# Software Environments for LAN Distributed Computing - A Comparison Study

K.Zieliński M.Gajęcki G. Czajkowski
Institute of Computer Science
University of Mining & Metallurgy

February 18, 1994

# Table of Contents

# Chapter 1

# Introduction

The LAN distributed computing belongs to a very hot area of research. Results of many projects reached recently a mature state and a rich set of software tools and programming languages for distributed processing is available on the market and in the public domain sector.

Parallel computing architecture could make local-area networks the most cost-effective platform [1] for high-performance computing applications such as on-line transaction processing, real-time graphics and video interaction. In the near future a CPU-brokering activity will be invisible to an average computer user. Likewise, the same user could send a complex application across the network to run on dozens of processors simultaneously and obtain rapid results, just as if all the work had been executed locally.

Currently a few available distributed computing environments support these capabilities to varying degrees [1, 30]. The ultimate acceptance of parallel network computing depends on the extent to which HP and SUN - the vendors of the Network Computing System (NCS) and Open Network Computing (ONC) , the two dominant Remote Procedure Call (RPC) environments - support parallel processing features such as those found in Network Linda, PVM and P4. They suffer from the lack of many key components of the high-performance network computing full support, including asynchronous RPC, naming/locator service and multithreading. This situation is going to improve when a new ONC+ environment will be released [1].

The goal of this report is to give some ideas about effectiveness of the LAN distributed computations. Among available software packages a six following tools were investigated: PVM, P4, ANSA, SR, Strand, and Linda. They have been chosen not only according to the availability rule but because they represent a different approach to the distributed processing.

The simplest solution such as PVM, offers only basic communication and synchronization functions. P4 does not introduce any new ideas with comparison to PVM but its communication primitives are a little more mature and much more efficient.

ANSA is a very large object oriented system that provides a whole infrastructure for an open distributed processing system development. The ANSA project seems to be predecessor of the project CORBA or DOE curried on by SUN and HP [51].

SR is a distributed language and a run-time system designed while having a distributed processing in mind. It provides a very rich set of communication and synchronization primitives and an elegant computational model.

Strand belongs to the family of the parallel logic programming languages. Because of simplifications and improvements it is far more efficient than many implementations of parallel Prolog known up to date. It is also a first available language that may be classified as a composition or a glue language.

Linda it is a simple communication library built around the tuple space concept. This leads to the declarative semantics of communication functions embedded into a pure imperative environment of C or Fortran.

The chosen software tools have been tested on different machines such as: SPARCStaion2, SUN SLC, IBM RISC 6000 connected by 10Mbps Ethernet. This gives an idea about the hardware performance and the operating system communication layer efficiency and their influence on the features of a network computer created from the given type machines.

The scope of the report is as follows. In Chapter 2 investigated software tools have been described. The potential limitations of their computational models have been discussed. Next in Chapter 3, the scenario of evaluation experiments has been described. The hardware and software configurations have been specified in details. They have been divided into a communication test and a case study that concerns a processing farm implementation. In Chapter 4 the experimental results have been presented and discussed. The comparison of the investigated software has been done in Chapter 5. The report has been summarized in Chapter 6. Examples of the test programs and detailed result tables were given in Appendices.

# Chapter 2

# Software for LAN Distributed Computing Overview

The investigated software tools for distributed processing belong to a very rapidly growing family of software. Nearly every week a new software environment for distributed processing is advertised. To this family belongs: Charm [9], Mentat [10], PCN [11] Hetero [12], Parform [13], Arcade [14], PHARROS [15], DAPHNE [16], Marionette [17], Spawn [18], Durra [19], Merlin [20], Concert [21], Enterprise and many others. Those cited above grab more or less attention of the distributed computing community. The selection of the six software tools under study nevertheless covers the basic computational models and the main trends in the distributed processing development.

Despite of the fact a great number of packages is currently available a new area of distributed processing would be opened soon. A modern operating systems with a full multithreading support, such as SOLARIS 2.2 [51] would created a new platform for the efficient distributed processing. In this circumstances the presented report is same kind of snapshot of a very rapidly changing domain of the Computer Science. The investigated software packages have some history of applications and belong to a very often cited elsewhere.

## 2.1 An overview of the PVM System

PVM (Parallel Virtual Machine) is a library of macros and subroutines developed at the Oak Ridge National Laboratory for programming a variety of parallel machines [52]. PVM is composed of a suite of user-interface primitives and supporting software that together enable the concurrent computing on a loosely coupled network of processing elements. 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. They view the PVM system as a flexible parallel computing resource that supports message passing, shard memory, and hybrid models of computation. Inter-instance communication construction include those for the exchange of data structure as well as high-level primitives such as broadcast, barrier synchronization, mutual exclusion, global extrema, and rendezvous. Hence the cooperation between modules may be organized on communication basis. No higher level abstractions as, for instance RCP, are provided.

Application programs under PVM may possess arbitrary control and dependency structures.

]

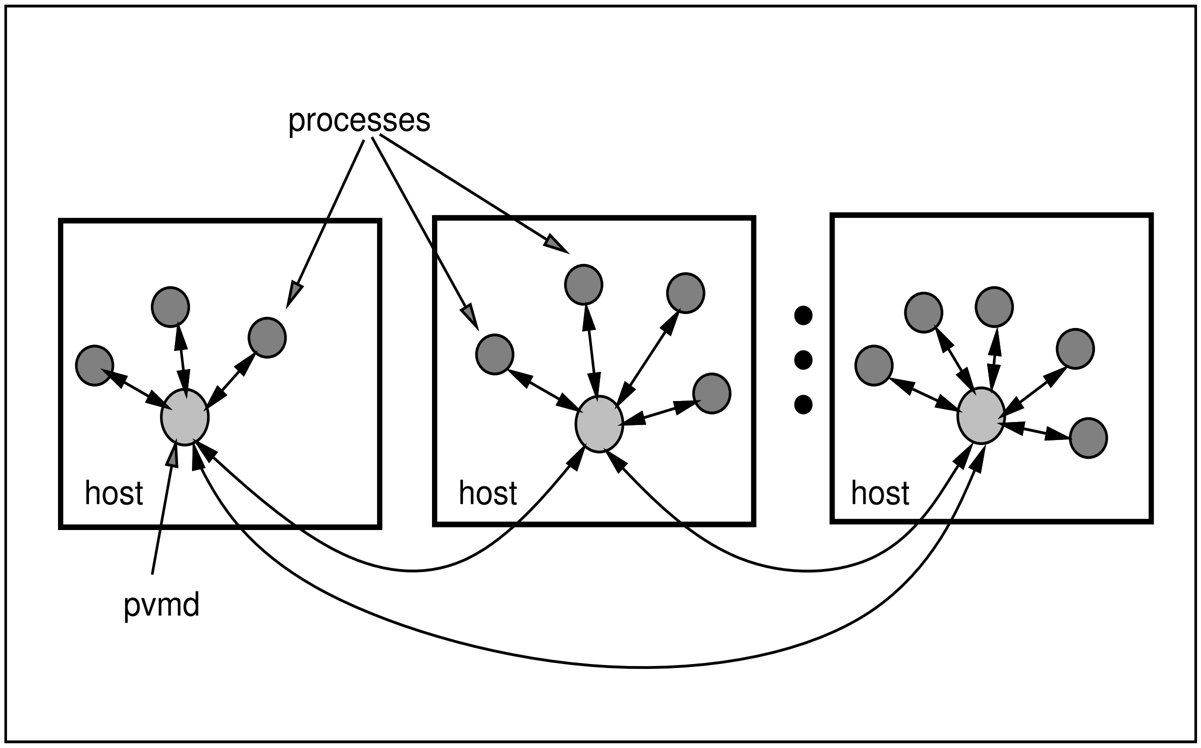


Figure 2.1: PVM computational model

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 on , any process may communicate and/or synchronize with any other. This is the most unstructured form of crowd computation, but in practice a significant number of concurrent applications are more structured. Two typical structures are the tree and the processor farm structure. The latter term is used to denote computations in which each computational process is identical and such applications also exhibit regular communication and synchronization patterns.

The major disadvantage of the PVM environment is that processing elements corresponds to heavyweight single threaded UNIX processes. Such a process is created by the execution of the typical C or Fortran program which contains invocations to the PVM library. Hence the blocking semantics of some communication operations stop the whole process. It limits an extent to which the potential parallelism of the application may be practically used.

### 2.1.1 PVM Computational Model

In the PVM system, an application is considered to consist of components. From the system point of view, a component corresponds to an object file that is is capable of being executed as a user-level process. A component is therefore a static entity and is identified by a name. A complete description of application components, i.e. the component name and all corresponding executable is obtained by the PVM support software. This software executes as a user-level process on each host in the participant pool. An initial set of participating hosts is statically identified; additions or deletions are possible during operation by means of an administration interface. The PVM support process termed **pvmd** on a host is responsible for all application component processes executing on that host. However, there is no central or master instance of **pvmd**. Control is completely distributed and all **pvmd** processes possess global knowledge in the interest of avoiding performance bottlenecks.

Across the network, **pvmd** processes communicate using UDP datagrams using the "well known port" approach for addressing. Each **pvmd** process maintains information components processes. A user **send** is addressed to a component name and instance number. The local **pvmd** determines the physical location of that process and forwards the message to the remote **pvmd**. The similar procedure concerns broadcast send operation, i.e. user process broadcasts are first delivered sequentially to local recipient user processes after which the local **pvmd** process broadcasts over the network to all other **pvmd** processes that in turn deliver the message to their local user process recipients.

Mutual exclusion is another primitive required both in response to user request as well as for **pvmd** coordination. All **pvmd** processes possess knowledge regarding the use of all locks, such an attempt will, of course, only be made when a lock is known to be free at the start of the claim.

## 2.2 An overview of P4 System

P4 is a library of macros and subroutines developed at Argonne National Laboratory for programming a variety of parallel machines [50]. The system provides two basic computational models: monitors for the shared-memory model, message-passing for the distributed-memory, and support for combining the both. P4 is intended to be portable, simple to install and use, and efficient. It can be used to program networks of workstations, advanced parallel supercomputers like the Intel Touchstone Delta and the Alliant Campus HiPPI-based system, and single shared-memory multiprocessors. It has currently been installed on more than 25 different systems with SUN SPARC, IBM RS6000 and CONVEX among them. Although P4 tries to be completely portable, there are a small number of specific exceptions that may need to be taken into account on a gvien machine.

The current release is version 1.2 and provides:

- **xdr** support for heterogeneous networks,
- SYS IPC support added for several machines (for shared-memory multiprocessing on workstations that support multiple processors),
- instrumentation added for automatic logging/tracing.
- automatic or user control of message-passing/buffer-management,
- high-resolution clock support for several machines,
- error/interrupt handling,
- an optional secure server for quick startup.

### 2.2.1 P4 Computational Model

The P4 supports the cluster model of parallel computation, in which subsets of processes share memory with one another, with the clusters communicating via messages. There are two procedures that can be used to create processes in P4, *p4_create_procgroup* and *p4_create* . Processes created via *p4_create_procgroup* are "P4-managed" whereas those created via *p4_create* are said to be "user-managed". The P4-managed have message queues allocated for them so that they can do message-passing, and they are able to run either on a shared-memory multiprocessor with the creating process or they can run on a separate machine. Processes created via *p4_create* cannot do message-passing, and they can only run on a shared-memory multiprocessor with the creating process.

The only disadvantage of *p4_create_procgroup* is that it must be built a 'procgroup' file describing the set of required slave processes before the master program begins execution. This eliminates the possibility of determining late in the execution exactly how many processes you want to use to solve a problem. Generally, this is not a problem, especially since is possible to combine these procedure in the following way: Use *p4_create_procgroup* to develop a network of processes that talk to each other via messages and next those processes can create further processes to help it out as necessary.

A procgroup file for a program written for the cluster model might look like this:

```
local 4
alliant1.abc.edu        5    /home/me/myprog
alliant2.abc.edu        3    /home/me/myprog
encore.somewhere.edu    2    /home/me/myprog
```

The corresponding allocation of processes is shown in Fig. 2.2

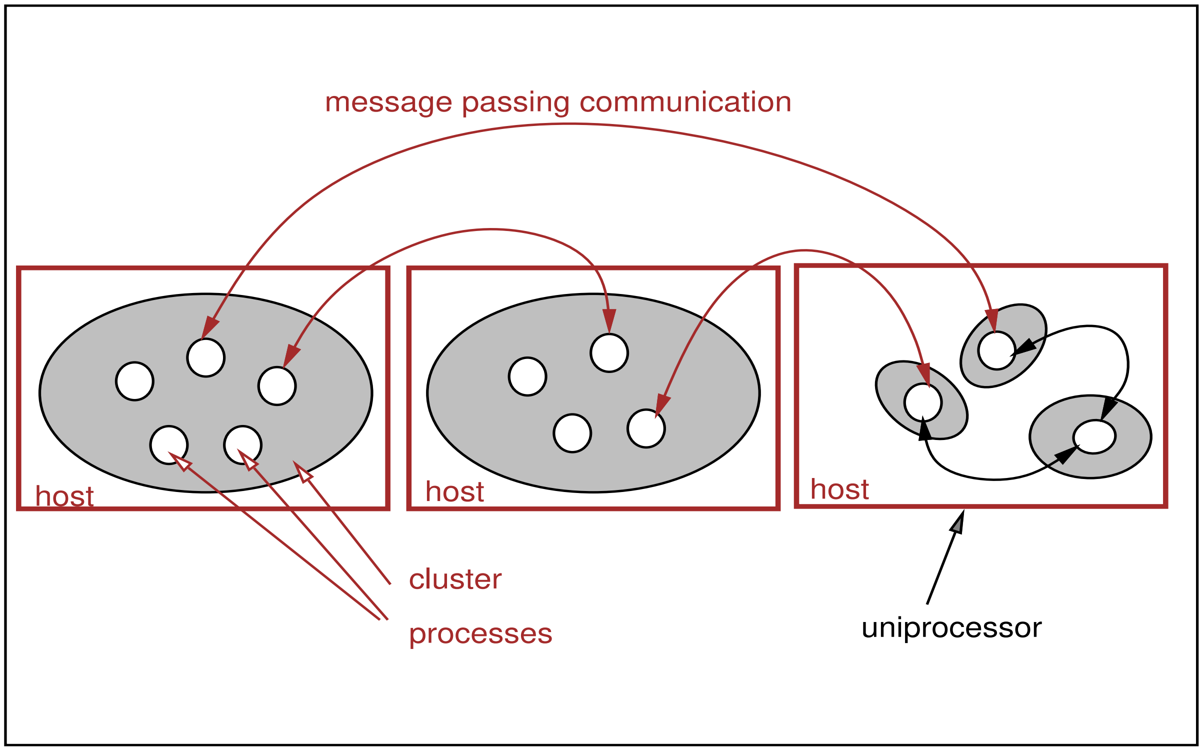


Figure 2.2: Cluster model of parallel computation

Every line in a procgroup file specifies the cluster allocate to one machine. Communication inside a cluster is performed via shared memory.

Some notes about contents of the procgroup file should be made at this point. First, the value of **n** on the local line can be zero, i.e. the master may have no local slaves. Second, the local machine may be treated as if it is a remote machine by merely entering it in some line as remote machine. Third, a single machine may be treated as multiple remote machine by having the same remote machine name entered on multiple lines in the procgroup file. Fourth, if a single machine is listed multiple times, those processes specified on each line form a single cluster (share memory). Fifth, the cluster size specified for a uniprocessor should be 1, because all slaves in a cluster are assumed to run in parallel and share memory.

P4 processes on remote machine are ordinarily created by *rsh*. Since *rsh* is relatively slow, P4 provides a way to get things started faster by running the program *serv_p4* in the background on the remote machine. Note that a start-up time must be added to the global time of the particular computation performing.

### 2.2.2 Overview of the Library

The P4 procedures are gathered into the following groups:

- Functions for managing processes and clusters,
- Functions for message passing,
- Functions for shared memory,
- Miscellaneous functions for:global operations, timing, debugging and etc P4 programs,

The most important functions for managing processes and clusters have already described.

Functions for message passing provide blocking e.i. rendezvous and nonblocking e.i. asynchronous semantics of the send procedures. These procedures are also supported in the version that call XDR for data conversion if the message is being passed to a machine of a different architecture.

P4 supports a number message passing operations for dealing with all processes at once called global operations. They apply the commutative operation on a element-by-element basis and broadcast the result to all nodes.

The abstraction provided by P4 for managing data in shared memory is monitors. P4 provides several useful monitors like *p4_lock*, *p4_unlock*, *p4_barier_t*, *p4_getsub_monitor_t*, *p4_askfor_monitor* as well as a general monitor type to help the user in constructing his own monitors.

Remaining P4 library functions concern various aspects of program debugging and timing. They make P4 a very useful programming tool.

## 2.3 An overview of the ANSA System

The Advanced Networked Systems Architecture (ANSA) [53, 46, 47] originated in a project undertaken by BT, DEC, GEC, HP, ICL, ITL, Olivetti, Plessey within the UK Alvey Information technology Programme. Much of the work is currently founded via the EC ESPRIT II Programme within a project called ISA. The architecture continues to be known as ANSA and is traded by APM Ltd. in Cambridge U.K.

One of the primary goals for ANSA is to simplify the design, construction, deployment, and maintenance of distributed applications.

A typical distributed application is constructed from several cooperating components. The interaction model supported by ANSA forces all interactions between components into a remote procedure call paradigm; a client of a service invokes an operation defined in the interface specification, with the server performing the operation at some later time. Two forms of remote invocation are supported : 1) interrogation, on which the invoking client activity waits for the server to perform the operation and return any results, and 2) announcement, in which the invoking client activity does not wait for the server to perform the operation.

The blocking nature of interrogations limits the performance of systems unless components with internal concurrency can be constructed. ANSA supports concurrency through the provision of a threads package. A component may *fork* additional threads for subsequent rendezvous using *join*. It may also *spawn* threads that execute totally independently to the parent thread.

ANSA provides event counters and sequencers as synchronization primitives from which critical regions can be constructed to protect concurrent access to a component's state by its threads.

One or more of components making up a distributed application may be provided as an independent, general service in the distributed environment. The binding may take place at different times in the life of the application. The latest time at which binding may occur is runtime, i.e dynamic binding. In ANSA the Trader provides a rendezvous mechanism for dynamic binding to well-known or published services.

Many distributed applications depend upon the dynamic instantiation of objects as the application proceeds. ANSA provides factories for object creation.

The features of ANSA described so far shown that this architecture may be classified as an open distributed computing environment in contrast to the PVM and P4 which are a typical run-time systems. The investigated software packages provide different models of distributed computation. The PVM and P4 assume that distributed computation are organized as a group of separate UNIX processes communicating through message passing mechanisms. The ANSA native model is based on the client-server paradigm with the RCP as an elementary communication and synchronization mechanism. The services provided by the system are accessible through operations invocation on distributed objects interfaces.

### 2.3.1 ANSA computational model

The major components of ANSA are depicted in Fig. 2.3 and their functions may be specified as follows:

The trader provides a directory structure that can be search by path name, by property values, or by some combination of the two. A server can expert an interface reference to the trader to make it accessible to other applications. The process of matching service offers with service requests is termed trading.

The factory is responsible for dynamic creation of application components. Interaction with an factory results in the creation of a capsule with the ability to instantiate and terminate objects of the required type.

The node manager provides configuration facilities for distributed applications, by providing an architectural interface for the creation, simple monitoring and destruction of ANSA services on a single node.

Each application components represents a capsule with transparencies services and objects

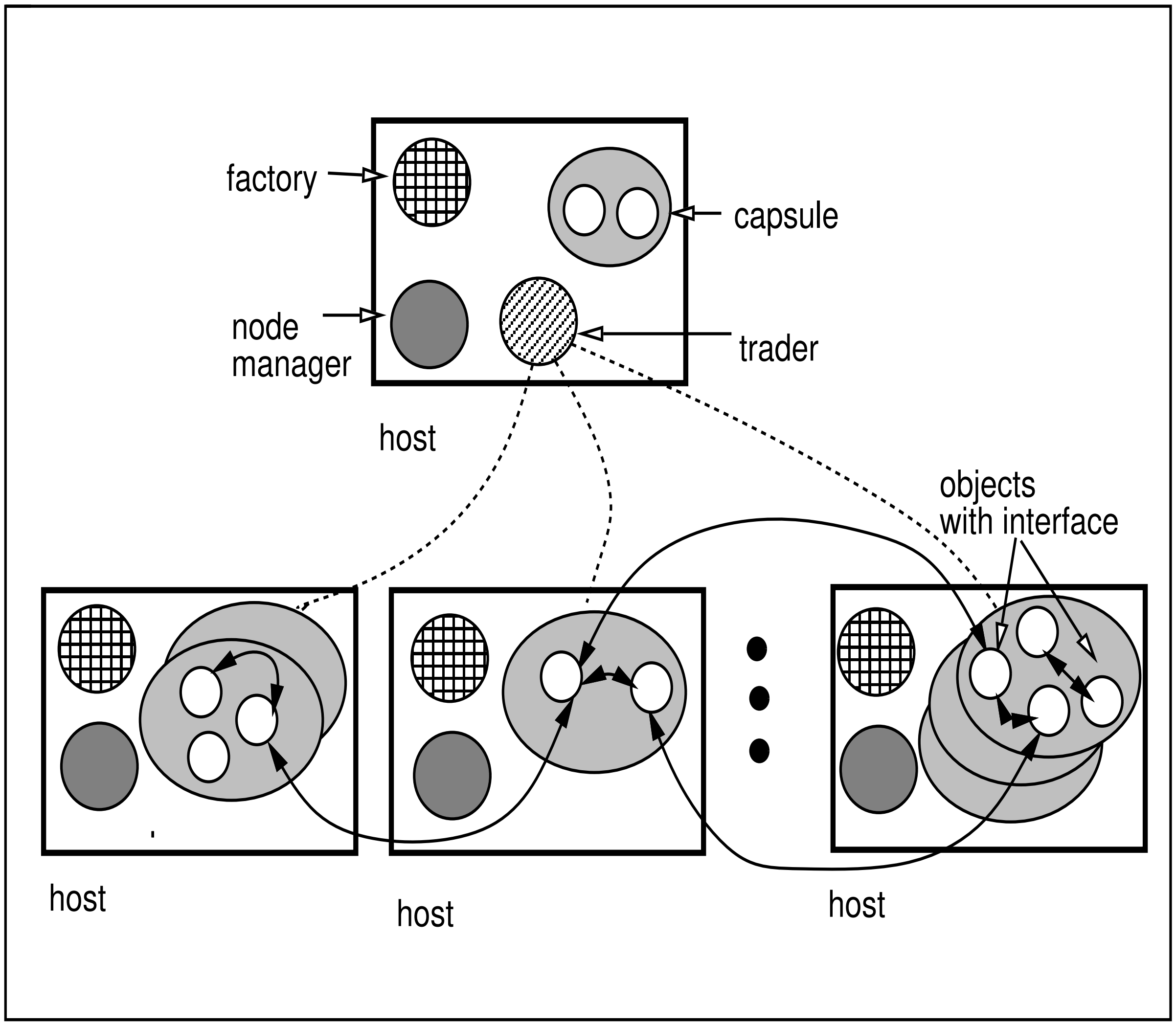


Figure 2.3: ANSA computational model

inside. Operations invoked on objects interfaces are performed in multithreading environment provided by the ANSA capsule.

## 2.4 An overview of the SR System

The SR concurrent programming language is developed at the University of Arizona for programming a variety of parallel machines. The earliest version, now called SR0, contained mechanisms for asynchronous message passing and rendezvous. This version has been redesign to provide additional mechanisms for remote procedure call, dynamic process creation, and semaphores, as well as a means for specifying distribution of program modules. The resent Version 2.0 described in this paper provides support for different execution environments, ranging from shared-memory multiprocessors to distributed systems [22, 23, 24, 25, 26, 27].

### 2.4.1 SR Computational Model

An SR program can execute within multiple address spaces, which can be located on multiple physical machines. Processes within a single address space can also share objects. Thus, SR supports programming in distributed environments as well as in shared-memory environments.

The SR model of computation allows a program to be split into one or more address spaces called virtual machines. Each virtual machines defines an address spaces an address space on one physical machine. Virtual machines are created dynamically; they are referenced indirectly through capability variables. Virtual machines contain instances of two related kinds of modular components: globals and resources.

The spec of global or resource can contain declarations of types, constants, and operations. A global's spec can additionally contain declarations of variables. An operation defines a service that must be provided somewhere in the program. It can be considered a generalization of a procedure: it has a name, and take parameters and return a result. An operation declared in a resource's spec must be serviced in that resource's body. Similarly, an operation declared in a global's spec can be serviced in the global's body; it can also be serviced within an importing resource or global.

The body of a global or a resource can contain declarations of additional objects are visible only within the body, not to any importer. Bodies also contain code that, among other things, services operations. The code is split into units called processes and procs. Processes are created implicitly when the enclosing global or resource is created. Instances of procs are created when they are invoked: they too execute as independent processes. All processes created within a global or a resource execute on the same virtual machine on which the enclosing global resource was created. Processes and procs can declare additional variables and operations; they must contain the code that services invocations of any locally declared operations.

Figure 1 summarizes SR's model of computation. In its form, a program consists of a single virtual machine executing on one physical machine, possibly a shared-memory multiprocessor. A program can also consist of multiple virtual machines executing on multiple physical machines. Hybrid forms are possible and in fact useful. Data and processors are shared within a virtual machine; different machines can be distributed across different physical machines.

