways towards using object oriented languages in software systems for which complete re-implementation is unfeasible for reasons of manpower.

The parallelization paradigm adopted in SEMPA is message passing, because message passing currently is the generally accepted standard for programming MPPs and NOWs, and because it is the only paradigm for which efficient implementations are available for a wide range of hardware platforms.

The following aspects of SEMPA are of particular relevance to software engineering for *parallel and distributed* systems:

Analysis of complex software systems. In SEMPA, two methods for acquiring the knowledge necessary to set up a parallelization concept are considered:

"human" analysis: The authors of the sequential program (mostly engineers) provide the information needed by computer scientists (who do the parallelization). This is usually an interdisciplinary effort requiring several iterations to overcome mutual misconceptions.

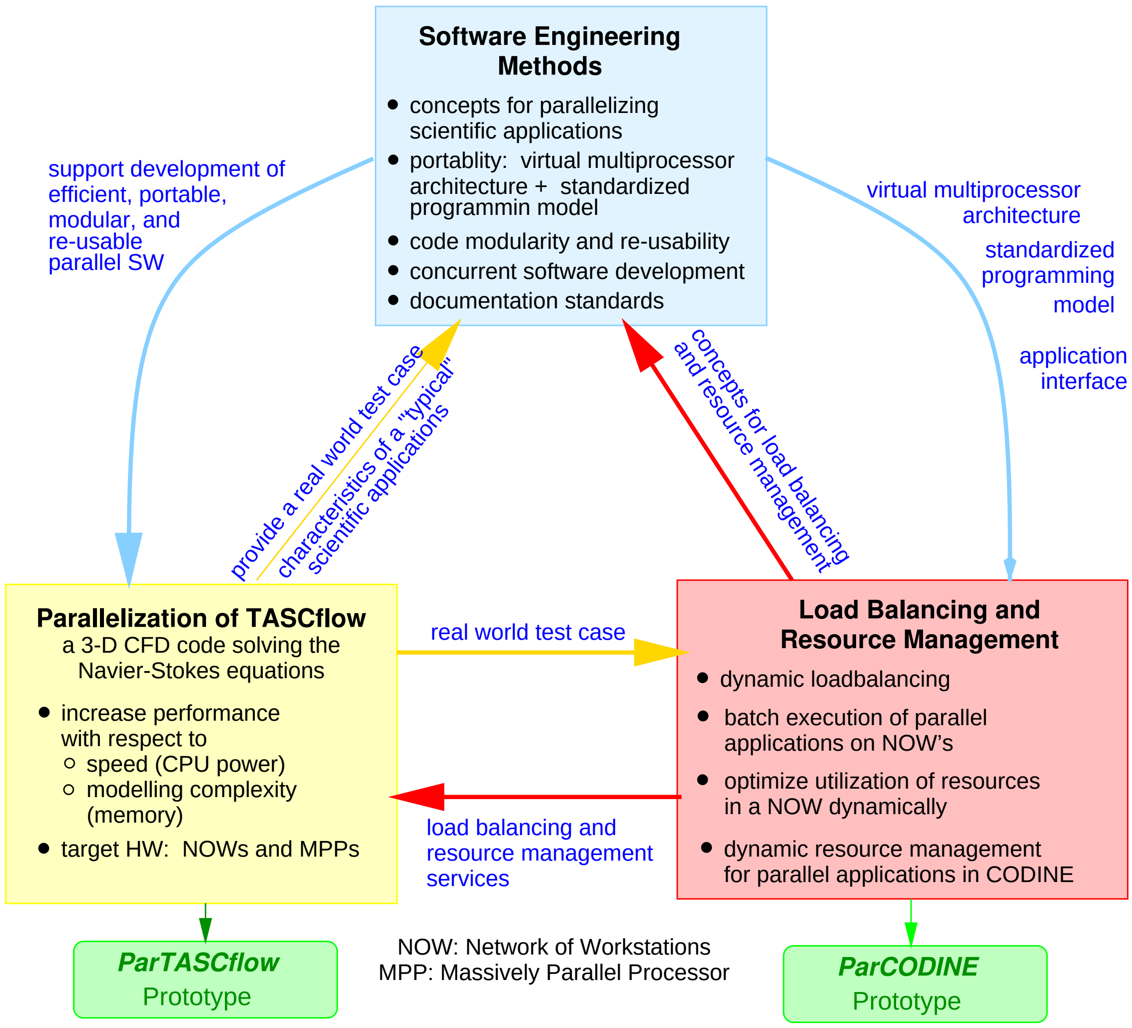


Figure 2: project objectives

tool supported analysis: Analysis of data and control dependencies with the help of interactive parallelization tools.

Design and Analysis Tools. A survey conducted by C.M. Pancake [PC94, CPW94] has shown that tools for programming multiprocessors often prove to be inadequate in practice, because on the one hand tool developers lack the necessary understanding of the requirements that developers of complex "real world" applications have, while on the other hand most application programmers refuse to spend much effort in trying to use tools.

Since at LRR-TUM there is a large research group designing tools for parallel systems (The Tool-Set [LW95]), SEMPA (and other projects in the research group *"parallel and distributed applications"*) can help to improve the exchange of information and experience between tool designers and tools users.

Resource management in NOWs: Efficient resource management is of particular interest for users running large simulations on NOWs, since it provides a low-cost entry to parallel processing especially to small to medium-size companies or research institutions.

The result of SEMPA will be

- a collection of SWE methods that have been approved in practice,
- a prototypical implementation of the parallel version of **TASCflow**,
- a prototype of a resource and load manager for batch execution of parallel applications.

Our emphasis in SWE is on practical applicability of the proposed methods. Starting with SWE guidelines that have been approved in practice, we develop and integrate new methods, which are motivated by our experience or by concepts published in literature. Our approach to SWE thus is evolutionary in that, starting with an approved set of methods, each step towards an improved methodology is evaluated immediately upon implementation.

We are interested in cooperation with other SWE projects, especially if their approach to SWE starts from the theory side.

Upon completion of the research project, our industrial partners intend to develop further the prototypes of the parallel CFD package and the resource manager towards products that can be marketed commercially.

5 Progress Report

The project has started in April, 1995. Up-to-date information about progress as well as project reports and publications related to SEMPA are available via WWW (URL http://wwwbode.informatik.tu-muenchen.de/parallelrechner/applications/sempa/). In the subsequent section, we summarize the results achieved so far.

5.1 Analyzing the sequential program

The first step in parallelizing **TASCflow** has been to acquire the necessary understanding of the algorithms it uses and their implementation. ASC and LRR-TUM have been organizing a series of meetings, covering the following topics (in that sequence):

1. basics of CFD, i.e. the governing equations and their physical interpretation, discretization methods, and numerical methods for solving the system of linear equations that results from discretization.
2. a global overview of the code structure and the main data structures.
3. a more detailed review of **TASCflow**'s main modules, stepping through each module subroutine by subroutine.