Processes on the same or different virtual machines can communicate through operation invocation. This communication is independent of their virtual machine locations

### 2.4.2 Language Support

SR is rich in the functionality it provides for concurrent programming: dynamic process creation, semaphores, message passing, remote procedure call, and rendezvous.

The idea is that operations can be invoked in two ways, synchronously *call* or asynchronously *send*, and can be serviced in one of two ways, by props or by input statements *in*.

SR also provides two statements - return and reply - that provide additional flexibility in servicing invocations. A process executing a return statement causes the invocation being serviced to complete. The process that executes a reply statement then continues execution with the statement following the reply.

SR provides support for sharing on several levels. First, processes within a resources instance can share variables. They can coordinate access to shared variables through shared semaphores or other operations declared within the resource. Second, processes that execute in possible different resource instances but on the same virtual machine can share variables and operations declared in the spec of globals.

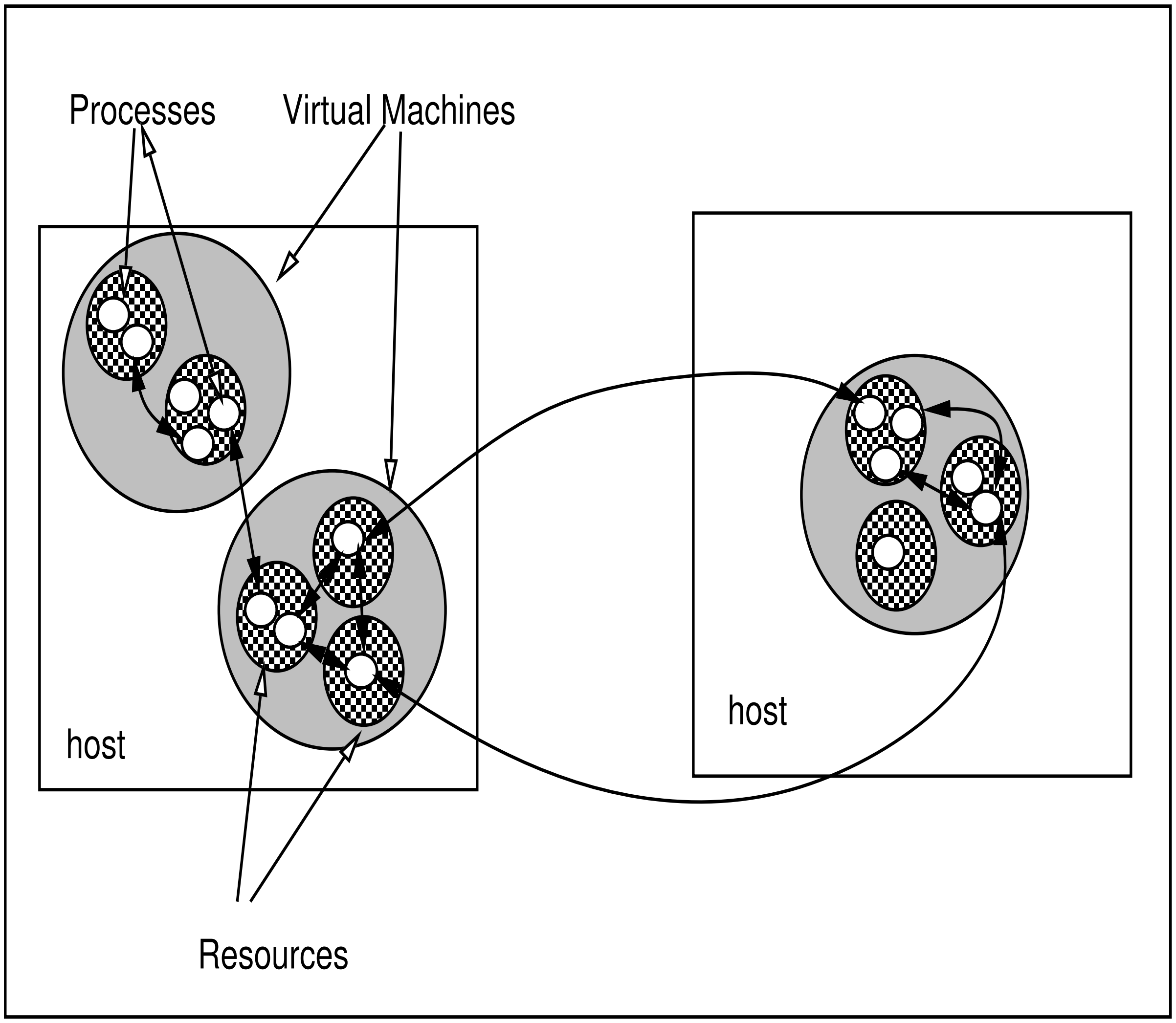


Figure 2.4: SR computational model

## 2.5 An overview of Strand 88 System

Strand 88 is a programming system for multiprocessing computers, produced by Strand Software Technologies Limited [32, 33]. Strand is based on a parallel model of execution. This provides it with parallel semantics that founded on a strong theoretical base. Therefore all the concurrency, flow of control and synchronization is expressed in the program without the need for understanding the hardware architecture or particular configuration. Strand handles a distribution of work dynamically.

### 2.5.1 Strand 88 computational model

The language consists of Strand kernels, Strand system calls and user procedures. Compilation of Strand code produces modules which are executed on the Strand Abstract Machine as it was shown in Fig. 2.5

Computation in Strand is performed by a set of interacting processes. Each process can be

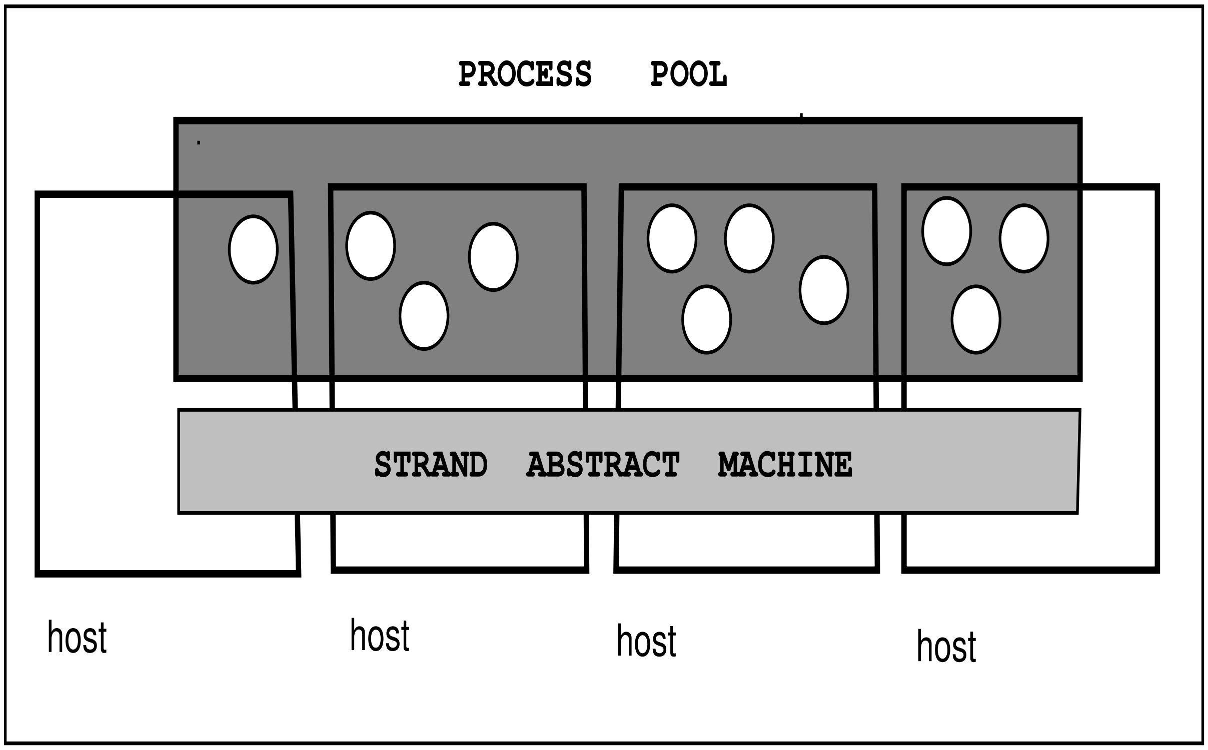


Figure 2.5: Strand 88 computational model

represented by a term of the form:

$$p(T_1, T_2, ..., T_n) \text{ n } \geq 0$$

where p identifies the program used to execute the process and $T_1, T_2, ..., T_n$ are the process arguments. A Strand program describes the actions that processes may perform. There are just three types of action: terminate, change state and fork. Hence a term process is used in Strand in very different meaning than in other software tools. The UNIX process is used for implementation one SAM on the given node and may contain several thousand of Strand processes.

Computation in Strand is performed by a pool of interacting, concurrent process. These processes communicate by assigning values to shared variables and synchronize by waiting for data to become available. These actions are specified by the collection of rules that constitute the program. Computation proceeds by repeatedly selecting a process and removing it from the pool. If the process is a predefined process, it is executed immediately; otherwise a reduction attempt is made. This involves selecting a rule from the program, matching the process to the rule head and executing the rule guard. If the preconditions specified by the head and guard are satisfied, then the process commits. This causes new copies of the processes defined in the rule body to be added to the process pool.

Basic feature of the Strand computation may by characterized as follows:

- At each step, the Strand interpreter nondeterministically selects a process from the process pool and rule from the program. The order in which processes execute is determined only by data availability, not their textual order in a program.

- All bodies of a rule under computation are performed concurrently.
- The reduction is made from left to right depth first. There is no unification-parallelism.
- Rule guards belongs to predefined set of predicate only and test variables may not be assigned by them.
- A variable can only be assigned once.
- The unification algorithm is asymetrical in that sense that may assign values to variables in the rule, but it may not assign values to variables in the process state.

A key feature of Strand is its foreign language interface. At present Strand 88 has interface for C and Fortran. The claim is made that existing application applications can be re-analyzed, through functional and data decomposition, and a Strand harness added, to allow the latent parallelism to be exploited.

## 2.6 An overview of Linda System

Linda is a generative communication paradigm, developed by David Gelernter and colleagues in the Department of Computer Science at Yale University [37, 38, 39, 40, 41] . The fundamental concept in Linda is the *tuple space*, an associative object store, populated by vectors of typed fields - *tuples* There are two types of tuple in tuple space: *passive* tuples (data) and *active* tuples (processes) which will become passive tuples on completion of execution.

Passive tuples are sequence of one or more typed fields. Each field has either an actual value or a formal pattern-matching value. Tuples are anonymous - they are neither owned by, or considered to have originated form, a particular process or processor. Tuple space can contain several identical tuples.

Tuple space is an associative memory and tuples in this space are acceded by pattern-matching on the values of their fields. Two tuples match if and only if they have the same name and number of arguments, and each pair of corresponding fields match. Two fields match if they have the same type and either they are both actuals with the same value, or one is an actual and one is a formal. A formal never matches another formal.

Linda is not a new language for parallel processing. Instead, Linda's operations are embedded in a sequential language to provide a dialect for parallel programming. Examples of tuples in C-Linda are **("foo"), (6, ?x, 9), (9, "fred")** . Note the use of the **?** prefix to denote a formal.

The tuple space abstraction has the following goals:

Simplicity The operation on tuple space supported by Linda are intended to be sufficiently simple to be added to any existing programming language to produce a hybrid parallel language.

Portability Linda aims to overcome the tremendous barrier to portability of parallel software caused by the great diversity of parallel software caused by great diversity of parallel hardware architectures by abstracting away from architectural details to provide a virtual parallel machine.

Scalability Decoupling processes though the tuple space make the same program possible to run on different number of processors without modification.

An illustration of the tuples space abstraction is shown in Fig. 2.6

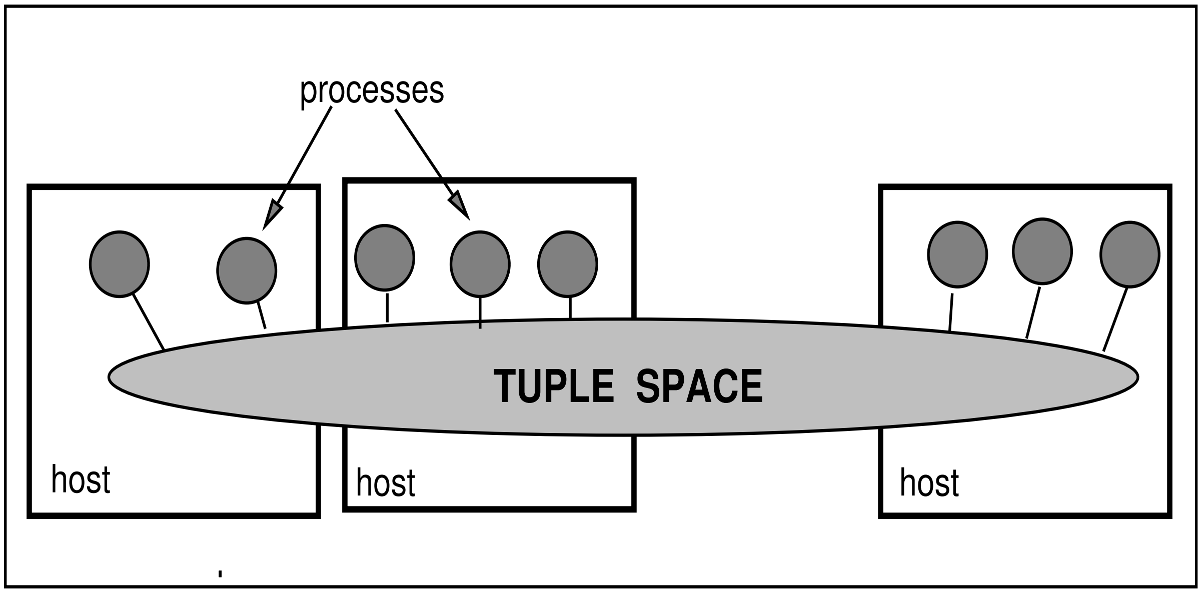


Figure 2.6: Linda computational model

### 2.6.1 Linda Computational Model

Linda provides six operations on the tuple space to support interprocess communication, shared data structures and process creation.

**out(t)**

A process adds a passive tuple t to the tuple space using **out(t)**. All the fields of the tuple are evaluated prior to the tuple being added to tuple space. Formals (host language variables) may occur in tuple fields.

The anonymity of tuples ensures that any subsequent matches with such a formal will have no effect on the corresponding variable in the process which out()ed the tuple. Hence formals in out()ed tuples are essentially place holder, or "don't care" values. Formals in out()ed tuples can be used to prevent some matches, since only templates with an actual in the corresponding field will match.

**in(s)**

This operation causes precisely one tuple t which matches the template s to be withdrawn from the tuple space. If there is not matching tuple space when **in(s)** is executed, the in()ing process is blocked until one becomes available.

Simple message passing between two processes can be implemented by **out()** and **in()**. The sender executes **out("destination", message)** and receiver executes **in("destination", ?message)**.

**rd(s)**

This operation is similar to **in(s)**, except that instead of removing the matching tuple from the tuple space, a copy is made to instantiate formals in s. Like **in(s)**, **rd(s)** blocks until a matching tuple is available.

**inp(s)** and **rdp(s)**

are the predicate forms of **in(s)** and **rd(s)** respectively. They behave similarly to their namesakes, but if no matching tuple is available these predicate operations immediately return failure rather than waiting for matching tuple to become available.

**eval()**

This operation allows active tuples to be added to tuple space. **eval()** is similar to **out()** except that the tuple's fields are evaluated after it is added to the tuple space rather than before. This evaluation implies the creation of a process to evaluate each of the tuple's fields. When all the fields have been evaluated, the eval()ed tuple becomes a passive tuple comprising the evaluated field.

### 2.6.2 Implementation issue

Some implementations of the network Linda as for instance the SCA Linda v.2.4.6 , we are currently using, has limitation with respect to the **eval()** operation implementation. In this version there is no way to run more than one **eval** concurrently on a single node. They could be run sequentially, but deadlock may occur. In fact SCA Network Linda **eval()** operation is useful for distribution rather heavy weight processes around the network nodes.

## 2.7 Summary

In this chapter only major features of the investigated software tools have been described. A more precise study may be found in the accompanied papers [44, 45, 47, 48, 49]. The presented review was oriented on comparison of the different computational models, from the pure imperative approach such as PVM and P-4 through the object oriented such as ANSA and SR to the declarative models as Strand 88 and the communication mechanisms in Linda.

# Chapter 3

# Scenario of Evaluation Experiments

The following classes of basic evaluation experiments have been performed:

- Communication efficiency evaluation,
- Case study that concerns two typical computational task with geometrical parallelism, i.e. : a fractal generation and a ray tracing.

The scenario of experiments has been described in the following sections. All of them have been performed on various hardware platforms and the six software packages described in the previous chapter.

## 3.1 Communication Efficiency Evaluation

A communication efficiency in distributed processing is a major factor that influence the whole computation performance. Its nature, pattern and frequency depends on a computational model. One of many possible scenarios assumes that data are distributed in the initial phase of computation and gathered in the final step. Very often distributed processes have to synchronize one with another by sending short messages with signals or partial results. This communication on a physical level is performed by sending messages of a given length over an existing communication medium.

The communication subsystem under study was built around the 10Mbps Ethernet LAN. A communication efficiency between two connected workstation depends on many factors among them, a data rate on a physical layer plays an important but very often not deciding role.A very substantial influence on effectiveness of communication on a process to process level have:

- Network interface design and modes of data handling it provides.
- Communication software implementations, i.e. device handler structure and implementation.
- Network software an interface such as the BSD Socket library or TLI efficiency.
- Choice of interface operations and a way they are used by an application software.

The factors pointed above make that efficiency of software for distributed computing is dependent substantially on the hardware and system software structure. This relation is very sophisticated and an experimental study leads to a final conclusion only.

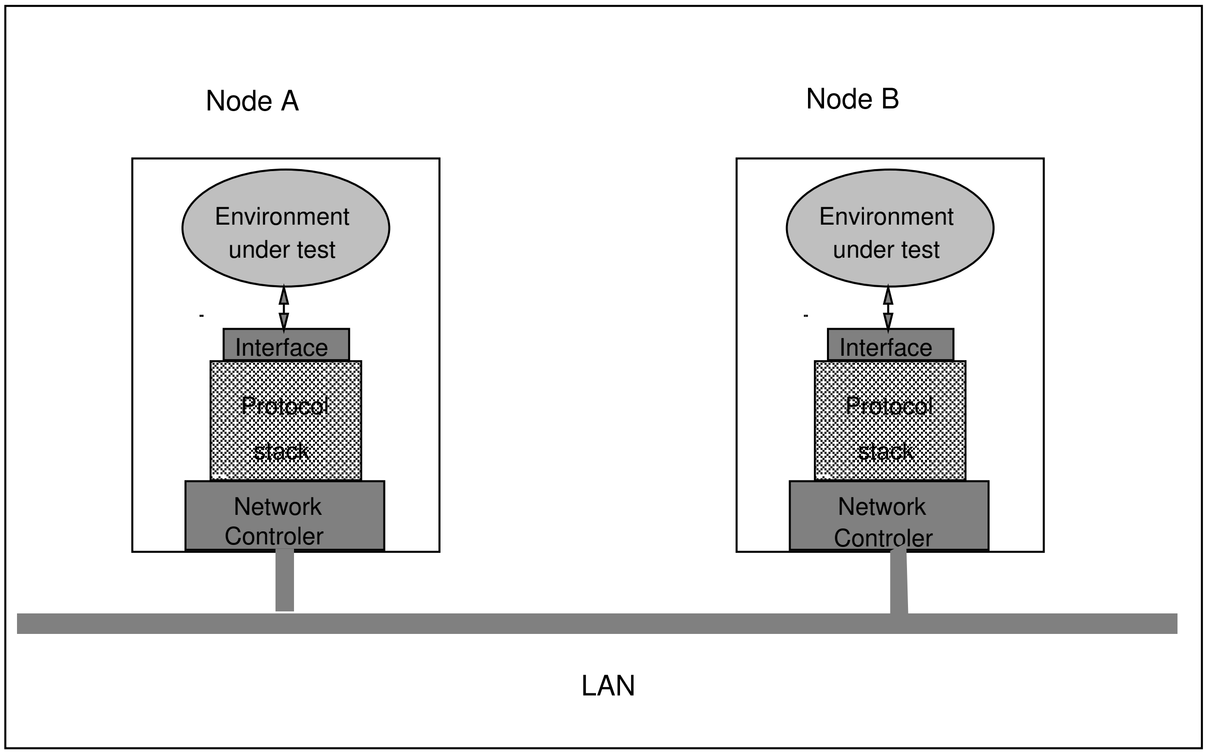


Figure 3.1: Communication test scenario

The scenario of the communication experiment reported in this paper has been shown in Fig. 3.1. Messages of various length are sent between two workstations of a given type. The send and receive operations were performed from the level of the given software package and repeated several times to improve accuracy of the measurement. For each length of messages the real time of a data transfer completion was measured as an average value of several repetitions.

## 3.2 Processor Farm

A processor farm computational model is based on the master/slave paradigm, shown in Fig. 3.2 This model is anthropomorphic: one logical processor, identified as the "master", is clearly superior to the others, his "slaves". In effect the master owns all logical processors and decides when and how to use each of them. When parallelization is appropriate, the master activates the desired number of slaves, then waits for them to complete their work before continuing. Typically, the master is responsible for balancing the load among available processors as well as managing all communication. A variant of the master-slave mode, called sometimes the processor-farm [2], uses a queue manager to keep the track of activities to be performed. As each logical processor, called the worker, finishes a task, it reports the results to the farmer and requests a new task; since tasks are distributed round-robin style as processors become available, load balancing is implicit.

The implementation of the processor farm computational model is very simple with the LAN based distributed computing. The nature of LAN communication systems ensure that no additional effort is necessary to provide direct communication of the one station with each other.

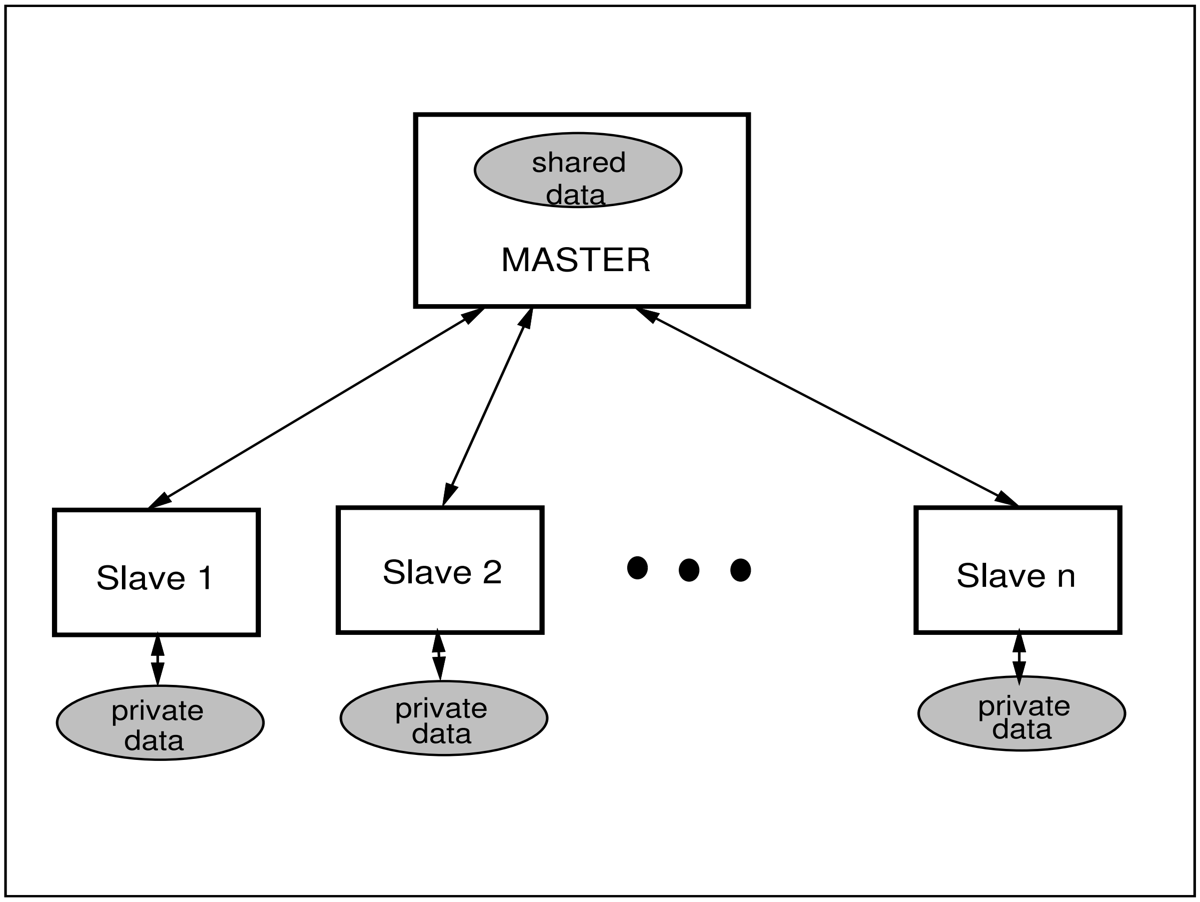


Figure 3.2: Master/slave model

Hence no complicated routing algorithms have to be applied.