Each meeting started with a presentation by ASC, which was followed by a discussion. At LRR-TUM, we documented our view of what we had learned in a meeting in form of an internal report, which then was reviewed by ASC. This procedure has proved to be an efficient way for know-how transfer between groups from different disciplines, since it has helped to identify and fix sources of misconception very early and quickly improved our understanding of each other's terminology and point of view.

As a final step, a framework has to be set up that defines a standard for documenting the design of the sequential program from the computer science point of view.

5.2 A Concept for Parallelizing TASCflow

Based on the insight gained from analyzing the program structure, a parallelization concept has been defined and documented [Luk95]. SEMPA follows a two-level concept of parallelism.

On the top level the SPMD model is used. The sequential algorithm[1] is replicated in multiple processes each of which operates on a partition of the problem description. An additional master

[1] augmented by additional code for communication and synchronization

process is used for program set up and for doing I/O. Using parallel I/O systems, which are available for a number of platforms, is being considered, too.

Partitioning is done node-based, i.e. the nodes of the (unstructured) grid are divided into disjoint sets. We use a public domain graph partitioning package (MeTiS [MET95]) for assigning nodes to partitions.

Below the SPMD level of parallelism, parallelization is considered at the level of processing nodes. Each replicated worker of the SPMD model can be further parallelized into a number of concurrent threads (light-weight processes having access to shared memory). This second level of parallelism can make use of multiple CPUs per processing node as they are available in new MPPs or workstations.

5.3 Interactive and automatic Parallelization Tools

The parallelization of an existing program, especially if it is complex and has been developed by many engineers over a long time, is a quite difficult and error-prone task.

Research projects as well as commercial efforts during the last years have been dealing with this problem. Most available tools are source-code analyzers for FORTRAN 77 programs which parallelize according to the SPMD model. One of those tools has already been subject of investigation within SEMPA. It is the quite sophisticated interactive an automatic parallelization tool FORGE [Res95]. There the most significant loops are identified by either using profiling information, or checking the code for the deepest loop nestings. Once the loops have been chosen, the arrays referenced inside of them are partitioned and distributed according to the partitions. The parallel processes then run the same program but only on a subset of the partitioned data structures following the so-called *owner computes* rule.

The advantage of these tools is that the user can get a better understanding of the program that he is about to parallelize. He also is taken off the burden to explicitly program message passing code. On the other hand he anyhow has to have a good understanding of how message passing really works because the tools are not yet at a point where they can produce efficient code. Neither can they really work on very complex packages as for example TASCflow.

5.4 New Languages

Moving from FORTRAN 77 to newer programming languages meets the requirements of modern computer architectures, programming paradigms, and software engineering aspects. Fortran 90 for example offers not only more complex data structures and data encapsulation but also provides language constructs (array operations) for concurrent execution. The latest of all FORTRAN evolutions, **H**igh **P**erformance **F**ortran, additionally has constructs for explicit data distribution as well as constructs for expressing concurrency.

FORTRAN 77 compilers produce fast object code and numerous numerical programming libraries are available due to its long time of existence. Object oriented design is still uncommon in scientific computing because the compilers (e.g. for C++) do not yet generate optimized code that is comparable to the one generated by FORTRAN 77 compilers. However, the fundamental ideas of object oriented programming – objects, class-hierarchies and polymorphism – are of great advantage to modern software engineering and can help to overcome the gap between the code development and its concept.

In SEMPA, we have decided that rewriting TASCflow as a whole in an object-oriented language is infeasible due to manpower restrictions. Instead, we have selected a module of reasonable size for implementation in Fortran 90, C++ (and possibly other languages) to demonstrate the integratability of object-oriented techniques to a scientific application, and to evaluate the appropriateness of these languages for our purposes.

References

[CPW94] CURTIS COOK, CHERRI M. PANCAKE, AND REBECCA WALPOLE. Supercomputing'93: Parallel User Survey: Response Summary. In "Proceedings of Scalable High Performance Conference" (May 1994).

[GBD+94] AL GEIST, ADAM BEGUELIN, JACK DONGARRA, WEICHENG JIANG, ROBERT MANCHEK, AND VAIDY SUNDERAM. "PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing". MIT Press (1994). www: http://www.netlib.org/pvm3/book/pvm-book.html.

[Luk95] PETER LUKSCH. A Concept for Parallelizing TASCflow. SEMPA-Report, Technische Universität München, Institut für Informatik (September 1995). www: //http:/wwwbode.informatik.tu-muenchen.de/parallelrechner/applications/sempa/Reports/ws-aug-95.ps. draft version.

[LW95] THOMAS LUDWIG AND ROLAND WISMÜLLER. The TOOL-SET. SFB 342–Bericht, Technische Universität München, Institut für Informatik (1995). to appear.

[MET95] "METIS: Unstructured Graph Partitioning and Sparae Matrix Ordering System". George Karypis and Vipin Kumar, University of Minnesota (1995).

[MPI94] MPI: A Message Passing Interface Standard. Technical report University of Tennessee, Knoxville, Message Passing Interface Forum (May 1994).

[PC94] C.M. PANCAKE AND C. COOK. What Users Need in Parallel Tools Support: Survey, Results and Analysis. In "Proceedings of Scalable High Performance Conference" (May 1994).

[Res95] APPLIED PARALLEL RESEARCH. "The FORGE Product Set". Applied Parallel Research Inc., 550 Main Street, Placerville, CA 95667 (February 1995).

[TUG93] 1st TASCflow User Conference – Presentations. Tech. Report ASCG/TR-93-03, Advanced Scientific Computing GmbH, Holzkirchen (1993).

[TUG94] 2nd TASCflow User Conference – Presentations. Tech. Report ASCG/TR-94-04, Advanced Scientific Computing GmbH, Taufkirchen (July 1994).

[TUG95] 3rd TASCflow User Conference – Presentations. Tech. Report ASCG/TR-95-04, Advanced Scientific Computing GmbH, Aying (May 1995).

Parallel Communication on Workstation Networks with Complex Topologies †

Alexander Pfaffinger

Institut für Informatik
Technische Universität München
Arcisstraße 21, D-80290 München 2

1 Introduction

In the field of parallel high performance computing, there is a trend towards workstation clusters, which are usually interconnected via Ethernet buses. Networked workstations are already available at many sites and typically their computing capacity is not fully exploited. Therefore, they offer a cost-effective alternative to dedicated parallel computers.

Recently, a number of machine independent parallel platforms like MPI, PVM, and LINDA has been developed. They support the distributed programming of a workstation cluster. These systems are now widely used in parallel numerical computing.

With increasing demand for communication between the parallel tasks, however, the bus-like nature of the Ethernet soon becomes a bottleneck. Hence, only few computing nodes can be used efficiently. The system is not scalable with respect to the number of processors.

By introducing additional buses and a more complex interconnection topology of the workstations, the communication bandwidth can be significantly improved. The required hardware investment are just some additional LAN adapter cards and Ethernet cables. Depending on the network topology, pairs of workstations may now communicate in parallel.

In this paper we will describe two network topologies which both need only two I/O-ports per workstation. In section 2 we will sketch out a tree-like structure which is optimally adapted to divide-and-conquer algorithms. In section 3 we will present a more general hypercube-like topology.

2 A Hierarchical Topology

A common model for parallel computing is the divide-and-conquer paradigm: a task is split into two (or more) subtasks that can be executed in parallel. Each of these subtasks can be divided in subsubtasks and so on.

This leads to a tree-like dependence graph. In the case of exactly two subtasks per node we obtain a binary tree. This graph also reflects the communication scheme of the parallel tasks. A task needs to talk only to its parent and its children.

When we consider the distribution of the jobs onto the computing nodes, we get a slightly different graph. The parallel algorithm starts at node one. At the point of division two new tasks are created, which should be placed at different nodes. Since the parent is idle when the two subjobs are calculating, one subtask can remain on the initial node. This leads to a "squashed" binary tree. Figure 1 shows a divide-and-conquer graph with 16 computing nodes. Each left child of a node is identified with its parent.

We will discuss the communication pattern in more detail. Most importantly, we must distinguish between the cheap communication within a node and the expensive external communication

†This work was supported by the Siemens AG.

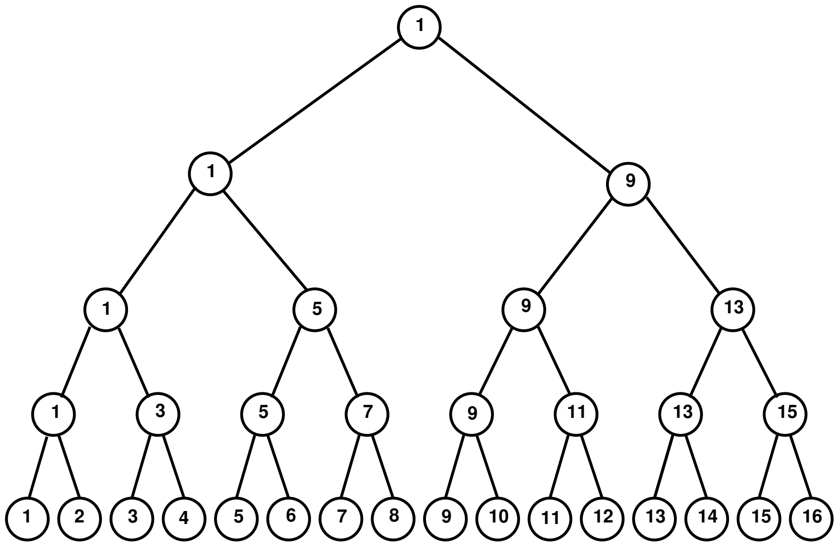


Figure 1: Binary tree with 16 computing nodes: dependence and communication graph of a parallel divide-and-conquer application. Each interior node is identified with its leftmost descendent.

between different nodes. The first one arises between parent and left child in Figure 1. Only the communication of each parent with its right child is external.

If we assume that all tasks at one layer of the tree need quite the same computation time, there are no simultaneous communications at different (edge-) levels of the tree. Therefore, it is desirable that all communications on one level are done in parallel, i.e. on different buses. The problem is to find a minimal bus topology in which at each level all nodes can communicate with their right child via different buses.

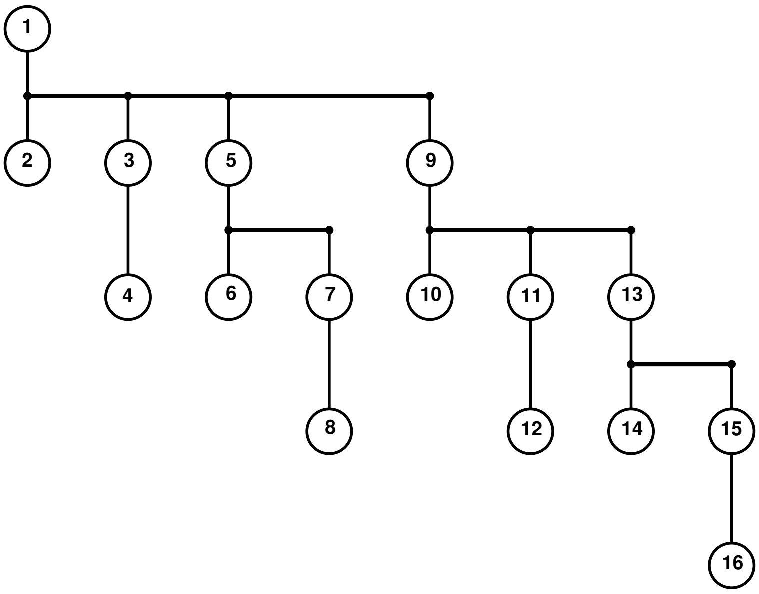


Figure 2: Tree-like Ethernet topology: all communication between parent and (right) child at one level of Figure 1 can be done in parallel via different buses.

Of course, this could be achieved by a complete interconnection, where each node is connected to any other node via a different bus (or link). This would mean that for n processing elements

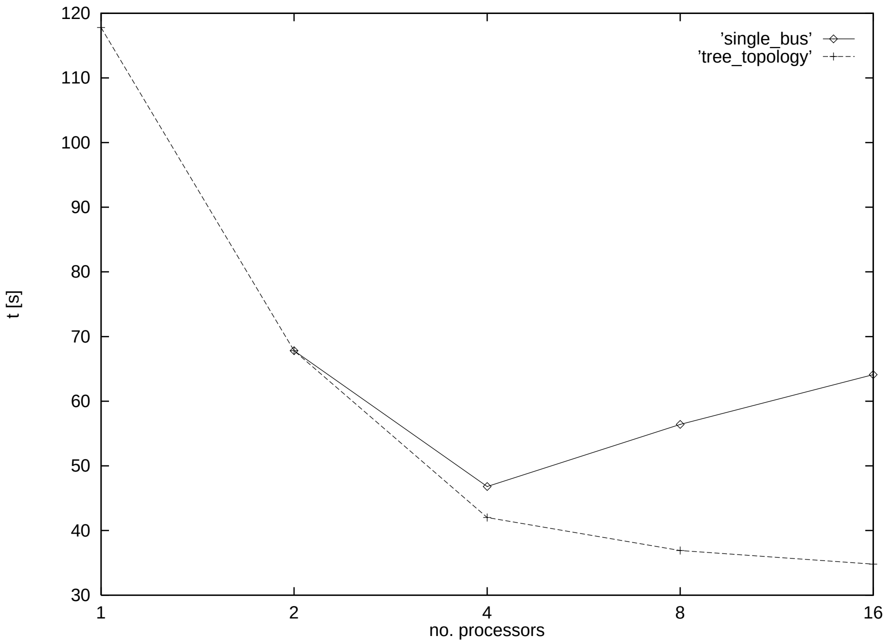


Figure 3: ARESO runtimes on the single bus and the hierarchical tree-like topology.

each node needs to have $n-1$ I/O-ports. But real hardware will always show a constant maximum. In this paper we will examine topologies where only two I/O ports per workstation are needed.