Two structures of the processor farm basic model shown in Fig. 3.3 have been studied. Model A is more efficient than B if a number of workstations is not very big so the farmer process is not very busy and does not create the system bottleneck. The situation changes when a number of workstations increases and an additional worker task in the same node there the farmer process is performed starts to disturbe the farmer activity. The critical number of nodes to swap from model A to B depends on:

- tasks granularity, and
- workstations type and their communication to computation power ratio.

The bottleneck in the LAN based processor farm has two main reasons:

1. The farmer functions are centralized so when a number of worker processes increases a workstation running the farmer process getting to be overloaded,
2. The communication medium is a common resource that may be used only sequentially. When a number of workstations increases a communication contention seems to a problem.

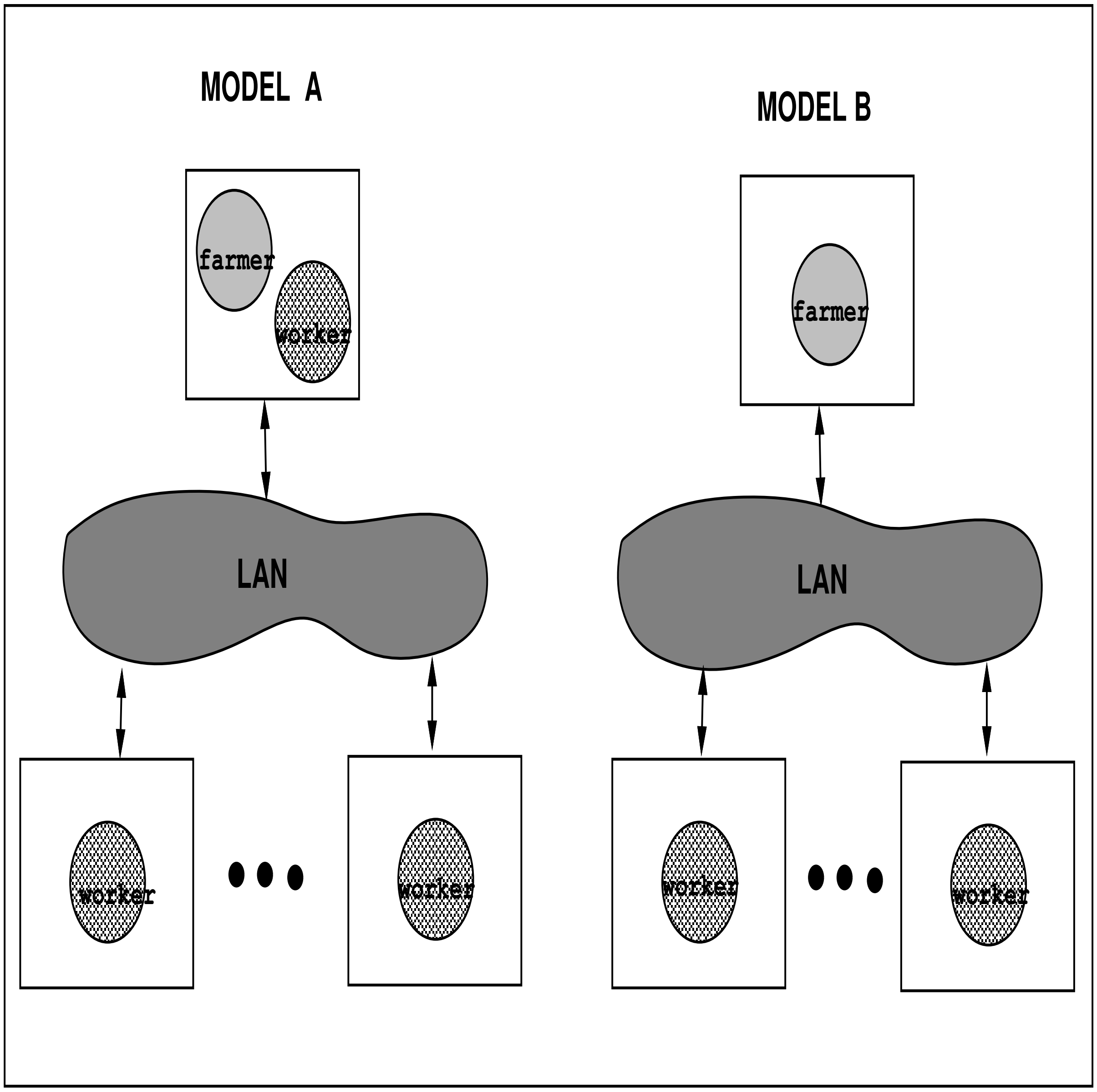


Figure 3.3: Two structures of a processor farm

The first problem may be solved using the hierarchical farm structure shown in Fig. 3.4. An application of the hierarchical farm structure needs a load sharing strategy between the farmer processes to be implemented.

The second problem may be partly overcome by a rebuilding of a communication subsystem using bridges. A communication cable should be divided into sections connected by bridges. A group of processes that create a subfarm should be allocated to the workstations connected to the same section of the cable. It makes possible to perform communication in parallel in the separate subfarms.

The implementation of the processor frame scheme in the various software environments under study has been shown in Appendix B. The attached programs implement exactly the

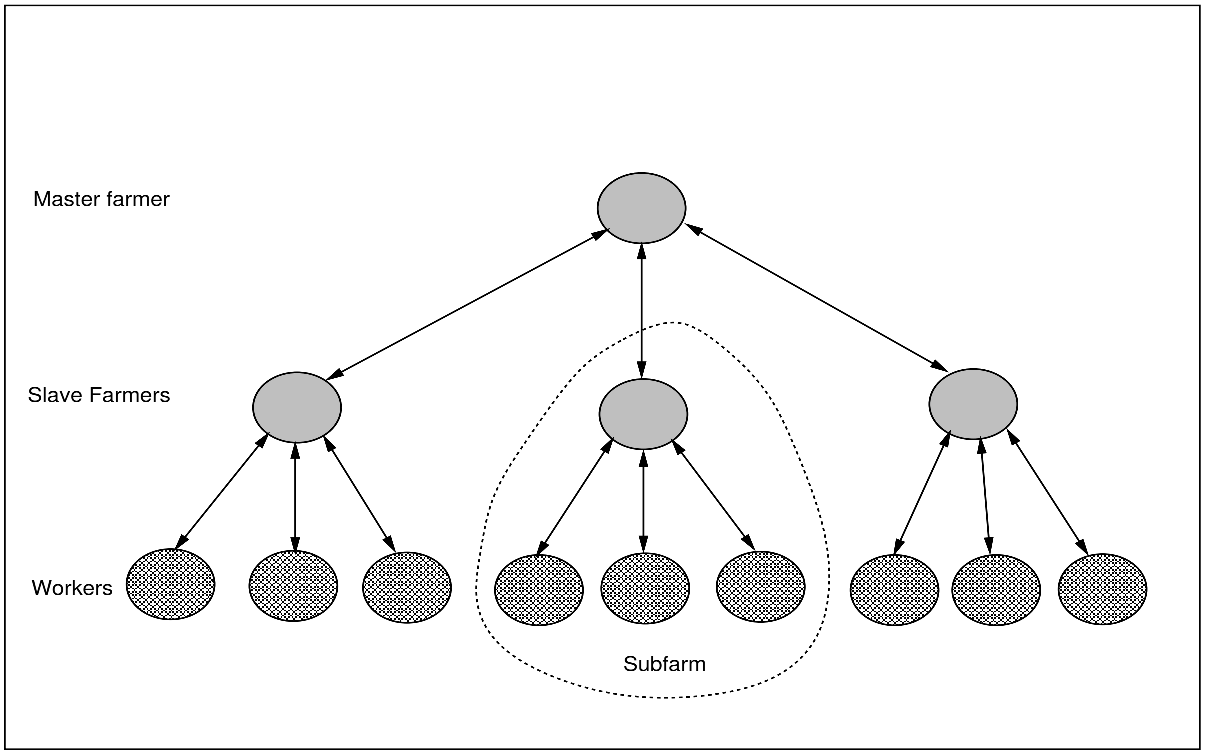


Figure 3.4: Hierarchical structure of a processor farm

same functional model of process cooperation so the code comparison illustrates compactness and expressiveness of the mechanisms provided by various tools. It is important to point out that various software platforms allow a different level of the internal processor fram parallelism.

The LAN communication system is rather slow with comparison to a shared memory or a bus data transfer and introduces a delay measured in milliseconds. To improve the communication efficiency, a double buffering mechanism of tasks allocation for the worker processor should be implemented. It makes communication delays less critical by an elimination of the idle period of time of worker processes. It is a time between reporting of the results to the farmer that is also a request for a new task, and its reception.

The implementation of the double buffering needs an asynchronous send and receive operation and a multithreading environment. Unfortunately, these features are only provided by a little more advance software for the distributed processing such as ANSA, SR, Strand and Linda. For any software environment this mechanism of data buffering may be simulated via allocation more than one worker process to every workstation. In such a situation the probability that two processes finish a computation at the same moment of time, send result back to the farmer and, next, stay idle waiting for new data is rather low. This method has been used for PVM and P4.

There are many ways to measure the performance of a parallel algorithm running on a multiprocessor system [3, 5, 6]. The most commonly used measurements are elapsed time, speed-up and efficiency. Although the definitions of these performance indexes are well known we remind them once again to avoid misunderstanding.

- *elapsed time* - The time to run a particular job.

- *speed-up* - More precisely the absolute speed-up has been used defined as the elapsed time

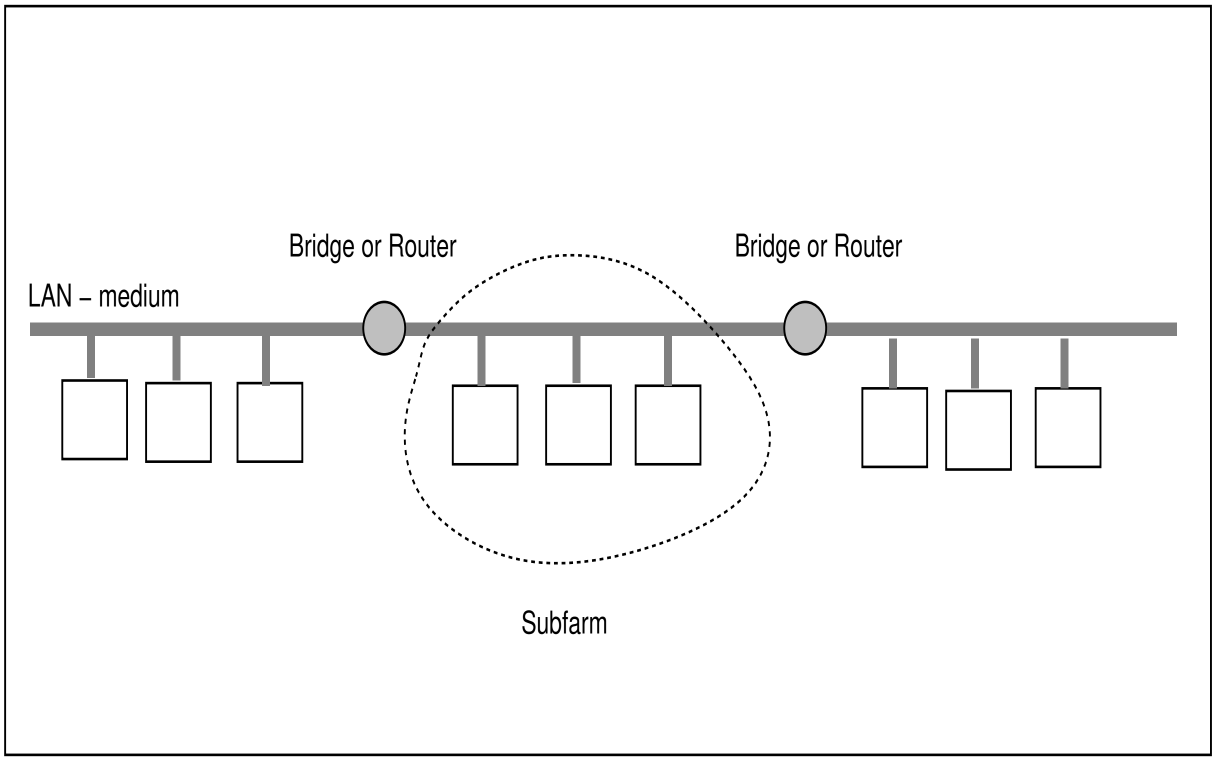


Figure 3.5: Improvement of communication subsystem efficiency by bridges

needed by 1 processor divided by the time needed by p processors:

$$s = \frac{T(1)}{T(p)}$$

The T(1) is called a sequential time of the parallel algorithm. This definition is a good measurement of parallel processing gain versus sequential processing [5].

- *efficiency* - The speed-up per processor:

$$e = \frac{T(1)}{pT(p)} = \frac{s}{p}$$

The following experiments have been performed:

- The time of a sequential algorithm computation on various workstations has been measured.
- Influence of tasks granularity on the computation performance has been evaluated. In experiment only one worker process per workstation was allocated. Measurements were repeated for various numbers of workstations.
- For the optimal granularity the relation between a number of stations and the processing time was identified. Based on these experiments a speed-up and an efficiency were calculated.

## 3.3 Case Study Problems Characteristics

### 3.3.1 Fractal

A generation of a fractal picture is one of the most common computational intensive task examples, easy to parallelize because of a geometrical parallelism. This task is very often used as a test of the processor farm software. It is characterized by the :

- easy change of the task weight by the size of the scene setting,
- task granularity change by the number of the iteration definition,
- various time of computation per pixel.

The task definition is so simple that a distributed program code almost contains only the basic framework of the processor farm software. It is very easy to separate the fractal dependent code and to use the framework for other applications.

### 3.3.2 Ray Tracing

#### Ray Tracing as a Computational Intensive Task

The fact that ray tracing is a very intensive computational problem is well-known and documented elsewhere [7, 8]. However to justify an effort for parallelization it seems to be very interesting to compare a relative performance of commonly available computers to a running of the typical ray tracing program.

For the experiment the DBW-Render package by Wiliam T.Baldridge has been used. In all, the DBW-Render package represents about 4000 lines of C code. It is largely based on original algorithms for the VAX by Tom Dahl and Tone Engel at Digital Equipment Corporation. That package has been chosen mostly because its C source version is available free what is sound for further work on its parallelization.

The DBW-Render package uses an algorithm called Distributed Ray Tracing. In addition to neatly solving the hidden surface problem, ray tracing (in general) affords the opportunity for doing excellent (that is, highly realistic) shading and illumination calculations.

The RAY program knows how to create images composed of four primitive geometric objects: spheres, parallelograms, triangles, and flat circular rings (disks with holes in them). Some of the features of the program are:

- Determination of correct shadows cast by arbitrarily shaped objects, onto arbitrarily shaped objects, due to arbitrarily placed light sources.
- Diffuse and specular reflections (with arbitrary levels of gloss or polish). Rudimentary modeling of object-to-object diffusely reflected light is also implemented, that among other things accurately simulates color bleed effects from adjacent contrasting colored objects.
- Mirror reflections, including varying levels of mirror smoothness or perfection.
- Retraction and translucency (which is akin to variable microscopic smoothness, like the surface of fetched glass).
- Two types of light sources: purely directional (parallel rays from infinity) of constant intensity, and spherical sources where intensity is determined by the inverse square law.

- Photographic depth-of-field. That is, the virtual camera may be focused on a particular object in the scene, and the virtual camera's aperture can be manipulated to affect the sharpness of foreground and background objects.

- Solid texturing. Normally, a particular object is considered to have constant properties over the entire surface of the object, often resulting in fake looking objects. Solid texturing is a way to algorithmically change the surface properties of an object to try and model some real world material. Currently the program has built in rules for modeling wood, marble, bricks, snow covered scenes, water (with arbitrary wave sources), plus more abstract things like color blend functions.

- Fractals. The program implements what's known as recursive triangle subdivision, which creates all manners of natural looking surface shapes.

The precision of an image generation, in terms of reality, influences the computation time and may be set on a certain level using the following parameters:

1. The maximum computation time may be set in the range 1 - 30 hours. When the given period of time is not long enough, then the colors of some subsets of pixels are calculated using approximation.

2. A number of rays passing through each pixel on a screen. A light intensity and a color is calculated for each ray. The final value for a pixel is an average of calculated values.

3. A threshold value of the variation calculated for each value of the ray light intensity of the given pixel. If the obtained value of the variation is greater than the chosen value of the threshold then the computation for that pixel is repeated for the doubled value of the rays number.

The DBW-Render is a professional software package. Its modularity and clarity helps during implementation of a parallel version of this program.

The experiment under study in that section concerns not a very complicated scene tested in Raport 1 and in [7] so the direct comparison is possible. It is evident that despite of architecture, the problem of a realistic image generation even for a very simple case takes enough time to justify an effort of its parallelization.

### Parallel model of the DBW-Render

The ray tracing computational task is characterized by data-oriented parallelism [2]. It means that the simultaneous performance of an operation on multiple data elements e.g. pixels is possible. Whenever the size of the data warrants parallelization, activities are replicated for application across data subsets. In effect, the program execution is viewed as a single thread of control which temporarily diverges into the parallel action sequence that converges later on. The methods used to distribute data and synchronize convergence become critical factors.

The described in previous section model has been used for the parallelization of the DBW-Render. The original structure of the sequential code is shown in Fig. 3.6. The body of the most inner loop contains the function **do_raytrace** that calculates of a pixel color.

The worker process contains this function extended by the communication harness. The communication harness complexity depends on a hardware connections topology of processors.

```
read_image_file ;
{ It reads a sort of command file, that
contains a geometric description of the
scene to be rendered, plus desired global
lighting conditions, camera parameters, etc. }
   for  i := 1 to SCREEN_WIDTH do
      for  j := 1 to SCREEN_HEIGTH do
         do_raytrace(i,j);
         { This functions calculates the coulor of
         each pixel on a screen separately }
   write_image_file ;
   { Results of computations are wirittem to
   disk for further processing if necessary}
```

Figure 3.6: The structure of the DBW-Render code

In general the choice of a topology should be a solution of trade-off the between potential parallelism of the wiring connection and complexity of the routing algorithm. For LAN distributed computing the full connected graph seems to be a right model of a communication subsystem.

# Chapter 4

# Experimental Results

## 4.1 Hardware and Software Configuration

The experiments reported in this chapter concern the workstation types running under the operating system shown in Table 4.1. The software package versions under study are summarized in Table 4.2. The workstations were connected through 10Mbps Ethernet LAN.

| Workstation Type | Operating System Version |
|---|---|
| SUN SLC | SUN OS 4.1.3 |
| SPARC Station 2 | SUN OS 4.1.1 |
| RS 6000-320 | AIX 3.1 |

Table 4.1: Workstations specification

## 4.2 Communication evaluation results

The code of the test programs used for the communication efficiency testing are shown in Appendix B. For Strand and Linda, that do not provide typical of message passing operations, their equivalents have been created. The measurements have been carried out when a system was quite and no additional traffic was quiet present in the medium. The domains of parameters taken into account are summarized in Table 4.3.

| Software Package | Version |
|---|---|
| PVM | 2.4.0 |
| P4 | 1.2 |
| ANSA | 4.0 |
| SR | 2.0 |
| Linda | 2.4.6 |
| Strand 88 | Buckingham Release June 1990 |

Table 4.2: Software under experiment versions

| Parameter | Value Domain |
|---|---|
| Worstation Type | SUN SLC, SPARC Station 2, RS 6000-320 |
| Software Package | PVM, ANSA, P4, Linda, Strand, SR |
| Message Length | from 16Byte up to 1024Byte with step 16Byte and repetition counter 10 000<br>from 1KByte up to 64KByte with step 1KByte and repetition counter 1000 |

Table 4.3: Domain of experiment parameters

The results of communication efficiency testing were presented in Fig. 4.1 to Fig. 4.12 and in the Appendix C in the tabular form. For every software package under study the diagrams

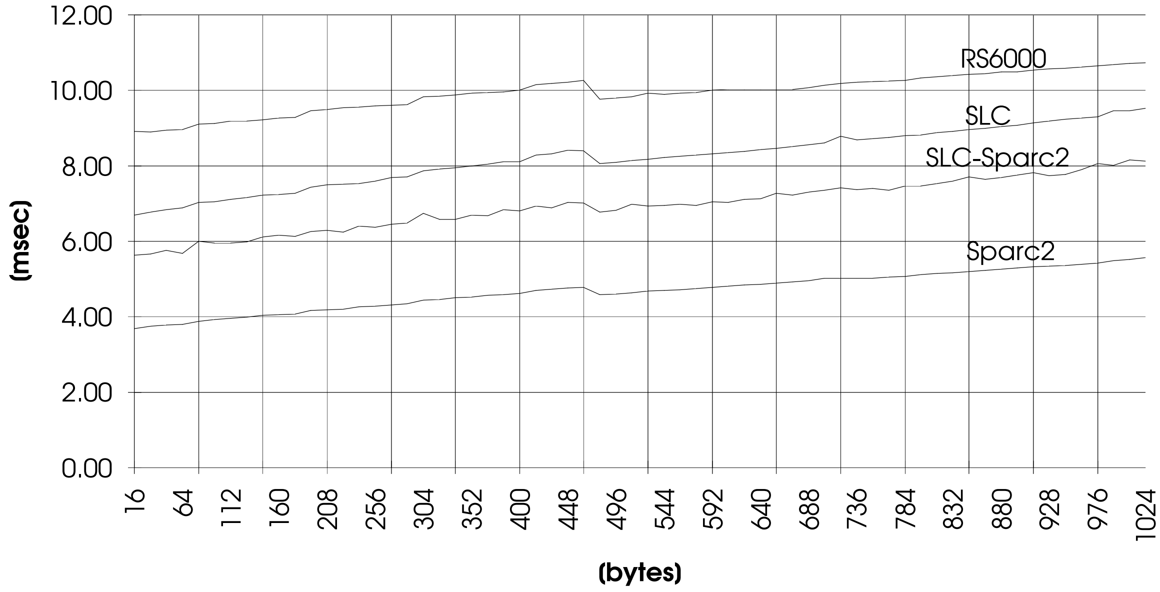


Figure 4.1: PVM transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

of data transmission time vs. data block length have been presented in seperate figures that

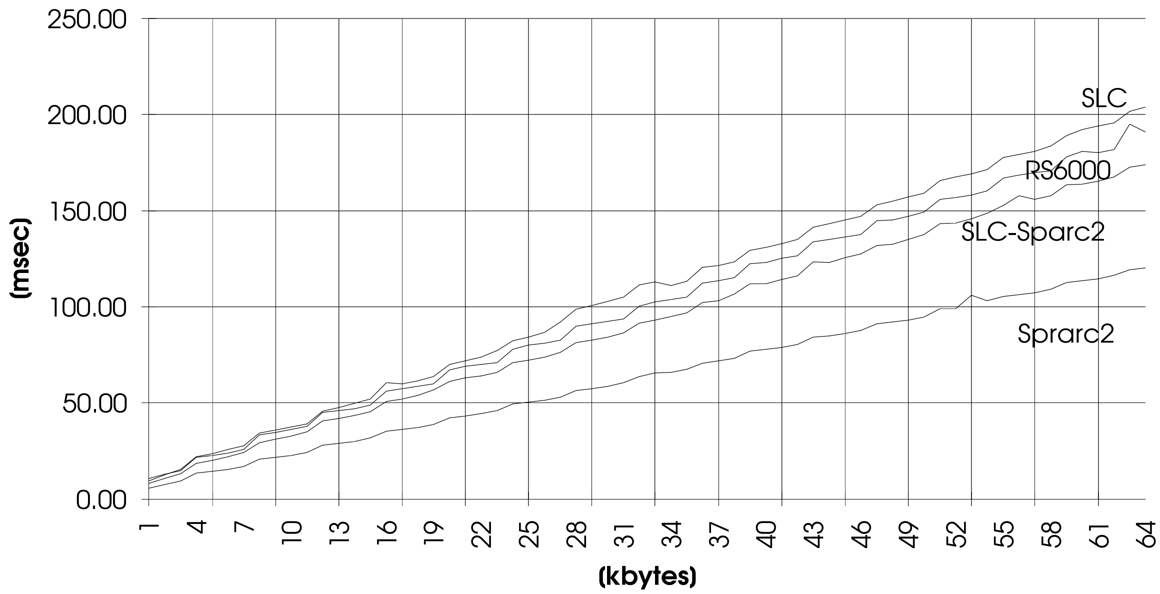


Figure 4.2: PVM transfer time of large data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