Figure 2 shows an appropriate tree-like solution with 8 buses that is obviously minimal in respect to the number of buses (we have 8 parallel communication at the deepest level). Its hierarchical structure reflects the recursive definition of the corresponding squashed binary tree.

The next higher network, which includes 32 nodes and 16 buses, would be constructed as follows: take a copy of the 16-node-network, add 16 to all of its node labels, and put the least labeled node (17 here) with an additional link into the bus that connects 1 and 2. This topology corresponds perfectly to the complete squashed binary tree with 32 leaf nodes.

In general, for a complete binary tree with 2^d leaves the corresponding bus network would consist of 2^d computing nodes and 2^{d-1} buses. Each node is connected to at most 2 buses and each bus is assigned to at most $d+1$ nodes.

For each level in the binary tree all data transfer at that level from parent to child (or vice versa) can be done in parallel. This is a prerequisite of scalability.

Numerical Applications

The architecture shown in Figure 2 has been implemented with 16 HP-720 workstations. Each odd-numbered node has been provided with an additional LAN adapter card. Node 1 is used as the gateway to outside networks.

On this installation several applications with a divide and conquer strategy have been parallelized. Besides the complex chip placement algorithm GORDIAN [5] this is the parallel Finite Element application ARESO [4, 6] which we will describe somewhat more in detail below.

ARESO is a solver for partial differential equations that is based on domain decomposition and recursive substructuring. In the case of a square domain ARESO starts on top level with a certain problem size N, typically a power of 2, that describes the number of grid points on the borderlines.

When the problem is split up in two parts, the size decreases by the factor $1/\sqrt{2}$. (More accurately, N is halved every second step.) At level l we, therefore, get $N_l = N/\sqrt{2}^l$. The computation time per node on level l is of order $O(N_l^3)$ while the communication amount per node is $O(N_l^2)$. This means that the message size per node decreases with increasing level l, but the number of communicating nodes increases. With a single Ethernet this leads to high collision rates.

Figure 3 compares the runtime of ARESO for a fixed problem size on the single bus and on the tree topology. At 4 nodes the tree structure has only a slight advantage over the single bus. But involving 8 and 16 computers the single bus is overloaded so that no speed up is obtained. The runtime is even longer than for 4 nodes. For the tree topology, in contrast, the algorithm still shows a speed up.

It is interesting that the advantage of multiple buses is not only restricted to divide-and-conquer algorithms. In [8] the runtime behavior of a parallel unstructured matrix application on the different topologies was compared. It concerns the distributed solution of a big sparse block matrix system. The matrix is split into different block rows that are distributed among the processors. Eliminated rows must be transferred to those processors who need the specific row for elimination of one of its own rows. A detailed discussion of the numerical problem and the algorithm can be found in [7, 6, 8].

Figure 4 shows the runtimes on the two topologies for different block sizes. Again, the multiple bus system is significantly faster than the single bus.

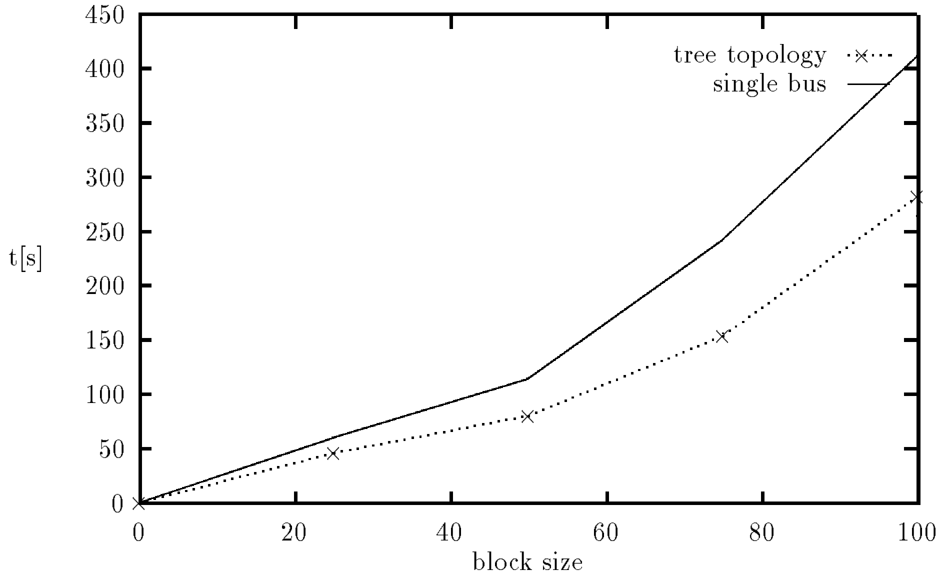


Figure 4: Runtime comparison of a distributed block matrix solver on the single bus and on the hierarchical tree network.

Nevertheless, the introduced tree topology is not very convenient for unstructured dependence graphs. We will sketch out a better solution in the next section.

3 The Dual Hypercube

The tree-like topology in the previous section is tailored to applications where the process graph is a complete binary tree that arises from the divide-and-conquer paradigm. However, for more general applications several inherent disadvantages and difficulties may arise.

If the dependence graph is less structured, e.g. even for divide-and-conquer with adaptivity, it is not clear how the processes should be mapped to the tree. Simple and convenient embedding schemes may result in a large amount of long-distance communication and an overload of the top level bus.

Another problem is fault tolerance. If workstation no. 9 in Figure 2 crashes, the half of the 16 computers is unreachable. A more symmetric topology with several communication paths between different nodes seems to be more comfortable.

A general topology would be the hypercube. But two main problems arise. First, the direct links between two nodes do not fit the bus-like nature of Ethernet. Secondly, we would need a logarithmic number of connections per workstation. In practice only a fixed number of connections per workstation is supported.

Both difficulties vanish if we exchange the roles of nodes and edges in the hypercube graph. Then, each edge represents a computing node with exactly two connections to buses, which are in their turn represented by the graph vertices. Because we changed the role of nodes and edges, we call this family of topologies *dual hypercube*. Figure 5 shows an example of dimension $d = 3$.

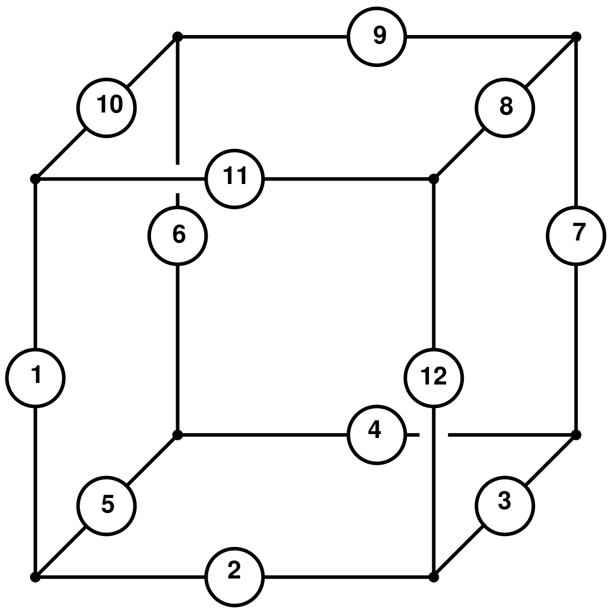


Figure 5: Dual hypercube of dimension 3.