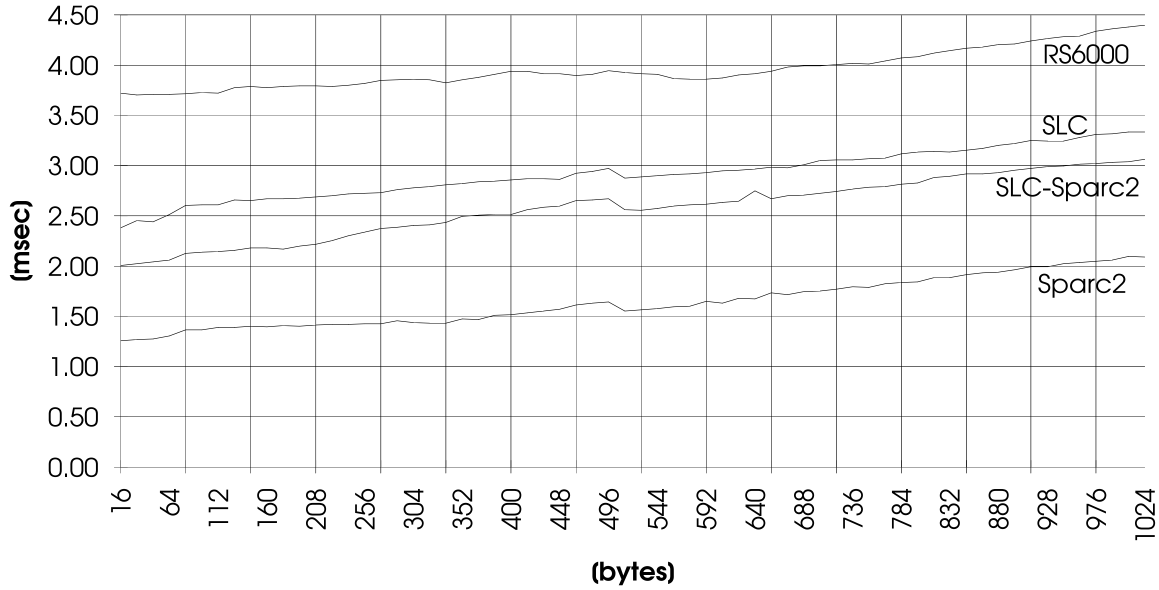


Figure 4.3: P4 transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

correspond to the small and large block ranges. The diagrams were parameterized by the

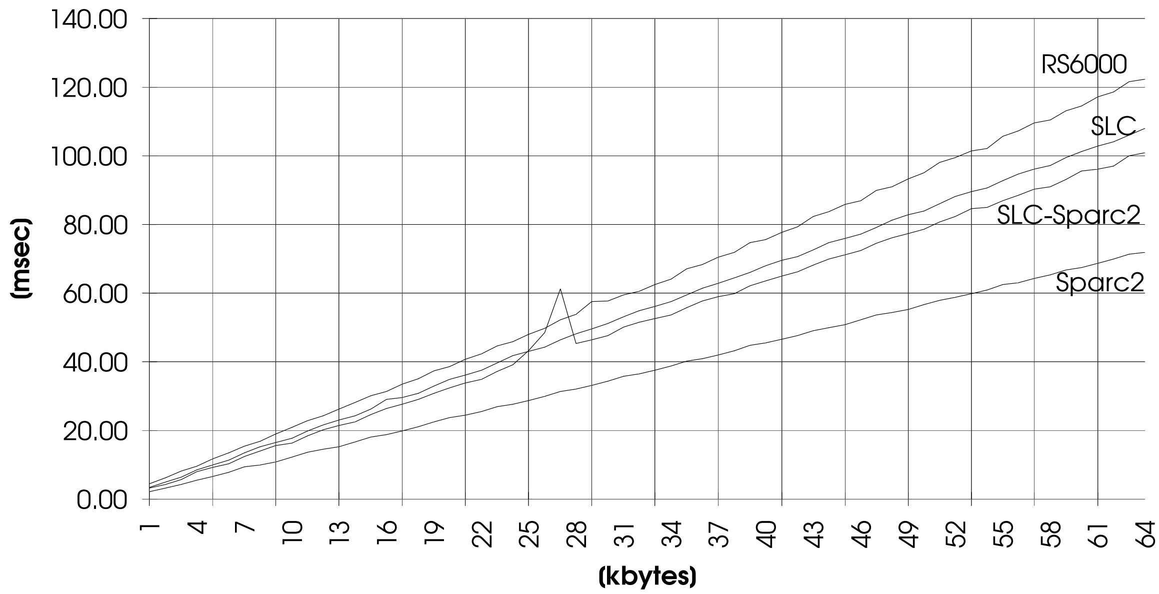


Figure 4.4: P4 transfer time of large data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

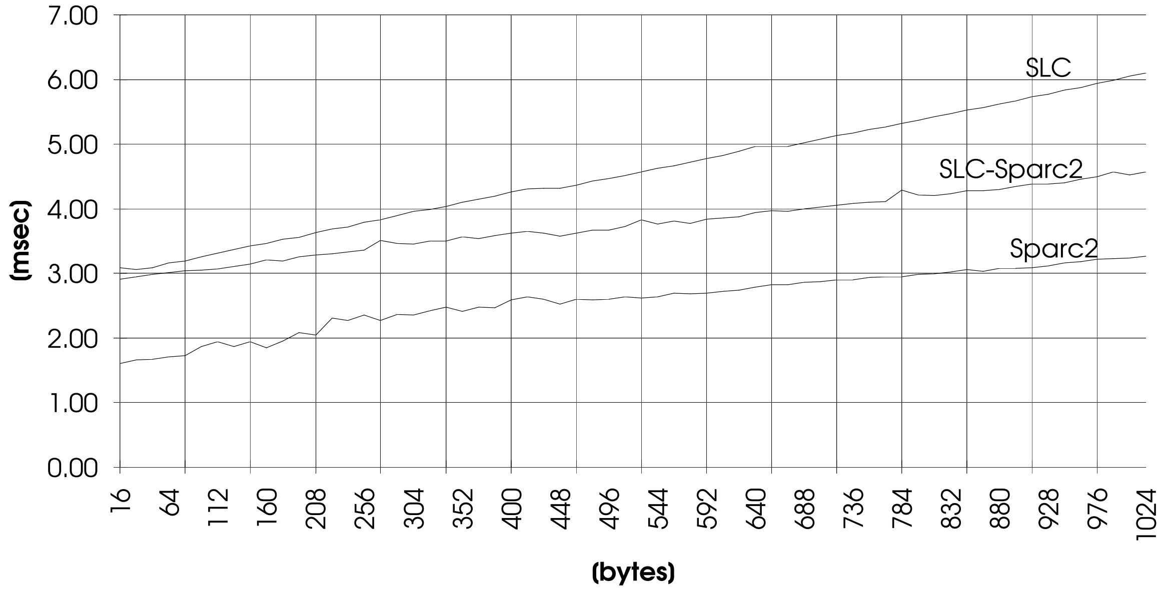


Figure 4.5: ANSA transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

workstation type.

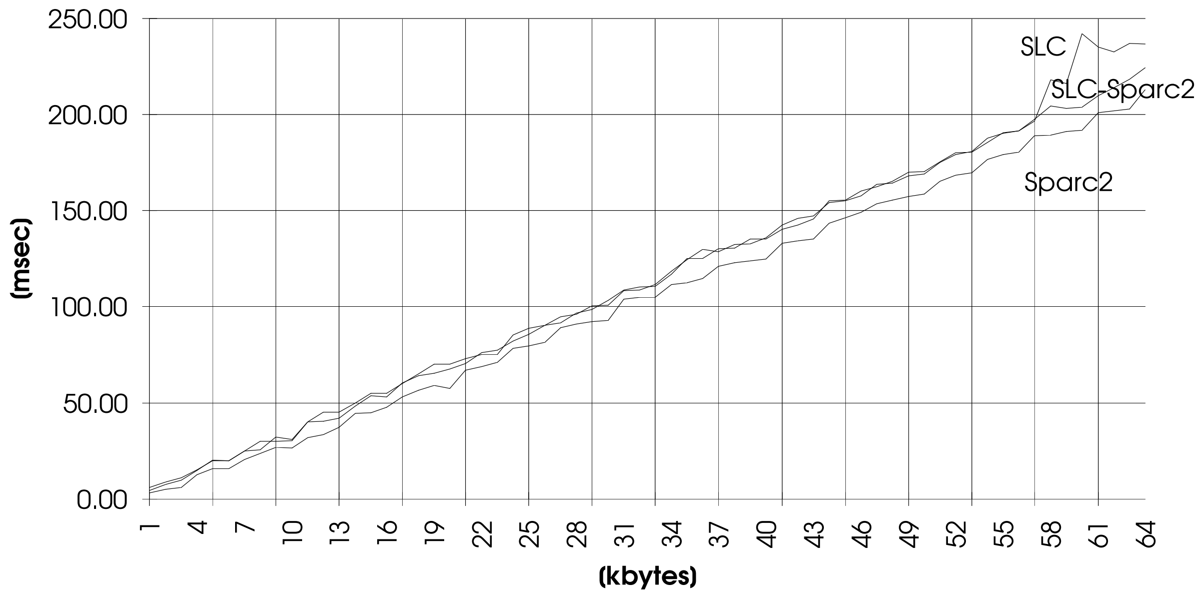


Figure 4.6: ANSA transfer time of large data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

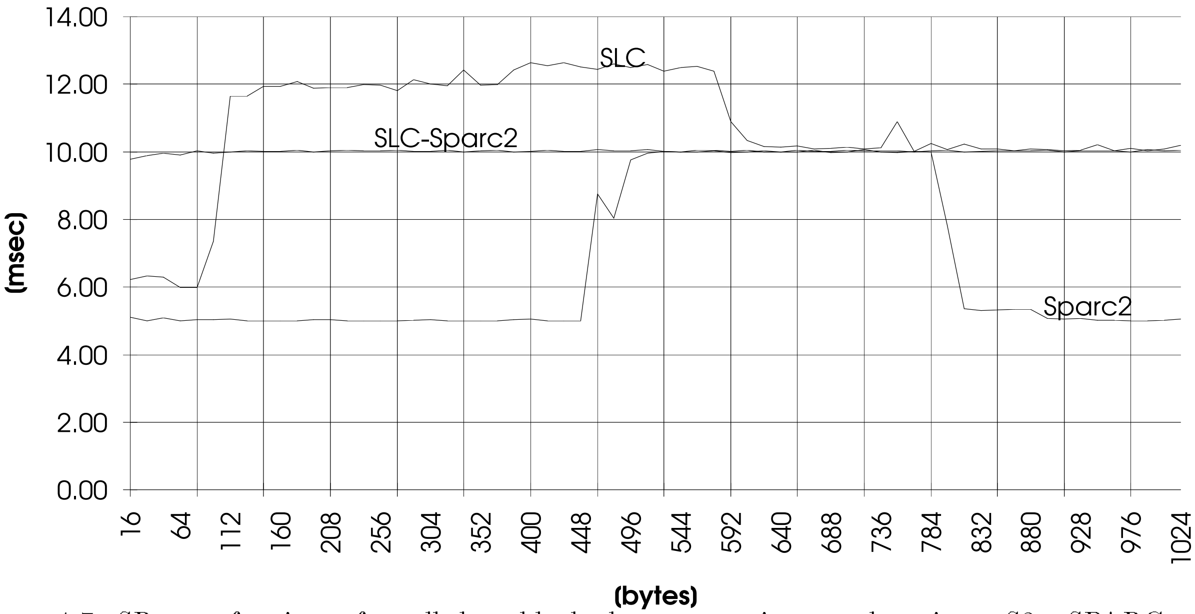


Figure 4.7: SR transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

The comparision of the results leads to the following conclutions:

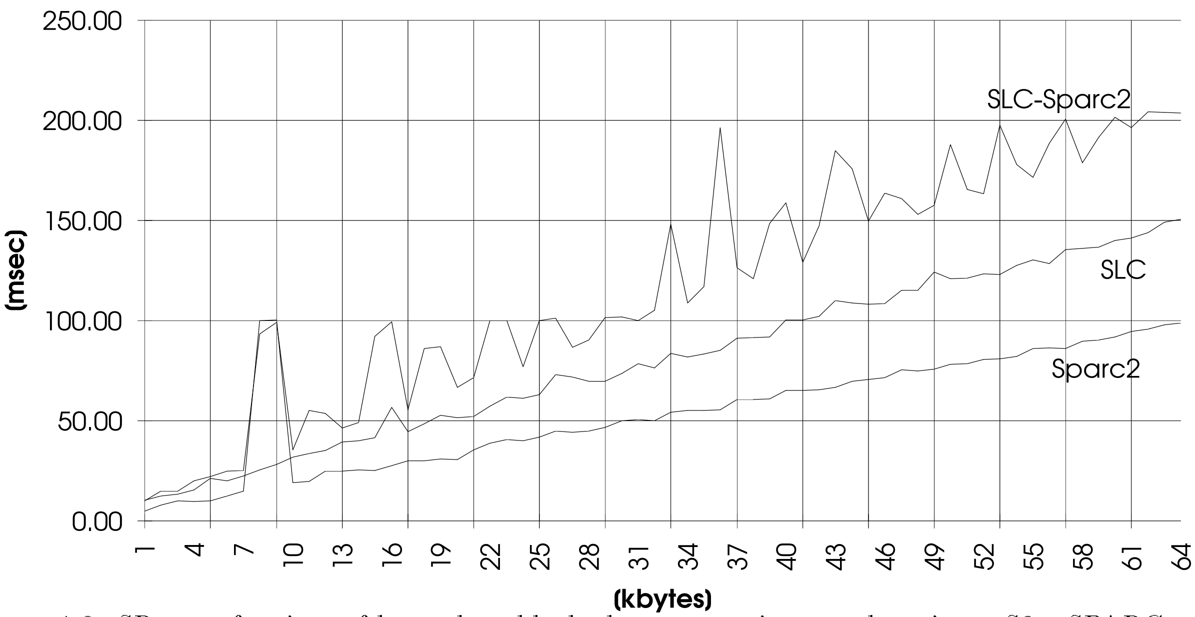


Figure 4.8: SR transfer time of large data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

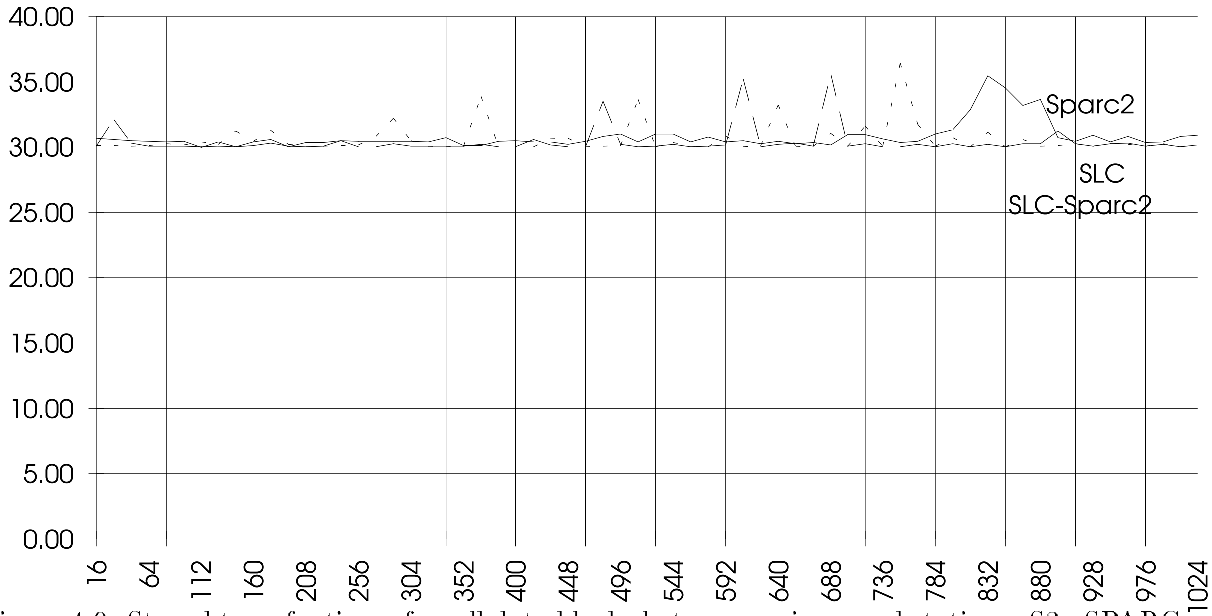


Figure 4.9: Strand transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

- The communication efficiency over the 10 Mbps Ethernet at the investigated software level

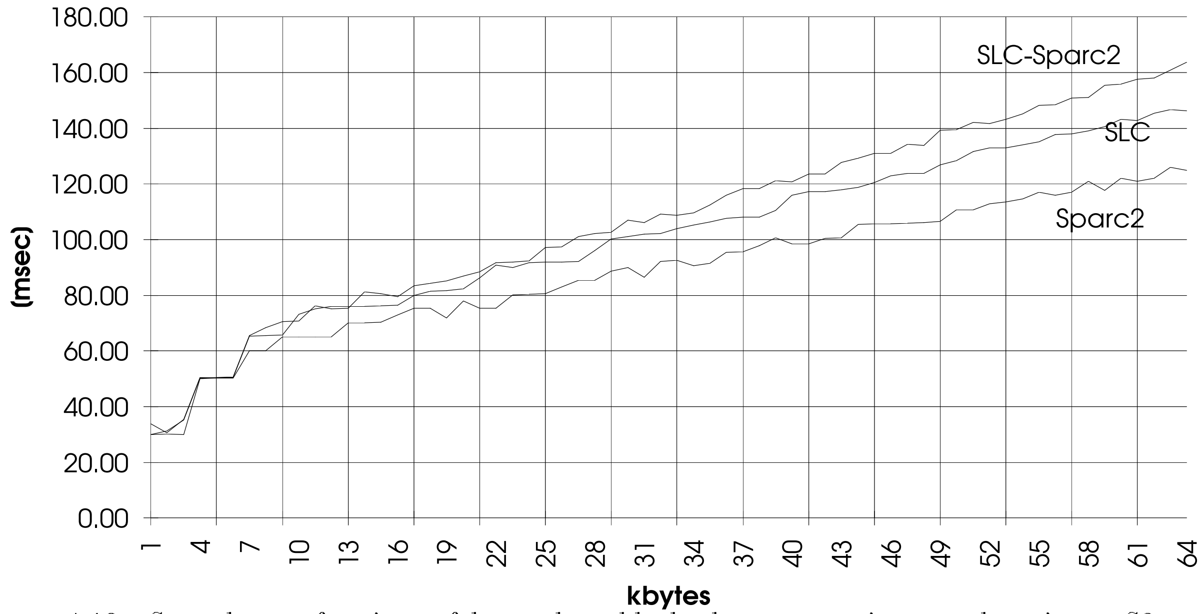


Figure 4.10: Strand transfer time of large data blocks between various workstations: S2 - SPARCstation2, SLC - SUN Sparc SLC.

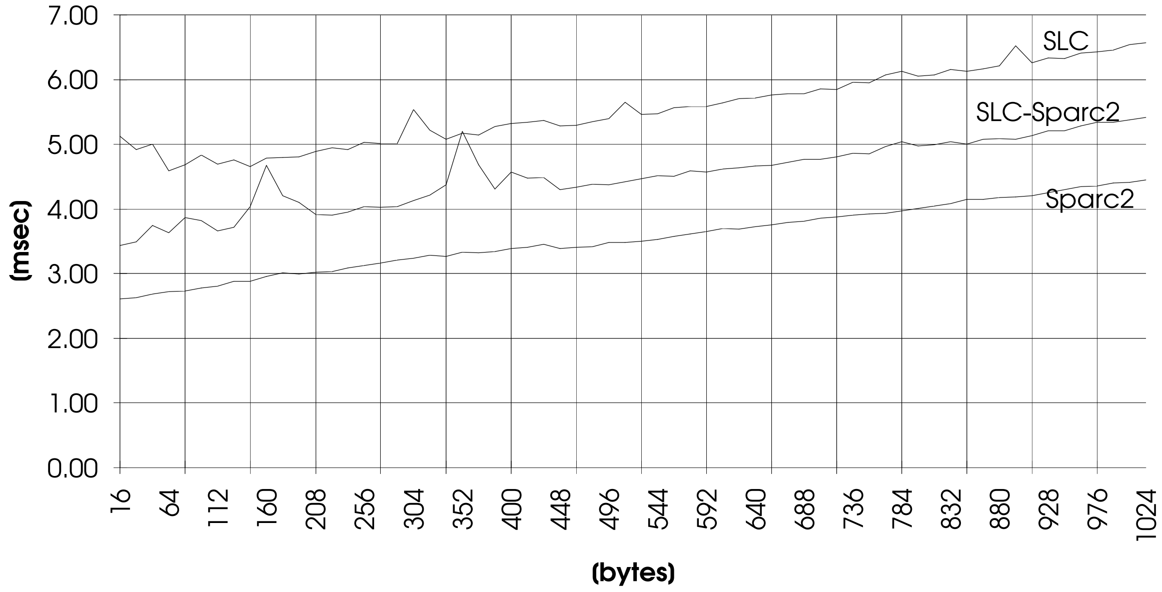


Figure 4.11: Linda transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

depends on the workstation speed. The sensitivity depends on the ratio of the transmission

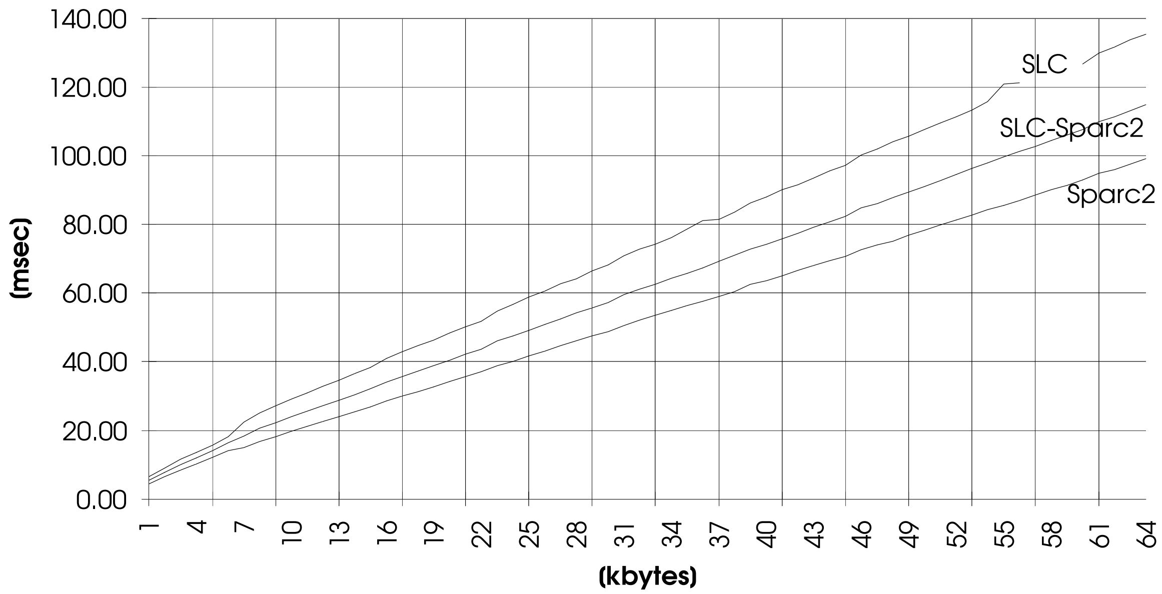


Figure 4.12: Linda transfer time of large data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

time to the processing time necessary for its initiation. The time of the transmission initiation includes internal data coping, and communication handler operations and is linearly reduced with the workstation processor speed. This conclusion is true only then when machines under the same operating system are compared.

- The communication efficiency between two SPARCstation2 was approximately 10 % better than between the SUN SLC and SPARCstation 2, and 20 % better than between two SUN SLC.

- The most surprising results concern RS6000. It is the most powerful machine under tests. Unfortunately, the communication efficiency obtained for the PVM and P4 software was very disappointing. The system AIX and communication software seem to be very inefficient with the data transfer over Ethernet. It course the communication bottleneck in the claster of RS6000 machines built around Ethernet. The similar results not reported here confirmed a similar bottlneck in communication via token ring.

- The packages PVM, P4, ANSA, and Linda have almost a linear characteristic of the transfer time vs. data length.

- SR and Strand have some anomalies of the data transfer efficiency.

  The Strand data transfer time is approximately independent on the data block length up to 3 Kbyte and is equal 30 miliseconds. For large data block the Strand efficiency increases and the investigated characteristic is almost linear.

  The SR has also anomalies of the data transfer efficiency in the small data length range. These anomalies are also observed for the large blocks for the mixed SUN SLC SPARCstation 2 configuration. The repoted results are an average of results obtained during many repetitions of the expriment so the discussed anomalies are not an random effect.

The comparison of the investigated package efficiency has been given in Table 4.4 for the data length 16Byte, 1KByte, and 64KByte

| Software | **16 Byte** | | | | **1 KByte** | | | | **64 KByte** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Package | S2 | S2-SLC | SLC | RS | S2 | S2-SLC | SLC | RS | S2 | S2-SLC | SLC | RS |
| Strand | 30.04 | 30.08 | 30.12 | | 30.03 | 30.16 | 30.23 | | 124.88 | 163.60 | 146.20 | |
| PVM | 3.68 | 5.63 | 6.69 | 8.91 | 5.55 | 8.25 | 9.51 | 10.74 | 120.24 | 173.86 | 203.99 | 190.84 |
| SR | 5.12 | 9.78 | 6.23 | | 5.06 | 10.05 | 10.20 | | 98.86 | 203.58 | 150.73 | |
| Linda | 2.61 | 3.43 | 5.12 | | 4.45 | 5.42 | 6.57 | | 99.09 | 114.89 | 135.41 | |
| ANSA | 1.61 | 2.91 | 3.09 | | 3.27 | 4.58 | 6.13 | | 213.09 | 224.27 | 236.54 | |
| Express | 1.11 | 1.52 | 2.13 | | 2.45 | 3.02 | 3.92 | | | | | |
| P4 | 1.26 | 2.01 | 2.38 | 3.72 | 2.17 | 3.12 | 3.34 | 4.30 | 71.90 | 100.80 | 107.94 | 122.22 |

Table 4.4: Summary of communication efficiency study

The results shown in this table have been depicted in Fig. 4.13, 4.14, and 4.15 respectively. The comparison vs. software tools lead to the following conclusions:

- P4 has the most efficient communication operations in the whole range of data block length under study.

- For small data blocks ANSA has also a very good efficiency. The ordering according to decreasing efficiency is as follows: P4, Express, ANSA, Linda, SR, PVM, and Strand. It is illustrated more precisely in Fig. 4.16 for the SPARCstation 2 to SPARCstation 2 communication.
- Strand efficiency is very bad for short messages.
- For large data blocks, Strand is doing even better than ANSA. The ordering according to decreasing efficiency is as follows: P4, Linda, Strand, SR, PVM ANSA. It is illustrated more precisely in Fig. 4.16 for the SPARCstation 2 to SPARCstation 2 communication. Unfortunately, this order is dependent on hardware platforms.

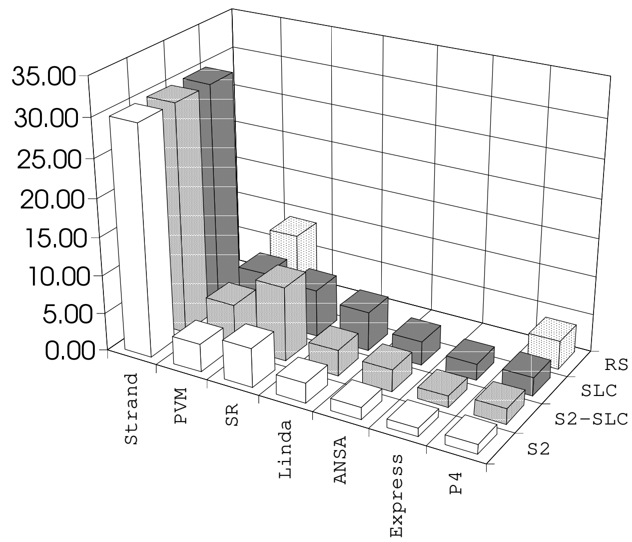


Figure 4.13: Transfer time of 16 Byte data block by various software packages

## 4.3 Fractal computation results

In the reported experiments the parameters shown in Table 4.5 has been used.

| Parameter | Value |
|---|---|
| Size of scene | 800 x 600 pixels |
| Max no. of iteration | 256 |
| Load of the network | There was no foreign traffic on the LAN |
| No. repetition | 5 |

Table 4.5: Fractal experiment conditions

The time of the computation has been measured from the moment of time when the first task has been sent to the worker process until the last result has been collected. So the time of the given distributed environment set-up has not been measured. The initialization time of the

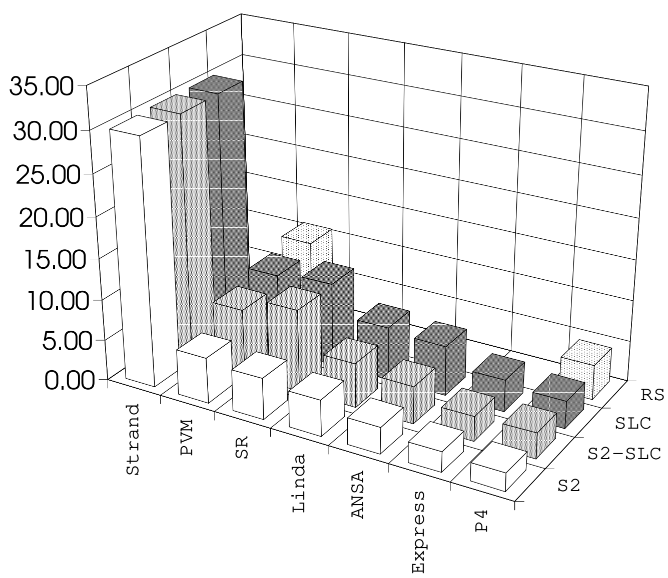


Figure 4.14: Transfer time of 1 KByte data block by various software packages

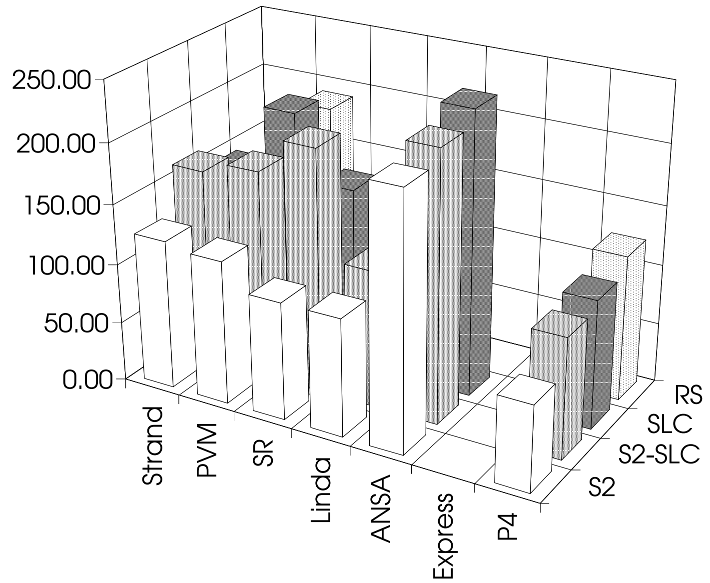


Figure 4.15: Transfer time of 64 KByte data block by various software packages

studied packages is very different and it

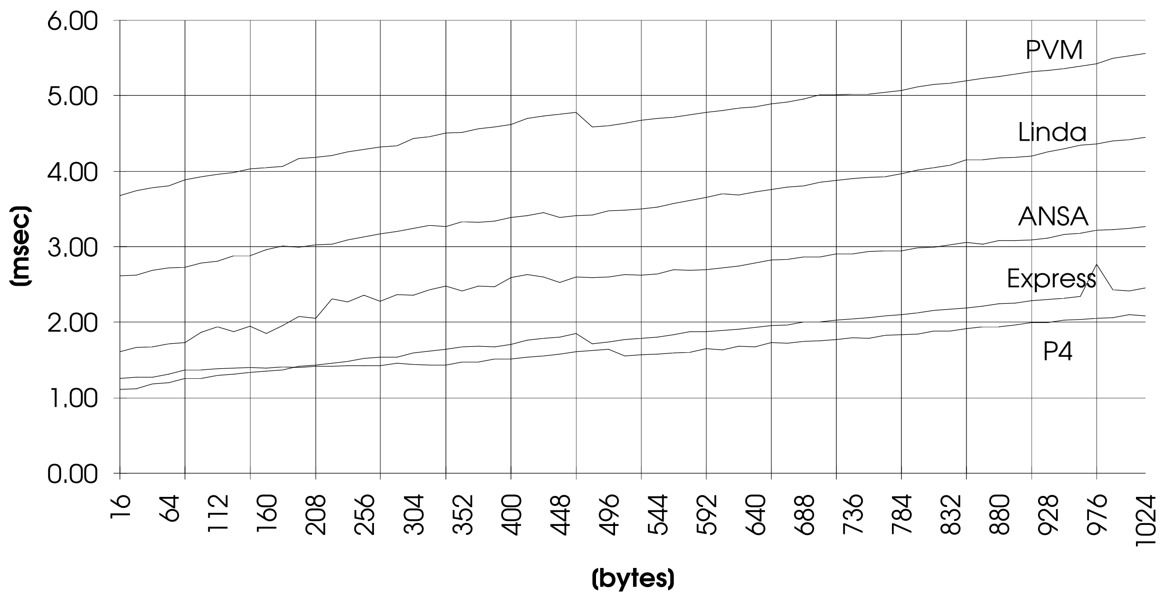


Figure 4.16: Data tranfer time between two SPARCstation2 using different software tools versus small data block lengt

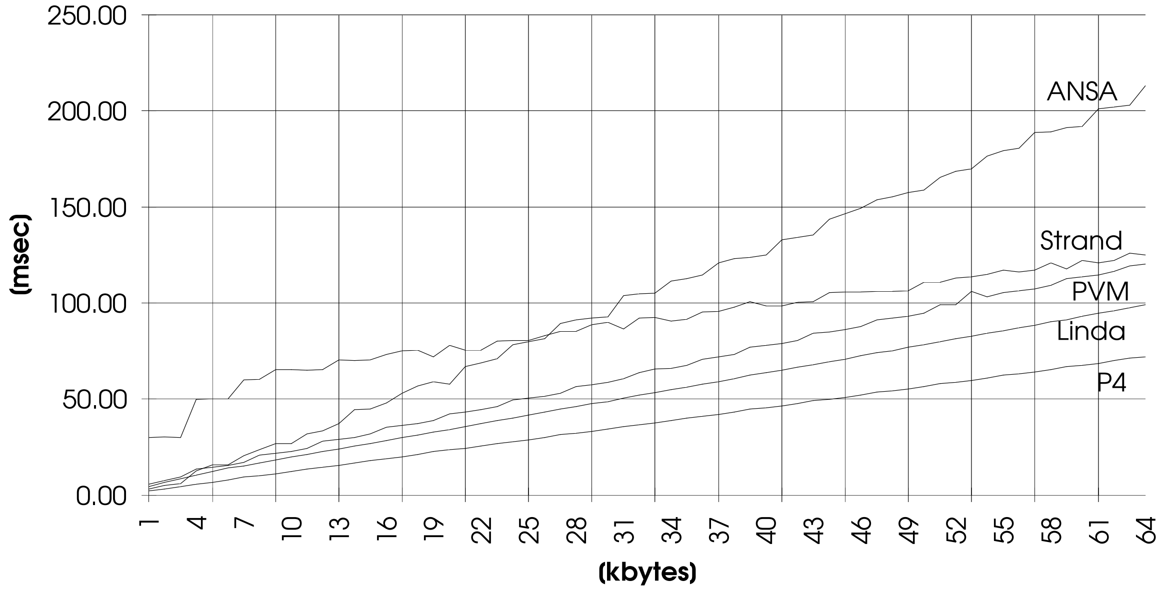


Figure 4.17: Data tranfer time between two SPARCstation2 using different software tools versus large data block length

is in fact a separate parameter that should be taken into account during their evaluation. This time is rather substantial so it may mask the performance of the computational phase.

To show that a fractal computational time is substantial and to compare a relative performance of the available processors this task was computed sequentially on various architectures. The obtained results have been summarized in Table 4.6 and in Fig. 4.18. These results are especially important then the trade-off between communication and processor power is investigated. It is also necessary to point out that that for the task under study modern RISC processors are approximately 10 times faster than transputer T800.

| No. of Architecture | Processor Type | Compiler | Time [sec] | Relative performance |
|---|---|---|---|---|
| 1 | T800/20 | 3LC v2.1 | 225.92 | 1.00 |
| 2 | SUN SLC | UNIX C | 174.00 | 1.30 |
| 3 | Intel 486/25 | UNIX C | 112.50 | 2.01 |
| 4 | SUN IPC | UNIX C | 95.00 | 2.38 |
| 5 | SPARC 2 | UNIX C | 44.00 | 5.13 |
| 6 | I860/40 | GNU | 21.70 | 10.41 |
| 7 | RS-6000-320 | XLC | 20.80 | 10.86 |
| 8 | HP 9000/720 | UNIX C | 17.21 | 13.13 |

Table 4.6: Fractal computation sequential time on different processors

The influence of task granularity on the computational time has been shown in Table 4.7 Fig. 4.19. This experiment has been repeated for various software tools and hardware platforms.

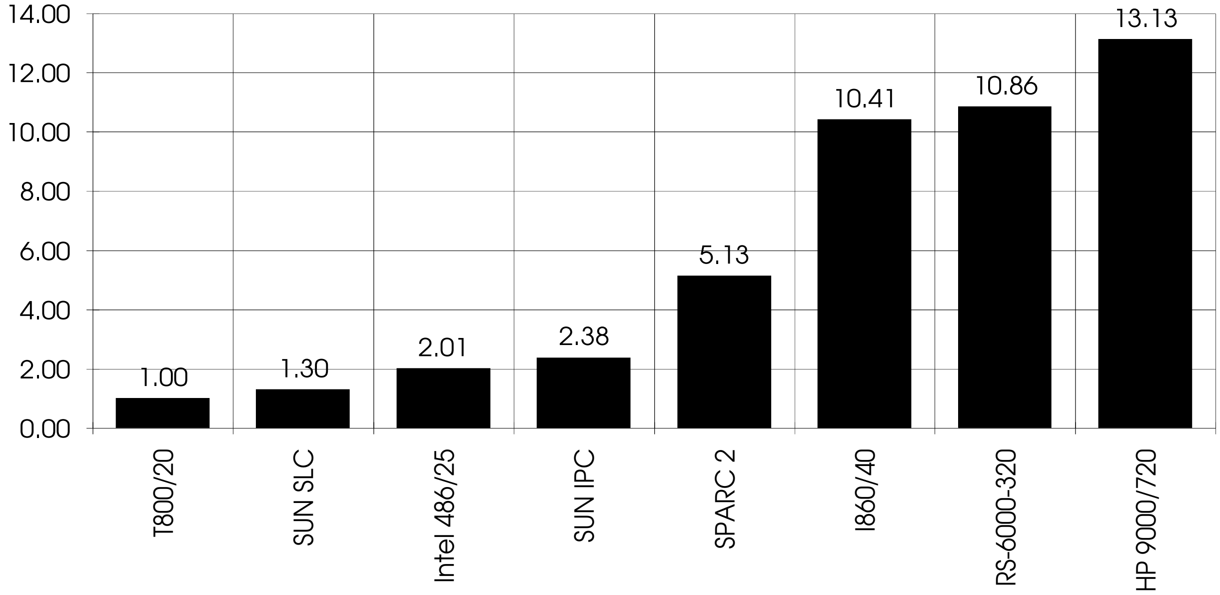


Figure 4.18: Fractal computation sequential time relative performance

The obtained results leads to conclusions that:

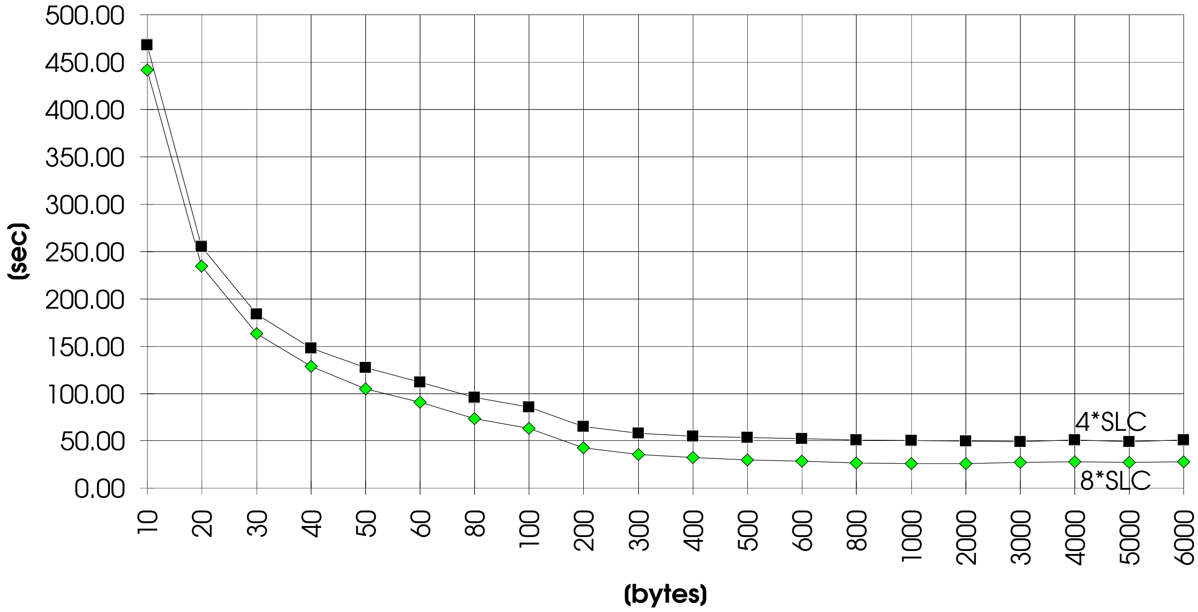


Figure 4.19: The influence of the task granularity on fractal computation efficiency on 4 and 8 SLC respectively using PVM

| | Time [sec] | |
|---|---|---|
| Pixel per task | 4 x SLC | 8 x SLC |
| 10 | 468.08 | 441.76 |
| 20 | 254.79 | 234.54 |
| 30 | 183.93 | 163.16 |
| 40 | 147.74 | 128.37 |
| 50 | 127.40 | 104.70 |
| 60 | 111.68 | 90.85 |
| 80 | 96.07 | 73.37 |
| 100 | 85.58 | 63.21 |
| 200 | 65.35 | 42.48 |
| 300 | 57.86 | 35.91 |
| 400 | 54.79 | 32.35 |
| 500 | 53.32 | 29.86 |
| 600 | 52.10 | 28.30 |
| 800 | 51.13 | 26.73 |
| 1000 | 50.40 | 25.72 |
| 2000 | 49.62 | 26.09 |
| 3000 | 49.45 | 27.25 |
| 4000 | 51.15 | 27.57 |
| 5000 | 49.32 | 27.43 |
| 6000 | 50.73 | 28.07 |

Table 4.7: The influence of the task granularity on fractal computation efficiency on 4 and 8 SLC respectively using PVM

- The small task granularity degrades the performance substantially.
- The optimal granularity for the investigated fractal lead in the range 800 to 2000 pixel per task and is independent from hardware and software platform. For the further study the 800 pixel per task granularity has been chosen.

The influence of task granularity on the computational time in Strand and PVM are shown in Table 4.8 and in Fig. 4.20. The communication in Strand as it was shown in the previous section is very inefficient for small messages. The PVM is doing a lot better in this range. The Strand processor farm is internally written in much more parallel way than the PVM farm and automatically exploits such a mechanism as double buffering. The results is very surprising. Strand is doing better than PVM for small task granularity. This is also very good example how sophisticated problem represents the comparison study of different software tools.

The comparision of these two software tools in term of speed-up and efficiency is shown in Table 4.9 and in Fig. 4.21 and 4.22, respectively. These computations were performed for the task granularity 800 pixel per task.

The double buffering scheme may be very easy implemented using SR. The influence of this mechanism on the system performance is in Table 4.11 and in Fig. 4.23. This experiment has been performed for the SPARCstation 2 as a farmer node and SLC as workers. Similar results have been presented for Strand in Table 4.9.

The double buffering improves efficiency for a greater number of nodes. It is justified by the fact that the double buffering helps to overcome a communication bottleneck.

So far the model A of the processor farm was investigated only. From the communication performance study a very low efficiency of the RS6000 system it is evident . How it influences

| Pixels per task | Time [sec] Strand | PVM |
|---|---|---|
| 10 | 285.25 | 443.53 |
| 20 | 175.18 | 236.35 |
| 30 | 139.81 | 165.79 |
| 40 | 118.93 | 129.63 |
| 50 | 108.18 | 108.67 |
| 60 | 99.37 | 92.81 |
| 70 | 91.13 | 83.47 |
| 80 | 86.71 | 76.30 |
| 90 | 84.06 | 71.46 |
| 100 | 80.79 | 66.66 |
| 200 | 67.48 | 46.38 |
| 300 | 54.20 | 39.51 |
| 400 | 49.62 | 35.97 |
| 500 | 44.74 | 33.84 |
| 600 | 44.44 | 31.32 |
| 700 | 41.79 | 32.30 |
| 800 | 46.49 | 30.61 |
| 900 | 42.96 | 30.27 |
| 1000 | 41.18 | 29.94 |
| 1100 | 40.71 | 30.08 |
| 1200 | 44.03 | 29.90 |
| 1300 | 38.19 | 30.25 |
| 1400 | 43.08 | 29.63 |
| 1500 | 44.03 | 29.94 |
| 1600 | 41.10 | 29.66 |
| 1700 | 45.02 | 30.09 |
| 1800 | 43.02 | 29.70 |
| 1900 | 42.37 | 29.66 |
| 2000 | 40.01 | 30.28 |

Table 4.8: The influence of the task granularity on fractal computation time on 8 SLC using Strand and PVM respectively

| No. of nodes | Strand Time | Strand Speedup | Strand Efficency | PVM Time | PVM Speedup | PVM Efficency |
|---|---|---|---|---|---|---|
| 1 | 174.00 | 1.00 | 1.00 | 174.00 | 1.00 | 1.00 |
| 2 | 111.57 | 1.56 | 0.78 | 93.74 | 1.86 | 0.93 |
| 3 | 77.12 | 2.26 | 0.75 | 63.23 | 2.75 | 0.92 |
| 4 | 66.30 | 2.62 | 0.66 | 48.44 | 3.59 | 0.90 |
| 5 | 53.25 | 3.27 | 0.65 | 39.33 | 4.42 | 0.88 |
| 6 | 47.15 | 3.69 | 0.62 | 33.32 | 5.22 | 0.87 |
| 7 | 43.89 | 3.96 | 0.57 | 29.42 | 5.91 | 0.84 |
| 8 | 37.56 | 4.63 | 0.58 | 26.96 | 6.45 | 0.81 |

Table 4.9: The comparison of Strand and PVM

the speedup and efficiency of the fractal computation is shown in Table 4.10. for the PVM platform. According to our discussion in previous chapter there are some possibilities to improve a processor farm computation when the model B of farm instead A is used for greater number of

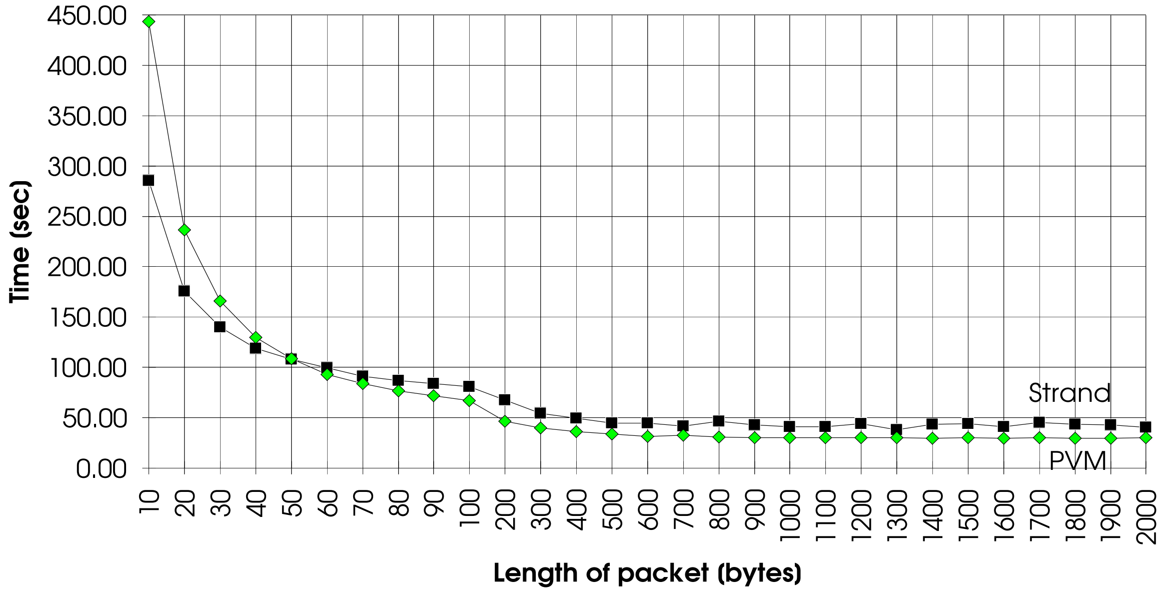


Figure 4.20: The influence of the task granularity on fractal computation time on 8 SLC using Strand and PVM respectively

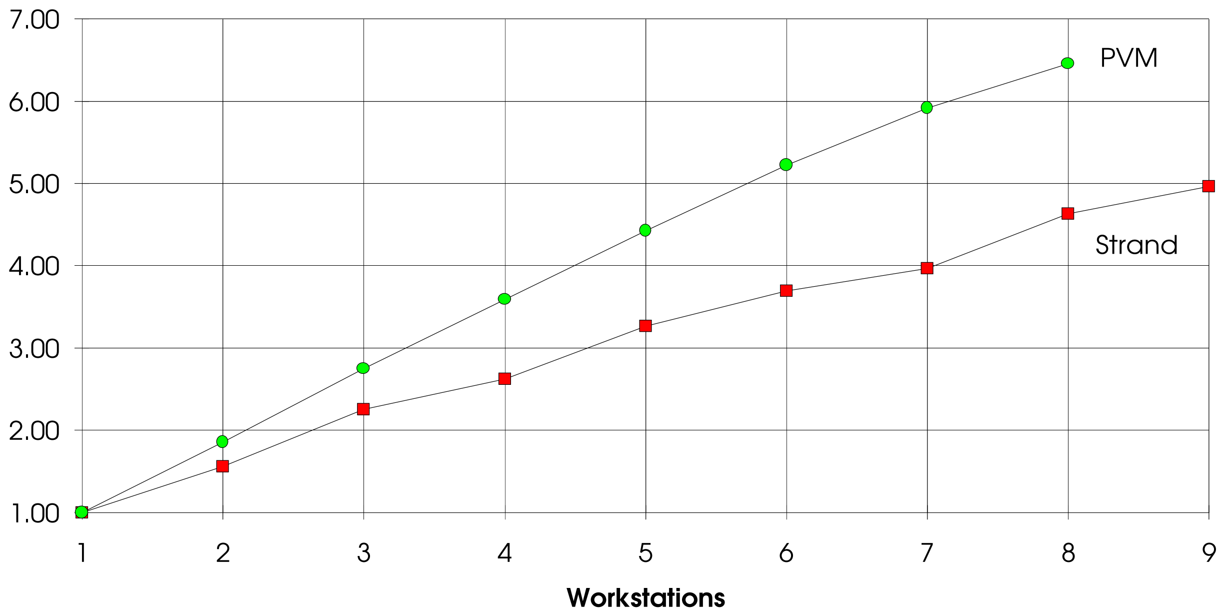


Figure 4.21: Speedup of fractal computation using PVM and Strand on SLC workstations

processors. With five RS6000 it is possible to improve a performance this way. The same is true