For dimension d we get $d2^{d-1}$ nodes and 2^d buses. Thus, we have $d/2$ as many nodes as buses, enough to provide a logarithmic diameter d and high throughput of the network. Since d is only the logarithm of the graph size, the amount of communication in each bus is expected to increase slowly. Each bus is assigned to exactly d workstations and each node needs to have only two I/O-ports. Some other basic properties can be found in [1, 2] where similar types of graphs were introduced.

Due to its proximity to the hypercube and cube-connected-cycles, the dual hypercube seems to be well suited for a broad class of algorithms.

3.1 Hardware Realization and Numerical Tests

By introducing four additional buses in the former tree-like network we could realize the 3-dimensional dual hypercube with 12 of the 16 workstations. Using different routing mechanisms we could directly compare the performance of the two distinct topologies.

While the routing on the tree could be done statically by standard UNIX software, it was a nontrivial issue to realize a reasonable routing scheme for the cube. Dynamic routers like *gated* do not allow to change the path from one node to another in short intervals (e.g. between two messages or packets). Parallel platforms like PVM don't support multiple paths between two workstations at all.

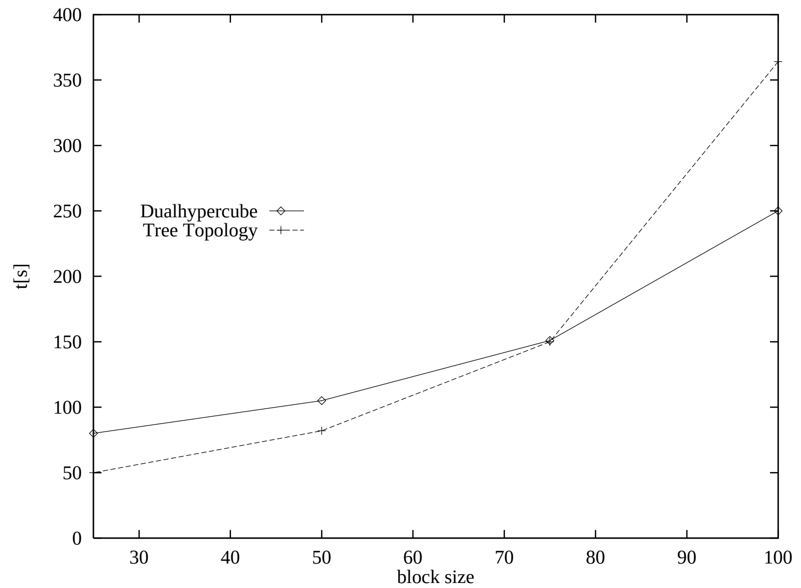


Figure 6: Runtimes of the sparse block matrix solver on the dual hypercube and the hierarchical topology.

Therefore, a new routing daemon MRouter [3] (based on the TCP protocol) was developed, which runs a random path strategy: before each message is sent from one node to another the route is determined uniformly random among all optimal (i.e. shortest) paths.

We adapted the sparse block matrix solver mentioned in section 2 to MRouter and compared it to the PVM version on the tree-like network using the 12 HP-720 workstations that belong to the three-dimensional dual hypercube. The runtime behavior on the different topologies is shown in Figure 6.

At low block sizes (up to 75) the version on the tree-like topology is faster due to the minor overhead of the UDP protocol used by PVM. When the size of the matrix blocks grows, however, the hypercubic network is more scalable.

4 Conclusions

We presented two classes of bus topologies for workstation clusters. The first one, a hierarchical tree-like network, is shown to be best suited for divide-and-conquer parallelization. This theoretical consideration is confirmed by real applications running on 16 HP-720 computers connected via Ethernet cables.

The second class of topologies, the dual hypercube, is better suited for more general algorithms requiring more flexibility. This includes adaptive divide-and-conquer applications and totally unstructured (random) communication dependencies.

References

[1] D.P. Agrawal and V.K. Janakiram. Evaluating the Performance of Multicomputer Configurations. *Computers*, May 1986.

[2] L.N. Bhuyan and D.P. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Transactions on Computers*, April 1984.

[3] M. Eckart. Parallelisierung auf hypercubeartigen Workstationnetzen. Master's thesis, Technische Universität München, 1994.

[4] R. Hüttl and M. Schneider. Parallel Adaptive Numerical Simulation. SFB-Bericht 342/01/94 A, Technische Universität München, 1994.

[5] H. Regler and U. Rüde. Layout optimization with algebraic multigrid methods (AMG). In *Proceedings of the Sixth Copper Mountain Conference on Multigrid Methods, Copper Mountain, April 4-9, 1993*, Conference Publication. NASA, 1993.

[6] M. Schneider. *Verteilte adaptive numerische Simulation auf der Basis der Finite-Elemente-Methode*. PhD thesis, Technische Universität München, 1994.

[7] M. Schneider, U. Wever, and Q. Zheng. Solving large and sparse linear equations in analog circuit simulation on a cluster of workstations. *The Computer Journal*, 36(8):685–689, 1993.

[8] B. Weininger. Lösen großer, dünnbesetzter linearer Gleichungssysteme auf einem Netz von Workstations. Master's thesis, Technische Universität München, 1994.

SFB 342: Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

bisher erschienen :

Reihe A

342/1/90 A Robert Gold, Walter Vogler: Quality Criteria for Partial Order Semantics of Place/Transition-Nets, Januar 1990

342/2/90 A Reinhard Fößmeier: Die Rolle der Lastverteilung bei der numerischen Parallelprogrammierung, Februar 1990

342/3/90 A Klaus-Jörn Lange, Peter Rossmanith: Two Results on Unambiguous Circuits, Februar 1990

342/4/90 A Michael Griebel: Zur Lösung von Finite-Differenzen- und Finite-Element-Gleichungen mittels der Hierarchischen Transformations-Mehrgitter-Methode

342/5/90 A Reinhold Letz, Johann Schumann, Stephan Bayerl, Wolfgang Bibel: SETHEO: A High-Performance Theorem Prover

342/6/90 A Johann Schumann, Reinhold Letz: PARTHEO: A High Performance Parallel Theorem Prover

342/7/90 A Johann Schumann, Norbert Trapp, Martin van der Koelen: SETHEO/PARTHEO Users Manual

342/8/90 A Christian Suttner, Wolfgang Ertel: Using Connectionist Networks for Guiding the Search of a Theorem Prover