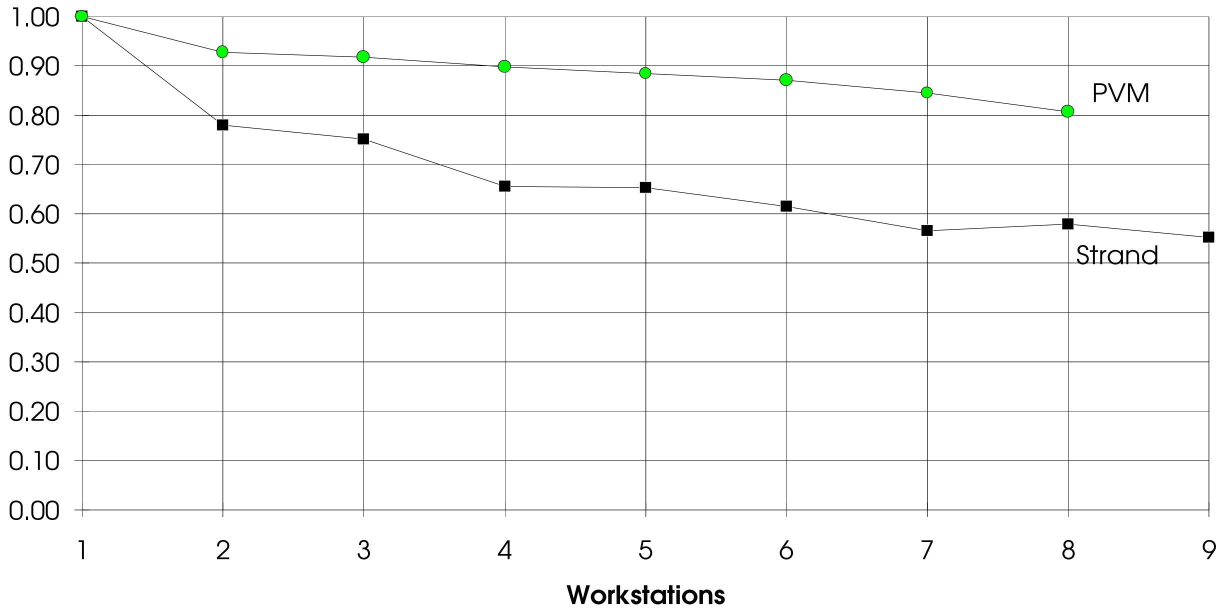


Figure 4.22: Efficiency of fractal computation using PVM and Strand on SLC workstations

| SPARC - 2 | | | | |
|---|---|---|---|---|
| No. of Processors | No. of Workers | Time [sec] | Speedup | Efficiency |
| 1 | 1 | 43.95 | 1.00 | 1.00 |
| 2 | 2 | 26.47 | 1.66 | 0.83 |
| 3 | 3 | 20.10 | 2.19 | 0.73 |
| 4 | 4 | 15.23 | 2.89 | 0.72 |
| 5 | 5 | 12.90 | 3.41 | 0.68 |
| 5 | 4 | 14.18 | 3.10 | 0.62 |
| RS6000 | | | | |
| No. of Processors | No. of Workers | Time [sec] | Speedup | Efficiency |
| 1 | 1 | 20.81 | 1.00 | 1.00 |
| 2 | 2 | 18.91 | 1.10 | 0.55 |
| 3 | 3 | 14.77 | 1.41 | 0.47 |
| 4 | 4 | 13.02 | 1.60 | 0.40 |
| 5 | 5 | 12.10 | 1.72 | 0.34 |
| 5 | 4 | 10.77 | 1.93 | 0.39 |

Table 4.10: Fractal computation comparision on SUN SPARCstation 2 and RS6000 platforms using PVM

for the SPARCstation 2 configuration but with a greater number of processors. Unfortunately only five SPARCstation 2 were available for the study hence this effect has been shown in Fig. 4.24 for SLC stations. The crossing of the curves A and B is also coursed in that case by the fact that the SLC processor is rather slow.

| No. of nodes | 1 Buffer | | | 2 Buffers | | |
|---|---|---|---|---|---|---|
| | Time | Speedup | Efficency | Time | Speedup | Efficency |
| 1 | 174 | 1.00 | 1.00 | 174 | 1.00 | 1.00 |
| 2 | 93 | 1.87 | 0.94 | 90 | 1.93 | 0.97 |
| 3 | 63 | 2.76 | 0.92 | 62 | 2.81 | 0.94 |
| 4 | 48 | 3.63 | 0.91 | 47 | 3.70 | 0.93 |
| 5 | 40 | 4.35 | 0.87 | 39 | 4.46 | 0.89 |
| 6 | 34 | 5.12 | 0.85 | 32 | 5.44 | 0.91 |
| 7 | 30 | 5.80 | 0.83 | 29 | 6.00 | 0.86 |
| 8 | 28 | 6.21 | 0.78 | 25 | 6.96 | 0.87 |
| 9 | 24 | 7.25 | 0.81 | 23 | 7.57 | 0.84 |
| 10 | 23 | 7.57 | 0.76 | 21 | 8.29 | 0.83 |
| 11 | 21 | 8.29 | 0.75 | 20 | 8.70 | 0.79 |
| 12 | 20 | 8.70 | 0.73 | 18 | 9.67 | 0.81 |
| 13 | 19 | 9.16 | 0.70 | 17 | 10.24 | 0.79 |
| 14 | 17 | 10.24 | 0.73 | 16 | 10.88 | 0.78 |

Table 4.11: Infulence of double buffering on SR fractal computation

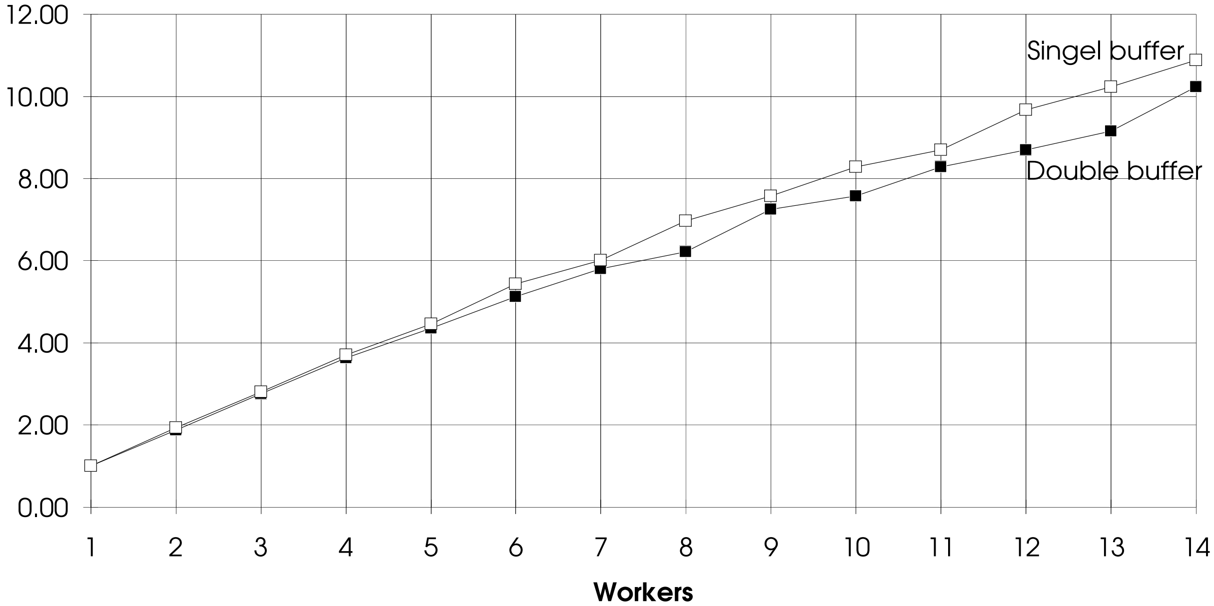

Figure 4.23: Influence of double buffering on SR fractal computation speedup

The SPARCstation 2 processor is twice times slower than RS6000 with the fractal computation but with 5 nodes the computational time is almost equal for both architectures. It leads to a general conclusion that Ethernet is a far too slow communication layer for the RS6000. The fractal distributed computations in such a system are very inefficient.

The summary of the influence of the model choice on various software running on SLC workstations. tools under study have been summarized in Tables 4.12 , 4.13 and 4.14. For ANSA only model B was investigated. These results are depicted in Figures 4.25, 4.26, 4.27

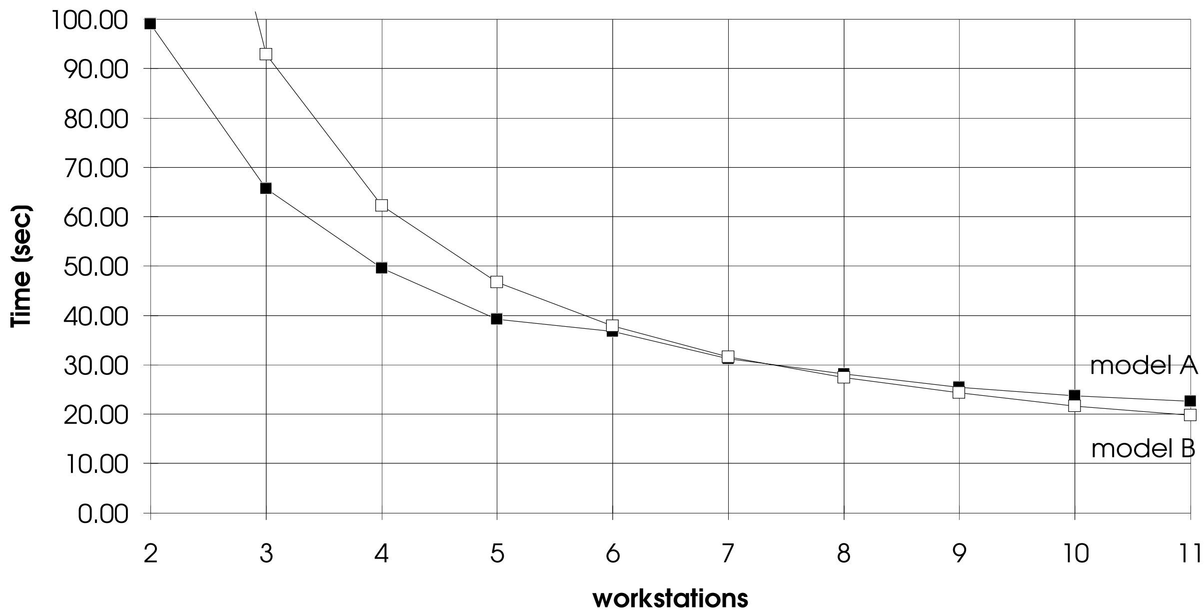


Figure 4.24: The processor farm model A and B comparison on SLC using PVM

and 4.28.

| No. of | Model A | | | | Model B | | | |
|---|---|---|---|---|---|---|---|---|
| Nodes | Workers | Time [sec] | Speedup | Efficiency | Workers | Time [sec] | Speedup | Efficiency |
| 1 | 1 | 174.0 | 1.00 | 1.00 | 1 | 174.00 | 1.00 | 1.00 |
| 2 | 2 | 99.08 | 1.76 | 0.88 | 1 | 184.84 | 0.94 | 0.47 |
| 3 | 3 | 65.68 | 2.65 | 0.88 | 2 | 92.88 | 1.87 | 0.62 |
| 4 | 4 | 49.54 | 3.51 | 0.88 | 3 | 62.20 | 2.80 | 0.70 |
| 5 | 5 | 39.20 | 4.44 | 0.89 | 4 | 46.79 | 3.72 | 0.74 |
| 6 | 6 | 36.82 | 4.73 | 0.79 | 5 | 37.84 | 4.60 | 0.77 |
| 7 | 7 | 31.25 | 5.57 | 0.80 | 6 | 31.54 | 5.52 | 0.79 |
| 8 | 8 | 28.21 | 6.17 | 0.77 | 7 | 27.43 | 6.34 | 0.79 |
| 9 | 9 | 25.45 | 6.84 | 0.76 | 8 | 24.37 | 7.14 | 0.79 |
| 10 | 10 | 23.72 | 7.34 | 0.73 | 9 | 21.62 | 8.05 | 0.80 |
| 11 | 11 | 22.57 | 7.71 | 0.70 | 10 | 19.82 | 8.78 | 0.80 |

Table 4.12: PVM performance of model A and B on SUN SLC workstations

The same experiments have been performed on SUN SPARCstation 2 workstations. The presented results lead to the following conclusions:

- The obtained efficiency depends on a number of nodes and is difficult to give one answer which software platform is the best one,
- Model A is very inefficient with a small number of nodes for the P4 platform. The local communication - that is communication between the farmer and the worker on the same

| No. of | Model B | | | |
|---|---|---|---|---|
| nodes | Workers | Time [sec] | Speedup | Efficiency |
| 1 | 1 | 174.00 | 1.00 | 1.00 |
| 2 | 2 | 178.45 | 0.98 | 0.49 |
| 3 | 4 | 89.71 | 1.94 | 0.65 |
| 4 | 6 | 60.26 | 2.89 | 0.72 |
| 5 | 8 | 45.14 | 3.86 | 0.77 |
| 6 | 10 | 36.65 | 4.75 | 0.79 |
| 7 | 12 | 30.64 | 5.68 | 0.81 |
| 8 | 14 | 26.35 | 6.60 | 0.83 |
| 9 | 16 | 23.32 | 7.46 | 0.83 |
| 10 | 18 | 20.97 | 8.30 | 0.83 |
| 11 | 20 | 19.07 | 9.12 | 0.83 |

Table 4.13: ANSA implentation of model B on SUN SLC Workstations performance

| No. of | Model A | | | | Model B | | | |
|---|---|---|---|---|---|---|---|---|
| nodes | Workers | Time [sec] | Speedup | Efficiency | Workers | Time [sec] | Speedup | Efficiency |
| 1 | 1 | 174.0 | 1.00 | 1.00 | 1 | 174.0 | 1.00 | 1.00 |
| 2 | 2 | 163.8 | 1.06 | 0.53 | 1 | 178.0 | 0.98 | 0.49 |
| 3 | 3 | 87.72 | 1.98 | 0.66 | 2 | 88.64 | 1.96 | 0.65 |
| 4 | 4 | 58.91 | 2.95 | 0.74 | 3 | 59.07 | 2.95 | 0.74 |
| 5 | 5 | 42.13 | 4.13 | 0.83 | 4 | 43.26 | 4.02 | 0.80 |
| 6 | 6 | 35.03 | 4.97 | 0.83 | 5 | 34.86 | 4.99 | 0.83 |
| 7 | 7 | 28.10 | 6.19 | 0.88 | 6 | 28.38 | 6.13 | 0.88 |
| 8 | 8 | 23.22 | 7.49 | 0.94 | 7 | 24.29 | 7.16 | 0.90 |
| 9 | 9 | 21.82 | 7.97 | 0.89 | 8 | 21.40 | 8.13 | 0.90 |
| 10 | 10 | 19.30 | 9.02 | 0.90 | 9 | 18.51 | 9.40 | 0.94 |

Table 4.14: P4 performance of model A and B on SUN SLC workstations

node is time consuming and degrades the whole performance. Hence P4 is doing worse than ANSA and PVM for small number of nodes. This effect is even more evident on the SUN SPARCstation 2 than on SLC.

- P4 fast communication ability gives to it some advantages with greater numbers of nodes using the model A. So on 10 SLC, P4 is doing better than other an software under study.
- The model B has not shown such anomalies as the model A. P4 is the most efficient platform in the whole range under study. A little worse is ANSA.
- The differences between the software tools under study are greater on SLC than on the SUN SPARCstation 2.
- A very characteristic is the drop of the efficiency for a small number of the workstation using the model B. It is justified by the fact that the farmer node is not involved in task computations.

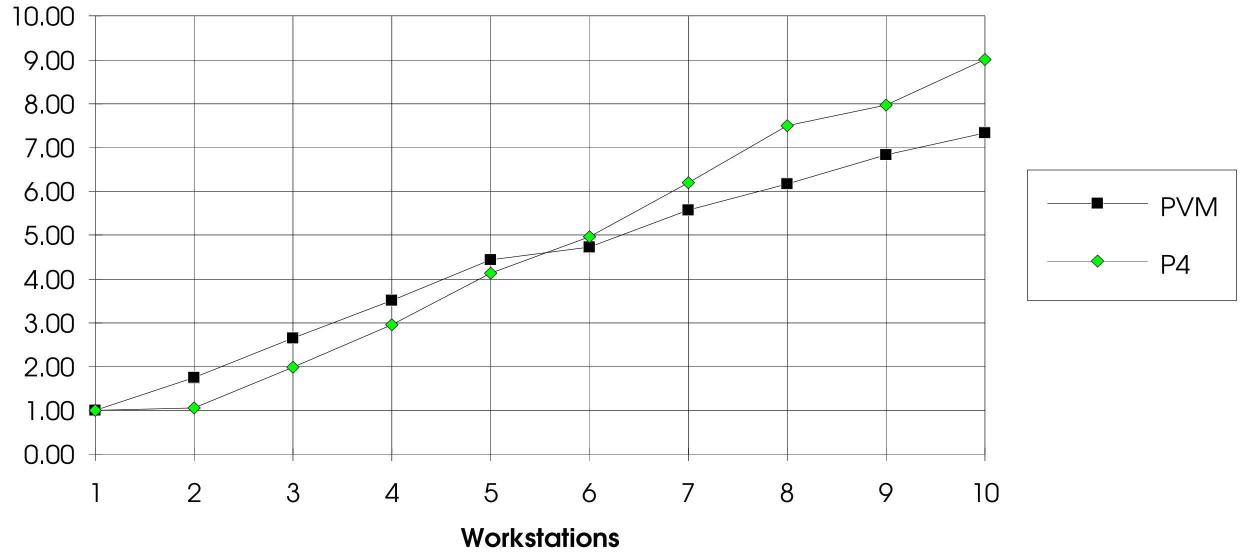


Figure 4.25: Fractal computation speedup over different software platforms using model A on SLC workstations

## 4.4 Ray tracing computation results

The ray tracing computations are much more demanding with comparison to the fractal. The relative performance of various workstations under study has been shown in Table 4.17 and in

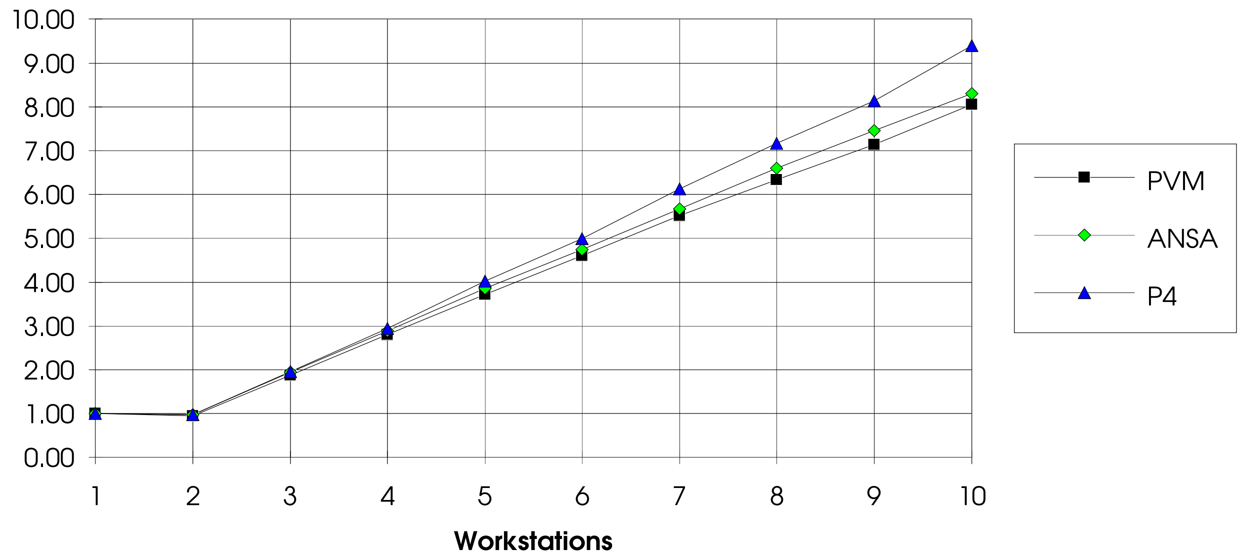


Figure 4.26: Fractal computation speedup over different software platforms using model B on SLC workstations

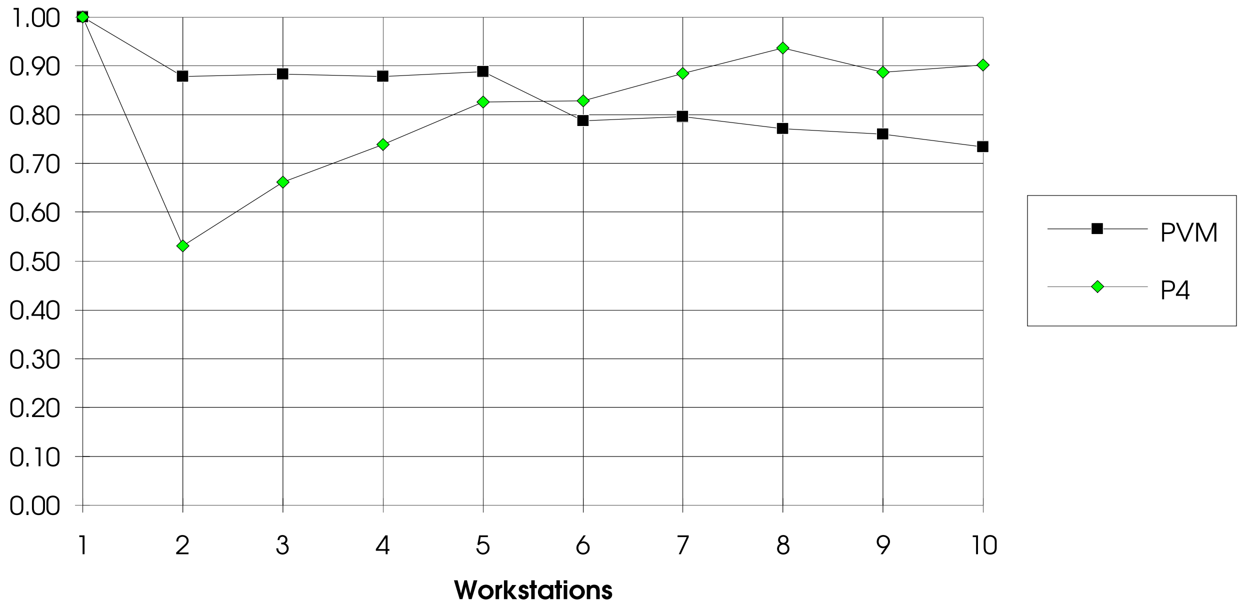

Figure 4.27: Fractal computation efficiency over different software platforms using model A on SLC workstations

Fig. 4.31. It is interesting that very fast machines as HP9000/720 and RS-6000 observed a very