342/9/90 A Hans-Jörg Beier, Thomas Bemmerl, Arndt Bode, Hubert Ertl, Olav Hansen, Josef Haunerdinger, Paul Hofstetter, Jaroslav Kremenek, Robert Lindhof, Thomas Ludwig, Peter Luksch, Thomas Treml: TOPSYS, Tools for Parallel Systems (Artikelsammlung)

342/10/90 A Walter Vogler: Bisimulation and Action Refinement

342/11/90 A Jörg Desel, Javier Esparza: Reachability in Reversible Free- Choice Systems

342/12/90 A Rob van Glabbeek, Ursula Goltz: Equivalences and Refinement

342/13/90 A Rob van Glabbeek: The Linear Time - Branching Time Spectrum

342/14/90 A Johannes Bauer, Thomas Bemmerl, Thomas Treml: Leistungsanalyse von verteilten Beobachtungs- und Bewertungswerkzeugen

342/15/90 A Peter Rossmanith: The Owner Concept for PRAMs

Reihe A

342/16/90 A	G. Böckle, S. Trosch: A Simulator for VLIW-Architectures
342/17/90 A	P. Slavkovsky, U. Rüde: Schnellere Berechnung klassischer Matrix-Multiplikationen
342/18/90 A	Christoph Zenger: SPARSE GRIDS
342/19/90 A	Michael Griebel, Michael Schneider, Christoph Zenger: A combination technique for the solution of sparse grid problems
342/20/90 A	Michael Griebel: A Parallelizable and Vectorizable Multi- Level-Algorithm on Sparse Grids
342/21/90 A	V. Diekert, E. Ochmanski, K. Reinhardt: On confluent semi-commutations-decidability and complexity results
342/22/90 A	Manfred Broy, Claus Dendorfer: Functional Modelling of Operating System Structures by Timed Higher Order Stream Processing Functions
342/23/90 A	Rob van Glabbeek, Ursula Goltz: A Deadlock-sensitive Congruence for Action Refinement
342/24/90 A	Manfred Broy: On the Design and Verification of a Simple Distributed Spanning Tree Algorithm
342/25/90 A	Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Peter Luksch, Roland Wismüller: TOPSYS - Tools for Parallel Systems (User's Overview and User's Manuals)
342/26/90 A	Thomas Bemmerl, Arndt Bode, Thomas Ludwig, Stefan Tritscher: MMK - Multiprocessor Multitasking Kernel (User's Guide and User's Reference Manual)
342/27/90 A	Wolfgang Ertel: Random Competition: A Simple, but Efficient Method for Parallelizing Inference Systems
342/28/90 A	Rob van Glabbeek, Frits Vaandrager: Modular Specification of Process Algebras
342/29/90 A	Rob van Glabbeek, Peter Weijland: Branching Time and Abstraction in Bisimulation Semantics
342/30/90 A	Michael Griebel: Parallel Multigrid Methods on Sparse Grids
342/31/90 A	Rolf Niedermeier, Peter Rossmanith: Unambiguous Simulations of Auxiliary Pushdown Automata and Circuits
342/32/90 A	Inga Niepel, Peter Rossmanith: Uniform Circuits and Exclusive Read PRAMs
342/33/90 A	Dr. Hermann Hellwagner: A Survey of Virtually Shared Memory Schemes
342/1/91 A	Walter Vogler: Is Partial Order Semantics Necessary for Action Refinement?
342/2/91 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Rainer Weber: Characterizing the Behaviour of Reactive Systems by Trace Sets
342/3/91 A	Ulrich Furbach, Christian Suttner, Bertram Fronhöfer: Massively Parallel Inference Systems

Reihe A

342/4/91 A	Rudolf Bayer: Non-deterministic Computing, Transactions and Recursive Atomicity
342/5/91 A	Robert Gold: Dataflow semantics for Petri nets
342/6/91 A	A. Heise; C. Dimitrovici: Transformation und Komposition von P/T-Netzen unter Erhaltung wesentlicher Eigenschaften
342/7/91 A	Walter Vogler: Asynchronous Communication of Petri Nets and the Refinement of Transitions
342/8/91 A	Walter Vogler: Generalized OM-Bisimulation
342/9/91 A	Christoph Zenger, Klaus Hallatschek: Fouriertransformation auf dünnen Gittern mit hierarchischen Basen
342/10/91 A	Erwin Loibl, Hans Obermaier, Markus Pawlowski: Towards Parallelism in a Relational Database System
342/11/91 A	Michael Werner: Implementierung von Algorithmen zur Kompaktifizierung von Programmen für VLIW-Architekturen
342/12/91 A	Reiner Müller: Implementierung von Algorithmen zur Optimierung von Schleifen mit Hilfe von Software-Pipelining Techniken
342/13/91 A	Sally Baker, Hans-Jörg Beier, Thomas Bemmerl, Arndt Bode, Hubert Ertl, Udo Graf, Olav Hansen, Josef Haunerdinger, Paul Hofstetter, Rainer Knödlseder, Jaroslav Kremenek, Siegfried Langenbuch, Robert Lindhof, Thomas Ludwig, Peter Luksch, Roy Milner, Bernhard Ries, Thomas Treml: TOPSYS - Tools for Parallel Systems (Artikelsammlung); 2., erweiterte Auflage
342/14/91 A	Michael Griebel: The combination technique for the sparse grid solution of PDE's on multiprocessor machines
342/15/91 A	Thomas F. Gritzner, Manfred Broy: A Link Between Process Algebras and Abstract Relation Algebras?
342/16/91 A	Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Thomas Treml, Roland Wismüller: The Design and Implementation of TOPSYS
342/17/91 A	Ulrich Furbach: Answers for disjunctive logic programs
342/18/91 A	Ulrich Furbach: Splitting as a source of parallelism in disjunctive logic programs
342/19/91 A	Gerhard W. Zumbusch: Adaptive parallele Multilevel-Methoden zur Lösung elliptischer Randwertprobleme
342/20/91 A	M. Jobmann, J. Schumann: Modelling and Performance Analysis of a Parallel Theorem Prover
342/21/91 A	Hans-Joachim Bungartz: An Adaptive Poisson Solver Using Hierarchical Bases and Sparse Grids
342/22/91 A	Wolfgang Ertel, Theodor Gemenis, Johann M. Ph. Schumann, Christian B. Suttner, Rainer Weber, Zongyan Qiu: Formalisms and Languages for Specifying Parallel Inference Systems
342/23/91 A	Astrid Kiehn: Local and Global Causes