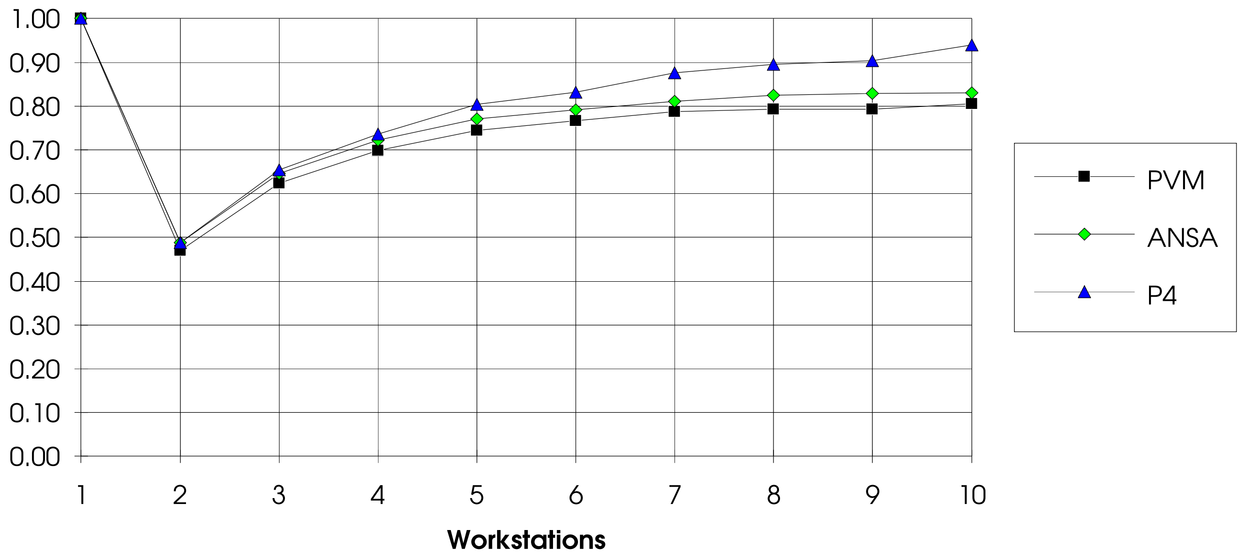

Figure 4.28: Fractal computation efficiency over different software platforms using model B on SLC workstations

| No. of nodes | Model A Workers | Time [sec] | Speedup | Efficiency | Model B Workers | Time [sec] | Speedup | Efficiency |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 43.95 | 1.00 | 1.00 | 1 | 43.95 | 1.00 | 1.00 |
| 2 | 2 | 27.95 | 1.57 | 0.79 | 1 | 54.63 | 0.80 | 0.40 |
| 3 | 3 | 18.89 | 2.33 | 0.78 | 2 | 27.07 | 1.62 | 0.54 |
| 4 | 4 | 15.67 | 2.80 | 0.70 | 3 | 19.88 | 2.21 | 0.55 |
| 5 | 5 | 13.17 | 3.34 | 0.67 | 4 | 14.70 | 2.99 | 0.60 |
| 6 | 6 | 12.21 | 3.60 | 0.60 | 5 | 12.77 | 3.44 | 0.57 |
| 7 | 7 | 10.89 | 4.04 | 0.58 | 6 | 10.40 | 4.23 | 0.60 |

Table 4.15: PVM performance of fractal computation with model A and B respectively on SUN SPARCstation2 workstations

| No. of nodes | Model A Workers | Time [sec] | Speedup | Efficiency | Model B Workers | Time [sec] | Speedup | Efficiency |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 43.95 | 1.00 | 1.00 | 1 | 43.95 | 1.00 | 1.00 |
| 2 | 2 | 64.42 | 0.68 | 0.34 | 1 | 46.59 | 0.94 | 0.47 |
| 3 | 3 | 32.03 | 1.37 | 0.46 | 2 | 25.96 | 1.69 | 0.56 |
| 4 | 4 | 21.74 | 2.02 | 0.51 | 3 | 19.07 | 2.30 | 0.58 |
| 5 | 5 | 14.42 | 3.05 | 0.61 | 4 | 15.25 | 2.88 | 0.58 |
| 6 | 6 | 16.27 | 2.70 | 0.45 | 5 | 12.62 | 3.48 | 0.58 |

Table 4.16: P4 performance of fractal computation with model A and B respectively on SUN SPARCstation2 workstations

substantial degradation ( 50% ) of relative performance with comparison with that obtained for the fractal. The slow workstations as SLC and Intel 486 obtain a similar relative performance as for the fractal. The general conclusion is that for the ray tracing computation the gap between communication efficiency and computation power is narrower. In the another words the LAN based computer is more close to an equilibrium with powerful workstations.

| No. | Processor Type | Compiler | Time [sec] | Relative performance |
|---|---|---|---|---|
| 1 | T800/20 | 3LC v2.1 | 1788 | 1.00 |
| 2 | SUN SLC | UNIX C | 1372 | 1.30 |
| 3 | SUN IPC | UNIX C | 958 | 1.87 |
| 4 | Intel 486/25 | UNIX C | 914 | 1.96 |
| 5 | Intel 486/33/256 | UNIX C | 646 | 2.77 |
| 6 | SPARC 2 | UNIX C | 388 | 4.61 |
| 7 | CONVEX C220 | UNIX C | 341 | 5.24 |
| 8 | RS-6000-320 | XLC | 323 | 5.54 |
| 9 | I860/40 | GNU | 298 | 6.00 |
| 10 | HP 9000/720 | UNIX C | 212 | 8.43 |

Table 4.17: Ray tracing computation sequential time on different processors

This conclusion is confirmed by the results presented in Table 4.18 and in Fig. 4.32. These

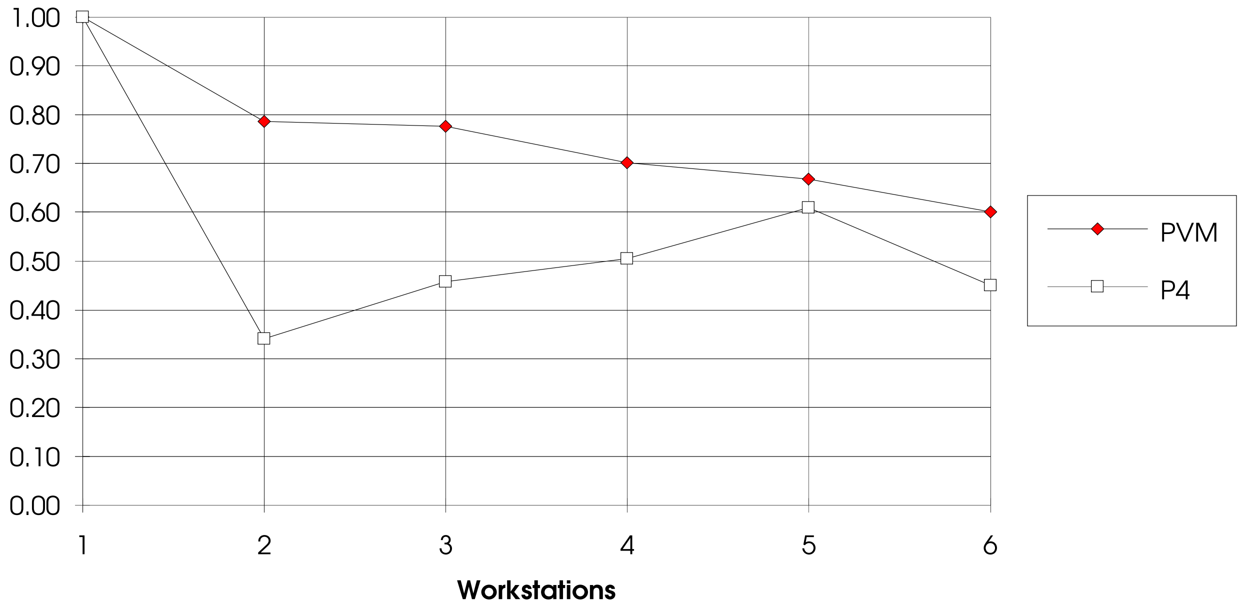

Figure 4.29: Fractal computation efficiency over different software platforms using model A on SUN SPARCstation 2 workstations

figures concern the PVM implementation of the model A processor farm. For the ray tracing

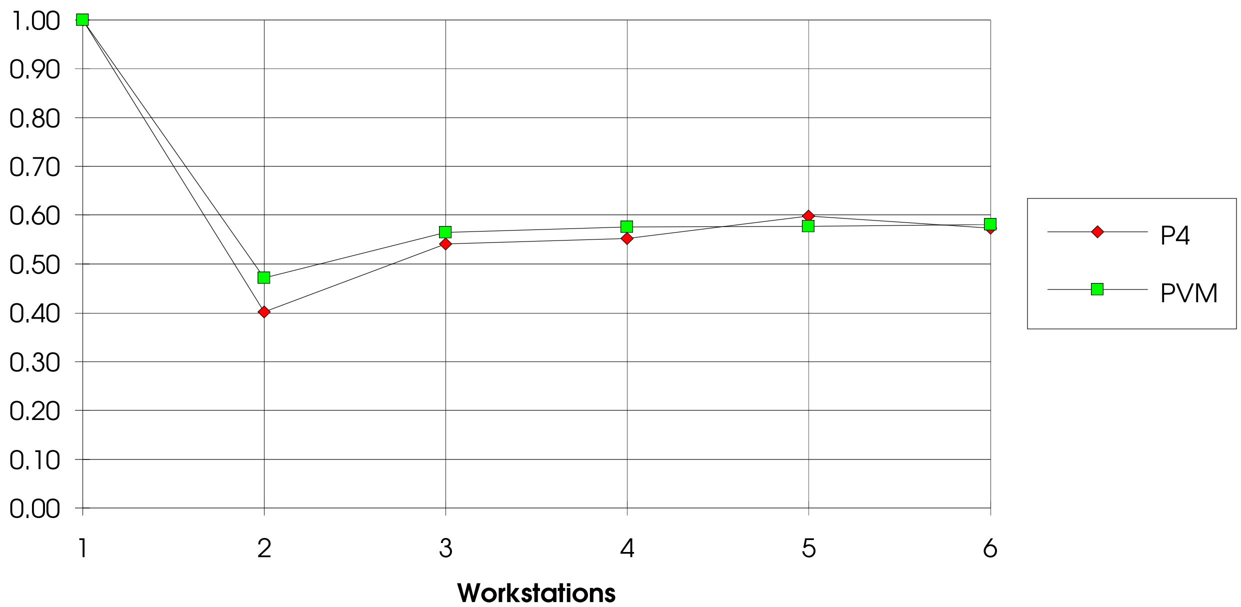

Figure 4.30: Fractal computation efficiency over different software platforms using model B on SUN SPARCstation 2 workstations

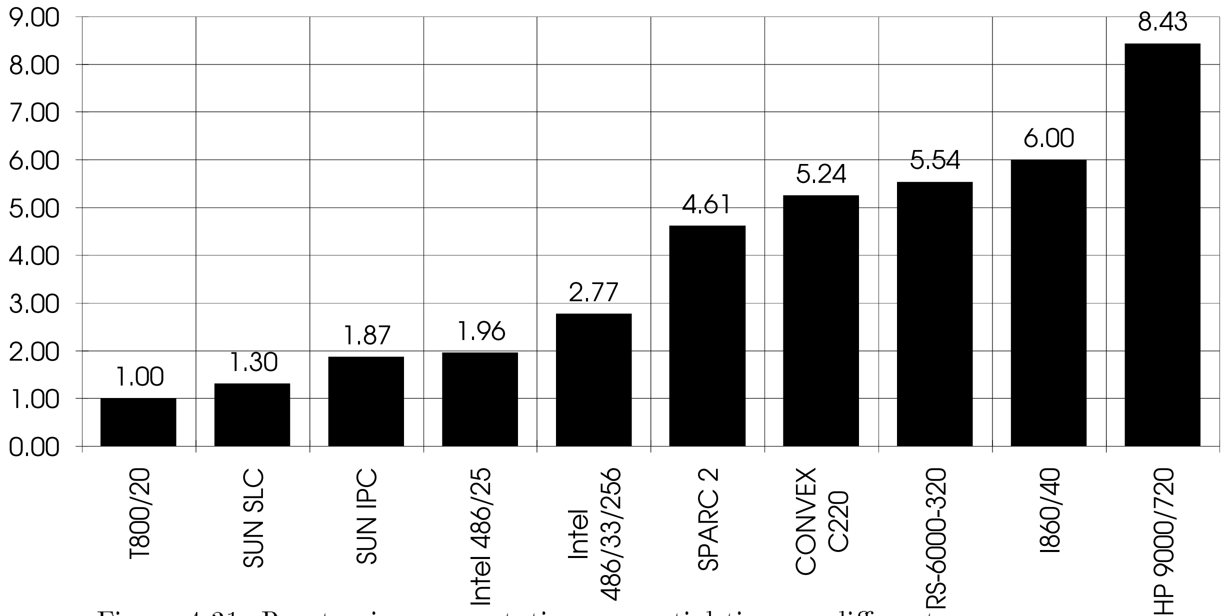


Figure 4.31: Ray tracing computation sequential time on different processors

computations five RS-6000 are doing better than seven SPARCstation 2. This result is opposite to those reported in Table 4.10. In general via distributed computing in the heterogeneous environment built of the seven SPARCstation 2 five RS-6000 and one HP9000/720 it was possible to speedup the ray tracing computation by 51 times. It is a very impressive example of the efficiency of the LAN distributed computing.

In general the obtained results for the ray tracing computation are very similar to those obtained for the fractal. The only difference comes from the different computational complexity.

| No. of Experiment | Procesor Type | Compiler | Time [sec] | Relative Performance |
|---|---|---|---|---|
| 1 | T800/20 | 3LC v2.1 | 1788 | 1.00 |
| 2 | SUN SLC | UNIX C | 1372 | 1.30 |
| 3 | SPARC 2 | UNIX C | 388 | 4.61 |
| 4 | RS-6000-320 | XLC | 323 | 5.54 |
| 5 | HP 9000/720 | UNIX C | 212 | 8.43 |
| 6 | 15*T800/20 | 3LC v2.1 | 123 | 14.54 |
| 7 | 7*Sparc 2 | UNIX C | 72.3 | 24.70 |
| 8 | 5*RS6000 | UNIX C | 65.7 | 27.19 |
| 9 | 7*S2+5*RS+1*HP | UNIX C | 34.5 | 51.69 |

Table 4.18: The maximum performance for ray tracing computation over PVM platform on various architectures

The comparison of the ray tracing and the fractal computations in term of the obtained speedup is depicted in Fig. 4.33.

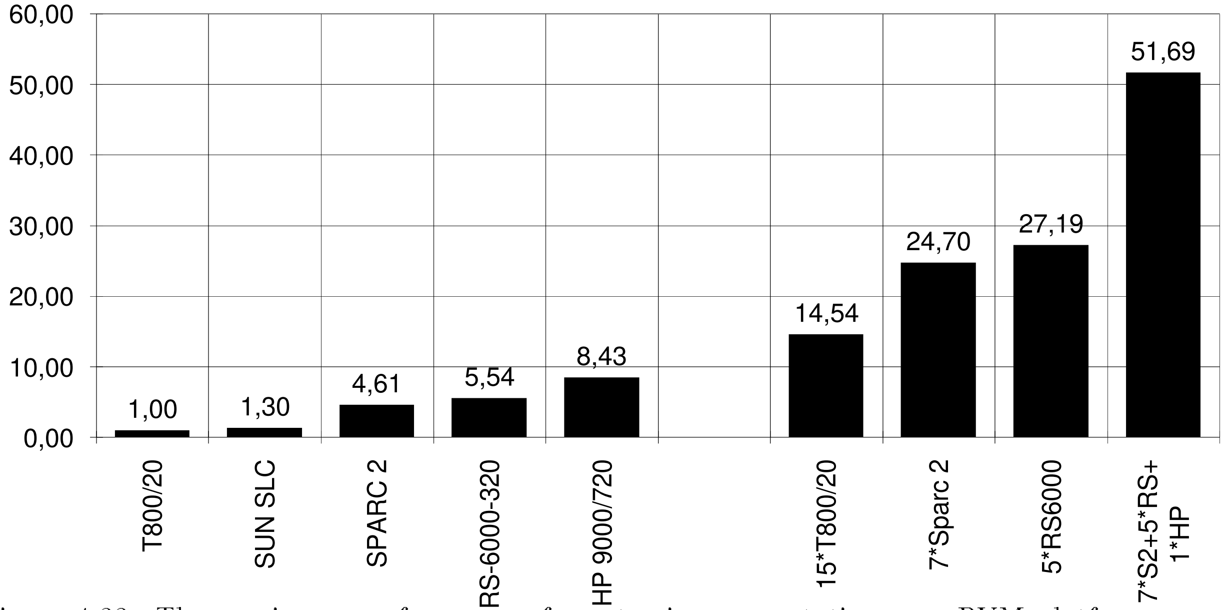


Figure 4.32: The maximum performance of ray tracing computation over PVM platform on various architectures

psfigfile=com.ps,height=9.2cm,width=16cm

Figure 4.33: Speedup of fractal and ray tracing computations over PVM on SLC workstations versus number of nodes

In general the LAN distributed computations are more efficient when a computing is a dominated factor over a communication. This is illustrated by the study of influence of a ray tracing picture precision on the obtained performance using PVM on SLC workstations. The results are shown in Table 4.19 and 4.20 and in Fig. 4.34 , 4.35. For more demanding case (r=2, v=0.2) has been obtained almost perfect speedup and efficiency.

| No. of nodes | Time | Speedup | Efficency |
|---|---|---|---|
| 1 | 345.63 | 1.00 | 1.00 |
| 2 | 174.16 | 1.98 | 0.99 |
| 3 | 116.63 | 2.96 | 0.99 |
| 4 | 88.09 | 3.92 | 0.98 |
| 5 | 70.84 | 4.88 | 0.98 |
| 6 | 59.13 | 5.85 | 0.97 |
| 7 | 50.86 | 6.80 | 0.97 |
| 8 | 45.44 | 7.61 | 0.95 |

Table 4.19: Ray tracing computaions using PVM implementation of model A processor fram ; ray per pixel=1, variance=1

| No. of nodes | Time | Speedup | Efficency |
|---|---|---|---|
| 1 | 1345.9 | 1.00 | 1.00 |
| 2 | 677.48 | 1.99 | 0.99 |
| 3 | 449.33 | 3.00 | 1.00 |
| 4 | 337.35 | 3.99 | 1.00 |
| 5 | 269.29 | 5.00 | 1.00 |
| 6 | 226.88 | 5.93 | 0.99 |
| 7 | 193.92 | 6.94 | 0.99 |
| 8 | 169.54 | 7.94 | 0.99 |

Table 4.20: Ray tracing computaions using PVM implementation of model A processor fram , ray per pixel=2, variance=0.2

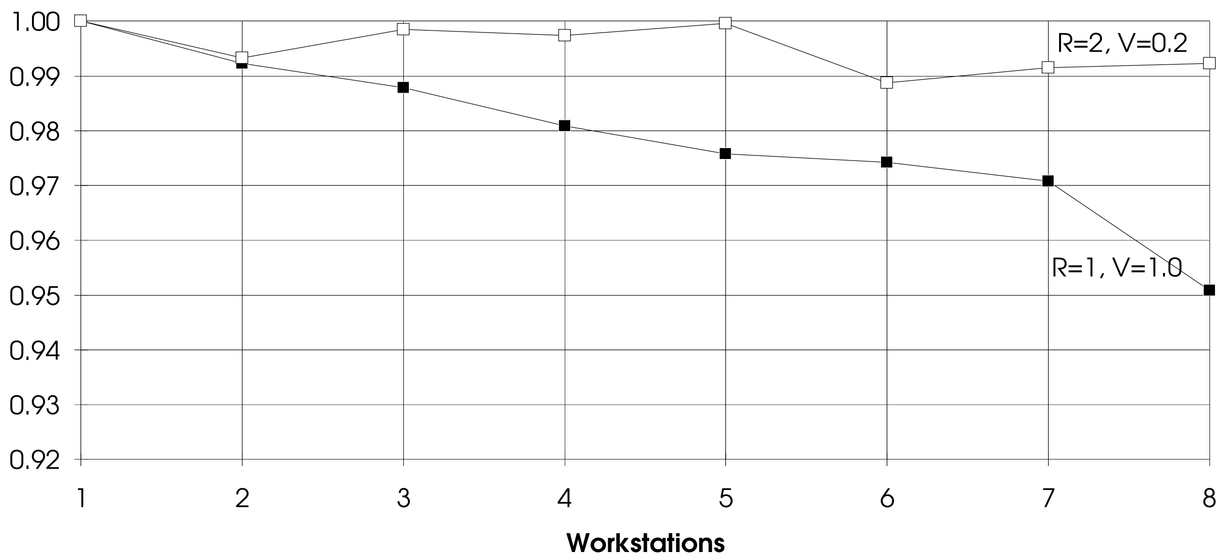


Figure 4.34: The influence of a ray tracing picture precision on the obtained speedup using PVM on SLC workstations

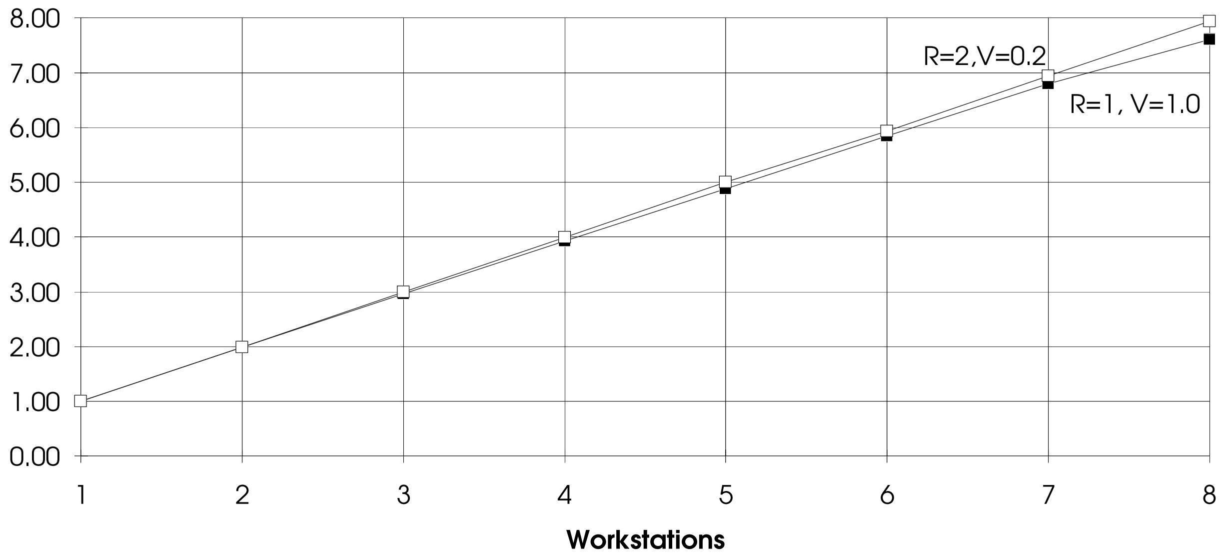


Figure 4.35: The influence of a ray tracing picture precision on the obtained efficency using PVM on SLC workstations

# Chapter 5

# Conclusions

The results of the presented experimental study and the analysis of the investigated software packages lead to the following general conclusions :

- The pakages for the LAN distributed computing provide a different degree of an internal parallelizm. Hence for same algorithms a potential parallelizm can not be fully exploited. The more advance software tools like Strand, ANSA, and SR provide mechamisms that are better suited to parallel computation demands. The investigated processor farm models are characterized by a very high degree of the potential parallelizm so the constrains introduced by the computational model could be observed.

- There no exesits a relation between the communication efficiency and the complexity of the software package. A very complex package such as ANSA provides very efficient communication operations. Unfortunately, the communication efficiency of some tools depends on the data length. The best examples of such bechaviour are Strand and SR which are better suited for the long data blocks transfer.

- The internal parallelizm of the computational model is a factor that could outperform the pure communication efficiency. Strand is the best example of that . Hence the comparision of communication efficiency only, is not a good measure of the whole package evaluation.

- The speedup of LAN distributed computations is very difficult to predict. It is, of course a much more difficult problem than for sequential computations. There has to be reached the same kind of equilibrium between a computationnal power and a communication efficiency. The point of this equilibrium moves because the computational power represented by nodes of the system changes for various computational problems.

- A very important factor of the LAN distributed computation is a task decomposition. In the case of computational problems with geometrical parallelizm this corresponds to the task granularity choice. A fault in this choice may degrade the whole system performance. The problems under study performed much better for the small than for the large granularity. Unfortunately, the meaning of a small or large granularity changes with a problem. From the presented results it is evident that the number of tasks a problem is devided into, should be at last ten time greater than the number of the LAN computer nodes.

- The complexity of the distributed program depends substantially on the computational model. Linda and Strand provide the most elegant and compact solution of the processor farm implementation provides . PVM and P4 are effective tools but need more effort to organize the distributed computations. The programmmig with these last two mentioned packages presents more complex communication connections between processes and an error prone task. The longer time of software testing and debbuging could be the same time not justified by the final gains from a slightly better performance.

- A great attention should be paid to the characteristic features of various packges such as for instance :
  - pure efficiency of communication of Strand using small data blocks,
  - very inefficient local communication in P4,

- Reagarding to the harware platfrom it is evident that a degradation of efficiency is observed for a small number of more powerfull workstations such as RS-6000 and SPARCstation 2 than for slower machines such as SLC. This bechaviour is coursed by the Ethernet communication saturation. It leads to the hipoteze that the maximum computational power that could be obtained from the LAN computer is independent of the types of the system that is built of, and converge to the same value. It ought to be proved that many more experiments with greater systems than were available for the reported study should be done.

After a year of expirience in the area of the LAN distributed computing we may conclude that this is really the way to get a big computational power but a lot more must done to gain all possible advantages from this approach.

Chapter 6

# Summary

# Bibliography


[1] J. Kobielus LANs Bid for Distributed Power, Network World, Dec. 3, 1990.

[2] C.M. Pancake, D. Bergmark, *Parallel Programming Languages: Do They Meet the Needs of Scientists and Engineers?* , Cornell Theory Center, CTC89TR18.

[3] A.H.Karp, H.P. Flatt, Measuring Parallel Processor Performance, Communication of the ACM, May 1990 Vol.33, Number 5.

[4] 3L Parallel C User Guide, 1989 3L Ltd Technical Documentation.

[5] X. Sun, J.L. Gustafson, Toward a Better Parallel Performance Metric, Parallel Computing, Vol.17, Dec. 1991.