Reihe A

342/24/91 A	Johann M.Ph. Schumann: Parallelization of Inference Systems by using an Abstract Machine
342/25/91 A	Eike Jessen: Speedup Analysis by Hierarchical Load Decomposition
342/26/91 A	Thomas F. Gritzner: A Simple Toy Example of a Distributed System: On the Design of a Connecting Switch
342/27/91 A	Thomas Schnekenburger, Andreas Weininger, Michael Friedrich: Introduction to the Parallel and Distributed Programming Language ParMod-C
342/28/91 A	Claus Dendorfer: Funktionale Modellierung eines Postsystems
342/29/91 A	Michael Griebel: Multilevel algorithms considered as iterative methods on indefinite systems
342/30/91 A	W. Reisig: Parallel Composition of Liveness
342/31/91 A	Thomas Bemmerl, Christian Kasperbauer, Martin Mairandres, Bernhard Ries: Programming Tools for Distributed Multiprocessor Computing Environments
342/32/91 A	Frank Leßke: On constructive specifications of abstract data types using temporal logic
342/1/92 A	L. Kanal, C.B. Suttner (Editors): Informal Proceedings of the Workshop on Parallel Processing for AI
342/2/92 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS
342/2-2/92 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: The Design of Distributed Systems - An Introduction to FOCUS - Revised Version (erschienen im Januar 1993)
342/3/92 A	Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, Rainer Weber: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems
342/4/92 A	Claus Dendorfer, Rainer Weber: Development and Implementation of a Communication Protocol - An Exercise in FOCUS
342/5/92 A	Michael Friedrich: Sprachmittel und Werkzeuge zur Unterstützung paralleler und verteilter Programmierung
342/6/92 A	Thomas F. Gritzner: The Action Graph Model as a Link between Abstract Relation Algebras and Process-Algebraic Specifications
342/7/92 A	Sergei Gorlatch: Parallel Program Development for a Recursive Numerical Algorithm: a Case Study
342/8/92 A	Henning Spruth, Georg Sigl, Frank Johannes: Parallel Algorithms for Slicing Based Final Placement
342/9/92 A	Herbert Bauer, Christian Sporrer, Thomas Krodel: On Distributed Logic Simulation Using Time Warp
342/10/92 A	H. Bungartz, M. Griebel, U. Rüde: Extrapolation, Combination and Sparse Grid Techniques for Elliptic Boundary Value Problems

Reihe A

342/11/92 A M. Griebel, W. Huber, U. Rüde, T. Störtkuhl: The Combination Technique for Parallel Sparse-Grid-Preconditioning and -Solution of PDEs on Multiprocessor Machines and Workstation Networks

342/12/92 A Rolf Niedermeier, Peter Rossmanith: Optimal Parallel Algorithms for Computing Recursively Defined Functions

342/13/92 A Rainer Weber: Eine Methodik für die formale Anforderungsspezifkation verteilter Systeme

342/14/92 A Michael Griebel: Grid– and point–oriented multilevel algorithms

342/15/92 A M. Griebel, C. Zenger, S. Zimmer: Improved multilevel algorithms for full and sparse grid problems

342/16/92 A J. Desel, D. Gomm, E. Kindler, B. Paech, R. Walter: Bausteine eines kompositionalen Beweiskalküls für netzmodellierte Systeme

342/17/92 A Frank Dederichs: Transformation verteilter Systeme: Von applikativen zu prozeduralen Darstellungen

342/18/92 A Andreas Listl, Markus Pawlowski: Parallel Cache Management of a RDBMS

342/19/92 A Erwin Loibl, Markus Pawlowski, Christian Roth: PART: A Parallel Relational Toolbox as Basis for the Optimization and Interpretation of Parallel Queries

342/20/92 A Jörg Desel, Wolfgang Reisig: The Synthesis Problem of Petri Nets

342/21/92 A Robert Balder, Christoph Zenger: The d-dimensional Helmholtz equation on sparse Grids

342/22/92 A Ilko Michler: Neuronale Netzwerk-Paradigmen zum Erlernen von Heuristiken

342/23/92 A Wolfgang Reisig: Elements of a Temporal Logic. Coping with Concurrency

342/24/92 A T. Störtkuhl, Chr. Zenger, S. Zimmer: An asymptotic solution for the singularity at the angular point of the lid driven cavity

342/25/92 A Ekkart Kindler: Invariants, Compositionality and Substitution

342/26/92 A Thomas Bonk, Ulrich Rüde: Performance Analysis and Optimization of Numerically Intensive Programs

342/1/93 A M. Griebel, V. Thurner: The Efficient Solution of Fluid Dynamics Problems by the Combination Technique

342/2/93 A Ketil Stølen, Frank Dederichs, Rainer Weber: Assumption / Commitment Rules for Networks of Asynchronously Communicating Agents

342/3/93 A Thomas Schnekenburger: A Definition of Efficiency of Parallel Programs in Multi-Tasking Environments

342/4/93 A Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: A Proof of Convergence for the Combination Technique for the Laplace Equation Using Tools of Symbolic Computation

Reihe A

342/5/93 A	Manfred Kunde, Rolf Niedermeier, Peter Rossmanith: Faster Sorting and Routing on Grids with Diagonals
342/6/93 A	Michael Griebel, Peter Oswald: Remarks on the Abstract Theory of Additive and Multiplicative Schwarz Algorithms
342/7/93 A	Christian Sporrer, Herbert Bauer: Corolla Partitioning for Distributed Logic Simulation of VLSI Circuits
342/8/93 A	Herbert Bauer, Christian Sporrer: Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving
342/9/93 A	Peter Slavkovsky: The Visibility Problem for Single-Valued Surface (z = f(x,y)): The Analysis and the Parallelization of Algorithms
342/10/93 A	Ulrich Rüde: Multilevel, Extrapolation, and Sparse Grid Methods
342/11/93 A	Hans Regler, Ulrich Rüde: Layout Optimization with Algebraic Multigrid Methods
342/12/93 A	Dieter Barnard, Angelika Mader: Model Checking for the Modal Mu-Calculus using Gauß Elimination
342/13/93 A	Christoph Pflaum, Ulrich Rüde: Gauß' Adaptive Relaxation for the Multilevel Solution of Partial Differential Equations on Sparse Grids
342/14/93 A	Christoph Pflaum: Convergence of the Combination Technique for the Finite Element Solution of Poisson's Equation
342/15/93 A	Michael Luby, Wolfgang Ertel: Optimal Parallelization of Las Vegas Algorithms
342/16/93 A	Hans-Joachim Bungartz, Michael Griebel, Dierk Röschke, Christoph Zenger: Pointwise Convergence of the Combination Technique for Laplace's Equation
342/17/93 A	Georg Stellner, Matthias Schumann, Stefan Lamberts, Thomas Ludwig, Arndt Bode, Martin Kiehl und Rainer Mehlhorn: Developing Multicomputer Applications on Networks of Workstations Using NXLib
342/18/93 A	Max Fuchs, Ketil Stølen: Development of a Distributed Min/Max Component
342/19/93 A	Johann K. Obermaier: Recovery and Transaction Management in Write-optimized Database Systems
342/20/93 A	Sergej Gorlatch: Deriving Efficient Parallel Programs by Systemating Coarsing Specification Parallelism
342/01/94 A	Reiner Hüttl, Michael Schneider: Parallel Adaptive Numerical Simulation
342/02/94 A	Henning Spruth, Frank Johannes: Parallel Routing of VLSI Circuits Based on Net Independency
342/03/94 A	Henning Spruth, Frank Johannes, Kurt Antreich: PHIroute: A Parallel Hierarchical Sea-of-Gates Router

Reihe A

342/04/94 A	Martin Kiehl, Rainer Mehlhorn, Matthias Schumann: Parallel Multiple Shooting for Optimal Control Problems Under NX/2
342/05/94 A	Christian Suttner, Christoph Goller, Peter Krauss, Klaus-Jörn Lange, Ludwig Thomas, Thomas Schnekenburger: Heuristic Optimization of Parallel Computations
342/06/94 A	Andreas Listl: Using Subpages for Cache Coherency Control in Parallel Database Systems
342/07/94 A	Manfred Broy, Ketil Stølen: Specification and Refinement of Finite Dataflow Networks - a Relational Approach
342/08/94 A	Katharina Spies: Funktionale Spezifikation eines Kommunikationsprotokolls
342/09/94 A	Peter A. Krauss: Applying a New Search Space Partitioning Method to Parallel Test Generation for Sequential Circuits
342/10/94 A	Manfred Broy: A Functional Rephrasing of the Assumption/Commitment Specification Style
342/11/94 A	Eckhardt Holz, Ketil Stølen: An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus
342/12/94 A	Christoph Pflaum: A Multi-Level-Algorithm for the Finite-Element-Solution of General Second Order Elliptic Differential Equations on Adaptive Sparse Grids
342/13/94 A	Manfred Broy, Max Fuchs, Thomas F. Gritzner, Bernhard Schätz, Katharina Spies, Ketil Stølen: Summary of Case Studies in FOCUS - a Design Method for Distributed Systems
342/14/94 A	Maximilian Fuchs: Technologieabhängigkeit von Spezifikationen digitaler Hardware
342/15/94 A	M. Griebel, P. Oswald: Tensor Product Type Subspace Splittings And Multilevel Iterative Methods For Anisotropic Problems
342/16/94 A	Gheorghe Ştefănescu: Algebra of Flownomials
342/17/94 A	Ketil Stølen: A Refinement Relation Supporting the Transition from Unbounded to Bounded Communication Buffers
342/18/94 A	Michael Griebel, Tilman Neuhoeffer: A Domain-Oriented Multilevel Algorithm-Implementation and Parallelization
342/19/94 A	Michael Griebel, Walter Huber: Turbulence Simulation on Sparse Grids Using the Combination Method
342/20/94 A	Johann Schumann: Using the Theorem Prover SETHEO for verifying the development of a Communication Protocol in FOCUS - A Case Study -
342/01/95 A	Hans-Joachim Bungartz: Higher Order Finite Elements on Sparse Grids
342/02/95 A	Tao Zhang, Seonglim Kang, Lester R. Lipsky: The Performance of Parallel Computers: Order Statistics and Amdahl's Law
342/03/95 A	Lester R. Lipsky, Appie van de Liefvoort: Transformation of the Kronecker Product of Identical Servers to a Reduced Product Space

Reihe A

342/04/95 A	Pierre Fiorini, Lester R. Lipsky, Wen-Jung Hsin, Appie van de Liefvoort: Auto-Correlation of Lag-k For Customers Departing From Semi-Markov Processes
342/05/95 A	Sascha Hilgenfeldt, Robert Balder, Christoph Zenger: Sparse Grids: Applications to Multi-dimensional Schrödinger Problems
342/06/95 A	Maximilian Fuchs: Formal Design of a Model-N Counter
342/07/95 A	Hans-Joachim Bungartz, Stefan Schulte: Coupled Problems in Microsystem Technology
342/08/95 A	Alexander Pfaffinger: Parallel Communication on Workstation Networks with Complex Topologies
342/09/95 A	Ketil Stølen: Assumption/Commitment Rules for Data-flow Networks - with an Emphasis on Completeness
342/10/95 A	Ketil Stølen, Max Fuchs: A Formal Method for Hardware/Software Co-Design
342/11/95 A	Thomas Schnekenburger: The ALDY Load Distribution System
342/12/95 A	Javier Esparza, Stefan Römer, Walter Vogler: An Improvement of McMillan's Unfolding Algorithm
342/13/95 A	Stephan Melzer, Javier Esparza: Checking System Properties via Integer Programming
342/14/95 A	Radu Grosu, Ketil Stølen: A Denotational Model for Mobile Point-to-Point Dataflow Networks
342/15/95 A	Andrei Kovalyov, Javier Esparza: A Polynomial Algorithm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs
342/16/95 A	Bernhard Schätz, Katharina Spies: Formale Syntax zur logischen Kernsprache der Focus-Entwicklungsmethodik
342/17/95 A	Georg Stellner: Using CoCheck on a Network of Workstations
342/18/95 A	Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller: Workshop on PVM, MPI, Tools and Applications

SFB 342 :	Methoden und Werkzeuge für die Nutzung paralleler Rechnerarchitekturen

Reihe B

342/1/90 B	Wolfgang Reisig: Petri Nets and Algebraic Specifications
342/2/90 B	Jörg Desel: On Abstraction of Nets
342/3/90 B	Jörg Desel: Reduction and Design of Well-behaved Free-choice Systems
342/4/90 B	Franz Abstreiter, Michael Friedrich, Hans-Jürgen Plewan: Das Werkzeug runtime zur Beobachtung verteilter und paralleler Programme
342/1/91 B	Barbara Paech1: Concurrency as a Modality
342/2/91 B	Birgit Kandler, Markus Pawlowski: SAM: Eine Sortier- Toolbox -Anwenderbeschreibung
342/3/91 B	Erwin Loibl, Hans Obermaier, Markus Pawlowski: 2. Workshop über Parallelisierung von Datenbanksystemen
342/4/91 B	Werner Pohlmann: A Limitation of Distributed Simulation Methods
342/5/91 B	Dominik Gomm, Ekkart Kindler: A Weakly Coherent Virtually Shared Memory Scheme: Formal Specification and Analysis
342/6/91 B	Dominik Gomm, Ekkart Kindler: Causality Based Specification and Correctness Proof of a Virtually Shared Memory Scheme
342/7/91 B	W. Reisig: Concurrent Temporal Logic
342/1/92 B	Malte Grosse, Christian B. Suttner: A Parallel Algorithm for Set-of-Support Christian B. Suttner: Parallel Computation of Multiple Sets-of-Support
342/2/92 B	Arndt Bode, Hartmut Wedekind: Parallelrechner: Theorie, Hardware, Software, Anwendungen
342/1/93 B	Max Fuchs: Funktionale Spezifikation einer Geschwindigkeitsregelung
342/2/93 B	Ekkart Kindler: Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick
342/1/94 B	Andreas Listl; Thomas Schnekenburger; Michael Friedrich: Zum Entwurf eines Prototypen für MIDAS