[6] X. Sun, L.Ni, Another View on Parallel Speedup, Proc. of Supercomputing'90.

[7] M. Gajęcki, K. Jędrzejek, K.Zieliński, *Parallelization of the DBW Render Package on a Transputer Farm* Machine GRAPHICS & VISION (przyjęto do publikacji)

[8] M. Gajęcki, K.Jędrzejek, K.Zieliński Load Balancing in the Parallel Implementation of the DBW-Render IEE Proceedings-E (złożono do publikacji)

[9] The Charm 3.2 Programming Language Manual, Dec. 18, 1992, TR, Univ. Of Illinois at Urbana Champaign.

[10] A.S. Grimshaw Easy-to-use Object-Orientet Parallel Processing with Mentat, Univ. of Virginia, 1992, TR No. CS-92-32.

[11] I. Foster et. al. Productive Parallel Programming: The PCN Approach, To appear in: Scientific Programming. 1993

[12] R. Bisian, A. Forin, Architectural Support for Multilanguage Parallel Programming on Heterogeneous Systems, Sec. Int. Conf. on ASPLOS, 1987.

[13] C.H. Cap, V. Strumpen, High Performance Platform for Parallel Computing in a Distributed Workstation Environment, Institut für Informatik, Univ. Zurich, 1992, TR IFI-92-07.

[14] D.L. Cohn, W.P. Delaney, M. Tracey, Platfrom for Heterogeneous Distributed Operating Systems, Proc. of the Usenix Distributed and Multiprocessor Systems Workshop, 1989,

[15] T. Geigel, M. Pagan, A Distributed Application for the Pharros Project, Proc. Int. Conf. on Parallel Proc. 1988.

[16] K.L Lorh, L. Nentwig, J. Muller, DAPHNE: Support for Disributed Computing in Heterogeneous Environment, European Workshop Proceedings, Progress in Distributed Operating Systems and Distributed Management, Springer-Verlag, 1989.

[17] M. Sullivan, D. Anderson, Marionette: A system for Parallel Distributed Programming Using a Master/Slave Model. Proc. of the ninth International Conference on Distributed Computer Systems, 1989.

[18] C.A Waldspurger, T. Hogg, B.A. Huberman, Spawn : A Distributed Computational Economy, IEEE Transacton on Software Engineering, 1992, February, Vol.18, No.2.

[19] C.B. Weinstock, Performance and Reliability Enhancement of the Durra Runtime Environment. SEI at CMU Univ., February 1989, CMU/SEI-89-TR-8.

[20] L. Wittie, C. Maples, Merlin: Massively Parallel Heterogeneous Computing. Proc. of the 1989. Int. Conf. on Parallel Proc. 1889.

[21] S.A. Yemini, G.S. Goldszmidt, A.D. Stoyenko, Yi-Hsiu Wei, CONCERT: A high-Level-Language Approach to Heterogeneous Distributed Systems, Proc. of the ninth International Conference on Distributed Computer Systems, 1989.

[22] G.R. Andrews. Synchronizing Resources. *ACM Trans. on Prog. Languages and Systems* , 4 (Oct. 1981), 405-430.

[23] G.R. Andrews. The distributed programming language SR – mechanisms, design and implementation. *Software–Practice and Experience 12*, 8 (Aug. 1982),719-754.

[24] G.R. Andrews and R.A. Olsson. The evolution of the SR language. *Distributed Computing 1*, 3 (July 1986), 133-149.

[25] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin and G. Townsend. An overview of the SR language and implementation. *ACM Trans. on Prog. Languages and Systems 10*, 1 (Jan. 1988), 51-86.

[26] G.R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, 1991.

[27] G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in practice*. Benjamin/Cummings Publishing Company, 1992.

[28] C.A.R. Hoare,*Communicating Sequential Processes.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

[29] U.S. Department of Defence, *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A, 1983.

[30] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1976.

[31] R.K. Raj, E.D. Tempero, H.M. Levy, A.P. Black, N.C. Hutchinson and E.Jul. Emerald: A General-Purpose Programming Language. *Software – Practice and Experience 21*, 1 (Jan. 1991), 91-11.

[32] Strand Software Technologies Limited: STRAND'88 User Manual, Buckingham Release, June 1990.

[33] Foster I., Taylor S.: STRAND - New Concept in Prallel Programming, Prentice-Hall, Inc, 1990.

[34] Sterling L., Shapiro E.: The Art of Prolog, The MIT Press Cambridge, 1986

[35] Bratko I.: Prolog Programming For Artificial Inteligence, Adison Wesley, 1986

[36] Carriero, N., Gelernter, D. *How to write parallel programs: A guide to the perplexed*, ACM Computer Surveys, wrzesień 1989.

[37] Gelernter, D. *Parallelism to the People, or, Experience with Programminig in Linda.*

[38] Gelernter, D., Philbin, J. *Spending your Free Time*, Byte, maj 1990.

[39] Ahuja, S., Carriero, N., Gelernter, D., Krishnaswamy, V. *Matching Language and Hardware for Parallel Computation in the Linda Machine*, IEEE Transactions on Computers, Vol. 37, nr 8, sierpień 1988.

[40] Lerer, W. *Linda Meets Unix*, IEEE, luty 1990.

[41] Gelernter, D. *Multiple Tuple Spaces in Linda*, PARLE 89.

[42] Dokumentacja Systemu POSYBL-1.102.

[43] P. Pardyak, Group Communication in an Object-Based Environment Proc. Second Int. Worskshop on Object Orientation in O. S., Dourdan, France, Sept. 24-25, 1992.

[44] G. Czajkowski, K. Zieliński, STRAND88 - język współbieżnego programowania w logice Informatyka nr 1 1993

[45] G. Czajkowski, K. Zieliński, Linda - Środowisko do Przetwarzania Równoległego i rozproszzonego w Sieciach Stacji Roboczych Informatyka (złożono do druku)

[46] M. Bijald, Gwóźdź, K. Zieliński, Rozproszone Przetwarzanie w systemach Otwartych - Model ANSA Informatyka, nr. 3, 1993.

[47] M. Gajęcki, K. Zieliński, A practical Experience with LAN Distributed Computing - PVM and ANSA Case Study , (sent for publication)

[48] A. Laurentowski, K. Zielński, SR - Obiektowo Zorientowany Język Programowania Rozproszonego Informatyka (złóżono do druku)

[49] P. Pardyak, K. Zieliński, Emerald - Język i System Roproszonego Programowania Obiektowego Informatyka (złóżono do druku)

[50] R. Butler, E Lusk, User's Guide to the p4 Programming System, Argone National Labolatory, 1992, TR ANL-92/17.

[51] Solaris Strategy, SunSoft TR 6/21/12

[52] A. Beguelin et. al., A Users' Guide to PVM Parallel Virtual Machine, TR, Oak Ridge Natinal Labolatory, 1991.

[53] ANSA Reference Manual, AMP Ltd., 1991.

# List of Figures

# List of Tables

## Appendix A

# Data transfer test code programs

## A.1 PVM communication test program

```
/* This program measures mean transmission times in PVM       */
/* The measurement is carried out for two kinds of char packs */
/* short (16-1024 bytes, with step 16 bytes)                  */
/* and long ones (1-64 Kbytes, with step 1 Kbyte)             */
/* A pack is sent and received 5000 times                     */

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int times,sizebuf,razy,step;
char buf[524288];

int procesor;
int me;

struct timeval tp1,tp2;
struct timezone tpz;
/*****************************************************************/
double czas(ptime1 , ptime2)
 struct timeval *ptime1;
 struct timeval *ptime2;
 {
   return ((ptime2->tv_sec - ptime1->tv_sec +
           (ptime2->tv_usec - ptime1->tv_usec)/1000000.0)*1000);
 }
/*****************************************************************/

master()
{
  int i,j;
  double time,tpp;

  for( i=0; i<razy; i++){

    gettimeofday(&tp1,&tpz);          /* start timer */

    for( j=0; j<times; j++){

      initsend();
      putbytes( buf, sizebuf);
      snd("st", procesor, 100);       /* send to worker */

      rcv( 200 ) ;                    /* receive from worker */
      getbytes( buf, sizebuf);
    }

    gettimeofday(&tp2,&tpz);          /* stop timer */

    time = czas(&tp1,&tp2)/1000.0;
    tpp = time/2/times*1000.0;
    printf("%6d  %7.4f \n", sizebuf, tpp );

    sizebuf += step;
```

```
  }

}
/******************************************************************/

worker()              /* worker() pingpongs packets back to master */
{
  int i,j;

  for( i=0; i<razy; i++){

    for( j=0; j<times; j++){
      rcv( 100 ) ;
      getbytes( buf, sizebuf);

      initsend();
      putbytes( buf, sizebuf);
      snd("st", 0, 200);
    }
    sizebuf += step ;

  }
}
/******************************************************************/

main ( argc, argv )
int argc;
char *argv[];
{

  me = enroll( "st");
  if (me<0) exit(0);

  if (me==0) {
    procesor = initiate("st","SUN4");    /* start worker proces */
    printf("\nSTART\n");
  }

  sizebuf = 16;
  step = 16;
  times = 5000;
  razy = 64;

  if (me==0) master();
  else       worker();

  sizebuf = 1024;
  step = 1024;
  times = 5000;
  razy = 64;
  if (me==0) master();
  else       worker();
  if (me==0) printf("\nEND\n");
  leave();
  exit(0);
}
```

## A.2 P-4 communication test program

```
/* This program measures mean transmission times in P4       */
/* The measurement is carried out for two kinds of char packs */
/* short (16-1024 bytes, with step 16 bytes)                  */
/* and long ones (1-64 Kbytes, with step 1 Kbyte)             */
/* A pack is sent and received 5000 times                     */

#include <stdio.h>
#include <malloc.h>
#include <sys/time.h>
#include "p4.h"

int times,sizebuf,razy,step;
char buf[524288];

int me;
struct timeval tp1,tp2,tp3;
struct timezone tpz;

int size, type, from;

/******************************************************************/
double czas(ptime1 , ptime2)
 struct timeval *ptime1;
 struct timeval *ptime2;
 {
   return ((ptime2->tv_sec - ptime1->tv_sec +
           (ptime2->tv_usec - ptime1->tv_usec)/1000000.0)*1000);
 }
/******************************************************************/

master()
{
  int i,j,len;
  double time,tpp;

  p4_global_barrier(23);
  printf("\nSTART\n");

  type = 200;
  from = 1;

  for( i=0; i<razy; i++){

    gettimeofday(&tp1,&tpz);              /* start timer */

    for( j=0; j<times; j++){

      p4_send(100, 1, buf, sizebuf);

      p4_recv(&type, &from, buf, &len);

    }

    gettimeofday(&tp2,&tpz);              /* stop timer */
```

```
    time = czas(&tp1,&tp2)/1000.0;    /* total time */
    tpp = time/2.0/times*1000.0;      /* time per packet */
    printf("%6d  %7.4f \n", sizebuf, tpp );

    sizebuf += step;

  }
  printf("\nKONIEC\n");
}

slave()
{
  int len;
  int i,j;

  type = 100;
  from = 0;

  p4_global_barrier(23);
  for( i=0; i<razy; i++){

    for( j=0; j<times; j++){

      p4_recv(&type, &from, buf, &len);

      p4_send(200, 0, buf, sizebuf);
    }
    sizebuf += step;

  }
}

/**********************************************************************/

main ( argc, argv )
int argc;
char *argv[];
{
    p4_initenv(&argc, argv);
    me = p4_get_my_id();
    if (me==0) p4_create_procgroup();      /* start worker proces */

    sizebuf = 16;
    step = 16;
    times = 5000;
    razy = 64;

    if (me==0) master(); else slave();

    sizebuf = 1024;
    step = 1024;
    times = 5000;
    razy = 64;

    if (me==0) master(); else slave();

    p4_wait_for_end();
}
```

## A.3 SR communication test program

```
global Types

   # structures for time measure:
  type struct_td  = rec( tv_sec : int; tv_usec : int )
  type struct_tzd =  rec( tv_minutewest: int; tv_dsttime : int )

  optype BytesTrans( char_buf[ 1 : * ] : char )  {send}

end


global Constants
  const TAKTY  := 500
  const RAZY   := 128
  var prog_name : string [20]
  var argc : int
  var mach1, mach2 : cap vm
end


global GlobalOp
  import Types

# well-known C functions for real-time measure:
  external gettimeofday( tp: ptr struct_td;  tzp: ptr struct_tzd ) returns t:int
  external perror( val str: string[*] )

end


resource Master
  import Types, Constants, GlobalOp
  import Worker
  op getwork( cap Worker )
  op tomaster : BytesTrans
  op PreMeasure()

body Master()

  var tm1, tm2 : struct_td
  var tmz : struct_tzd
  var ret_gt, i : int
  var time : real    # tranmission loop time (in seconds)
  var tpp : real     # ping time (in msec)
  var workcap : cap Worker


  process  TransmitMaster
    var sizebuf : int
    receive getwork( workcap )
    PreMeasure()

    var delta : int

    fa delta := 16 to 1024 by 1008 ->
```

```
      printf("sizebuf     time        tpp \n" )
      fa i:=1 to RAZY ->
        ret_gt := gettimeofday( @tm1, @tmz )
        if ret_gt != 0 ->  perror( prog_name ) fi

        sizebuf := delta*i
        var charbuf[1:sizebuf] : char

# resource Master responds to pings from Worker:

fa j :=1 to TAKTY ->
          send workcap.frommaster( charbuf )
          receive tomaster( charbuf )
        af

        ret_gt := gettimeofday( @tm2, @tmz )
        if ret_gt != 0 ->  perror( prog_name ) fi
        time := tm2.tv_sec - tm1.tv_sec + (tm2.tv_usec - tm1.tv_usec)/1000000.0
        tpp := (time/2.0/TAKTY)*1000.0

# do output   :
        printf( "%d  %10.6f  %10.6f \n", sizebuf, time, tpp )
      af
    af
  end TransmitMaster


  proc PreMeasure()

    ret_gt := gettimeofday( @tm1, @tmz )
    if ret_gt != 0 ->  perror( prog_name ) fi
    fa i:=1 to TAKTY ->  af
    ret_gt := gettimeofday( @tm2, @tmz )
    if ret_gt != 0 ->  perror( prog_name ) fi

    time := tm2.tv_sec - tm1.tv_sec + (tm2.tv_usec - tm1.tv_usec)/1000000.0
    printf("Petla pusta:  %.6f seconds \n", time )

  end PreMeasure
end Master



resource Worker
  import Constants, Types, GlobalOp
  import Master
  op getmast( cap Master )
  op frommaster : BytesTrans

body Worker()
  var mastcap: cap Master

  process TransmitWorker
    var i : int
    var sizebuf : int

    receive getmast( mastcap )
```

```
    var delta : int

# do transmission:

    fa delta := 16 to 1024 by 1008 ->
      sizebuf := 1
      fa i:=1 to RAZY ->
          sizebuf := delta*i
          var charbuf[1:sizebuf] : char
          fa j :=1 to TAKTY ->
            receive frommaster( charbuf )
            send mastcap.tomaster( charbuf )
          af
      af
    af
  end TransmitWorker

end Worker


resource main
  import Constants, Types, GlobalOp
  import Master, Worker

body main()

  getarg(0, prog_name)
  var m1, m2 : string[20]
  var mast : cap Master
  var work : cap Worker
  var one := 1
  var two := 2

  argc := numargs()
  write ( "argc: ", argc )

  if argc = 0 ->
    write( "No arguments passed" )
    stop
  [] argc = 1 ->
    getarg( 1, m1 )
    m2 := m1
    write( "Run program on", m1  )
  [] argc > 1 ->
    getarg( 1, m1 )
    getarg( 2, m2 )
    write( "Run transmission between", m1, "and", m2 )
  fi

  write ( "m1:" , m1 )
  write ( "m2:" , m2 )

  locate( one, m1 )
  locate( two, m2 )
  mach1 := create vm() on one
  mach2 := create vm() on two

  mast := create Master() on mach1
```

```
  work := create Worker() on mach2

  # send created resources their counterpart's capabilities to enable
  # invoking send operations for transmission:

  send mast.getwork( work )
  send work.getmast( mast )

  final
    destroy mach1
    destroy mach2
  end

end main
```

## A.4 ANSA communication test program

```
/* Header Start */
/*
 *  File: client.dpl
 *  Author:M.Bijald M.Gwozdz
 *
 *  General Description:
 *    Client dla uslugi message passingu
 *
 */
#ifndef LINT
static char rcsid[] = "$Header: client.dpl,v 1.10 1992/03/18 14:05:30 can Exp $";
#endif
/* Header End */

#include "/home/tulip/ansa/install/sun4_sos_4.1/include/capsule/AWtime.h"
#include "AWstdio.h"
#include "kbinput.h"

#define S sizeof(char)
#define BLOCK 1024*S
#define END 2500

GLOBAL ansa_Cardinal Ansa_InitialTasks = 2;

! USE MyEcho
! DECLARE {ir} : MyEcho CLIENT
ansa_InterfaceRef ir;

#ifdef __STDC__
void mytest(int l, int bufsize, char z)
#else
void mytest(l,bufsize,z)
int l,bufsize;
char z;
#endif
{
 FILE *fp;
 struct timeval start;
 struct timeval stop;
 struct timezone tz;
 char *tab;
 BuffType buf,buf1;
 long rtime=0;
 int j;
 ansa_Voucher v;

        tab=calloc(1,l*bufsize);
        memset(tab,'a',l*bufsize);
        buf.data=tab;
        buf.length=l*bufsize;
        for (j=0;j<END;j++)
          {
            gettimeofday(&start, &tz);
!           {buf1} <- ir$Myecho(buf)
            gettimeofday(&stop, &tz);
```

```
            rtime =rtime + ((stop.tv_sec-start.tv_sec)*1000000-start.tv_usec+stop.tv_usec);
          }
        fp=fopen("/home/tulip/mabi/msgpas/master/msgpas.wyn","a+");
        fprintf(fp,"dl. bloku = %d%cB",l,z);
        fprintf(fp," czas %ld mikrosec, sredni czas pzeslania bloku danych %ld \n",rtime, rtime/(2*END));
        fclose(fp);
        rtime=0;
        free(tab);
 }

#ifdef __STDC__
void body( int argc, char *argv[], char *envp[] )
#else
void body( argc, argv, envp )
int argc;
char *argv[];
char *envp[];
#endif
{
 int i;

!   {ir} <- traderRef$Import("MyEcho", "/", "")

     for (i=16;i<=1024;i+=16)
       mytest(i,S,' ');

    for (i=1;i<=64;i++)
       mytest(i,BLOCK,'k');

!    ir$Discard
}



/* Header Start */
/*
 *
 *  File: server.dpl
 *  Author:M.Bijald,M.Gwozdz
 *
 *  General Description:
 *    Server dla uslugi message passingu
 */
#ifndef LINT
static char rcsid[] = "$Header: /disc2/usr/local/src/ANSAware4.0/master/src/examples/Echo/RCS/server.dpl,v 1.15 1992/01/08 15:09:07 can Exp $";
#endif
/* Header End */

#include "AWstdio.h"
#include "AWunistd.h"

#define PROPSIZE 1024
char propbuf[PROPSIZE];

! MANAGED MyEcho

! USE MyEcho
! USE Trader
! DECLARE { ir, p->ref } : MyEcho SERVER
```

```
#ifdef __STDC__
void body(int argc, char *argv[], char *envp[] )
#else
void body( argc, argv, envp )
int argc;
char *argv[];
char *envp[];
#endif
{
    ansa_InterfaceRef ir;

!   {ir} :: MyEcho$Create( 16 )
    (void) system_init_properties( propbuf, PROPSIZE, argc, argv );
!   {} <- traderRef$Export( "MyEcho", "/ansa/testservices", propbuf, ir )

}

typedef struct objstate {
    ansa_InterfaceRef ref;
    ansa_Boolean export;
} ObjState;

#ifdef __STDC__
ansa_StatePtr
Create_MyEcho_Object(int argc, char *argv[], char *envp[], InstantiateResult *results)
#else
ansa_StatePtr Create_MyEcho_Object(argc, argv, envp, results)
int argc;
char *argv[], *envp[];
InstantiateResult *results;
#endif
{
ObjState *p;

    p = (ObjState *)malloc(sizeof(ObjState));
!   {p->ref} :: MyEcho$Create(1)

    if( system_getenv("ANSA_MANAGED_EXPORT", envp) != (char *)0)
    {
p->export = ansa_TRUE;
     (void) system_init_properties( propbuf, PROPSIZE, argc, argv );
! {} <- traderRef$Export("MyEcho", "/ansa/testservices", \
    propbuf, p->ref)
    }
    else
     p->export = ansa_FALSE;

    results->length = 1;
    results->data = &(p->ref);
    return (ansa_StatePtr)p;
}

#ifdef __STDC__
ansa_Boolean
Destroy_MyEcho_Object(ansa_StatePtr state)
#else
ansa_Boolean
```

```
Destroy_MyEcho_Object(state)
    ansa_StatePtr state;
#endif
{
ObjState *p;

    p = (ObjState *)state;
    if (p->export == ansa_TRUE)
!    traderRef$Withdraw( p->ref )
!   {} :: MyEcho$Destroy( p->ref )
    free ((char *)p);
    return ansa_TRUE;
}

#ifdef __STDC__
int
MyEcho_Myecho(ansa_InterfaceAttr *_attr, BuffType src,BuffType *res)
#else
int MyEcho_Myecho( _attr, src, res)
ansa_InterfaceAttr *_attr;
BuffType src;
        BuffType *res;
#endif
{
        res->length=src.length;
        res->data=&(src.data[0]);

return 1;
}
```

## A.5 Strand communication test program

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% a STRAND88 module for testing mean transmission
% times in SUN/SPARC workstation network
% call format from STRAND shell:
%    ModuleName:do_it_all(Orders, TimeList)
% where TimeList is an output argument, containing results
% and Orders is list of tuples:
% { StartPacketSize, StepSize, LimitSize, SendTimes }
% Read warning at the description of do_it_all/2 process
% Module name is speed and should not be changed

-compile(free).
-exports([do_it_all/2, tasks_gen/2, snd/6]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% the work of rcv/6 is very simple: is sends back whatever
% it has received
rcv([ H | T ], Answers) :- Answers := [ H | AnswersT ], rcv(T, AnswersT).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% snd/6 sends the first pack and then leaves the work of sending
% the rest to the sender/6. A question arises: why there are
% two processes - sender/6 and snd/6, instead of just one.
% Answer sender/6 is synchronised by the return pack availability,
% while snd/6, which sets up a connection, expects no pack addressed to
% itself
snd(PacksToSend, Pack, PacksIn, PacksOut, TailIn, TailOut) :-
   PacksOut := [ Pack | PacksOut1 ],
   PacksToSend1 is PacksToSend - 1,
   sender(PacksToSend1, PacksIn, PacksOut1, TailIn, TailOut).

sender(0, [ _ | PacksIn ], PacksOut, TailIn, TailOut) :-
   TailIn := PacksIn, TailOut := PacksOut.
sender(PacksToSend, [ Pack | PacksIn ], PacksOut, TailIn, TailOut) :-
   PacksToSend > 0 |
   PacksOut := [ Pack | PacksOut1], PacksToSend1 is PacksToSend - 1,
   sender(PacksToSend1, PacksIn, PacksOut1, TailIn, TailOut).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% procedures below measure transmission times

% loop([], Tasks, PacksIn, PacksOut, TimeList)
% TimeList may be closed if there is nothing left to do
% otherwise, { PackSize, PacksToSend } is the current task
% such a pack must be generated
% PacksIn and PacksOut are links with the rcv/6, which can be
% regarded as a perpetual process
% after the time has been measured, snd/6 returns tails of
% PacksIn and PacksOut to use them as new communication channels
% in the next loop step
loop([], [], _, _, TimeList) :- TimeList := [].
loop([], [ { PackSize, PacksToSend } | Rest ], PacksIn, PacksOut, TimeList) :-
   gen_pack(PackSize, Pack, Done),
   display_time({ PackSize, PacksToSend, Time }),
```

```
   time_measure(Done, { speed:snd(PacksToSend, Pack, PacksIn,
                              PacksOut, TailIn, TailOut), Time }, Status),
   TimeList := [ { PackSize, PacksToSend, Time } | TimeList1 ],
   loop(Status, Rest, TailIn, TailOut, TimeList1).

display_time({ A, B, Time}) :- tuple(Time) | display_nl({ A, B, Time})!.

% time_call/2 is a STRAND88 system strand call
time_measure([], Task, Done) :- time_call(Task, Done)!.

% a list of orders { StartSize, StepSize, LimitSize, SendTimes }
% is converted into a list of tasks: { PaskSize, Times }
tasks_gen(Orders, Tasks) :- tasks(Orders, Tasks / []).
tasks([], H / T) :- H := T.
tasks([ Order | Rest ], H / T) :- task(Order, H / M), tasks(Rest, M / T).

task({ Start, _, Limit, _}, H / T) :-
  Start > Limit | H := T.
task({ Start, Jump, Limit, Times}, H / T) :- otherwise |
  H := [ { Start, Times } | H1 ],
  Start1 is Start + Jump,
  task({ Start1, Jump, Limit, Times}, H1 / T).

% procedure exported by this module
% it gets orders in the format of list of requests:
% { startSize, StepSize, LimitSize, SendTimes }
% and returns a list of transmission times
%
% computations never terminate, since rcv/6 remains active
%
% WARNING: Orders should start with a dummy request, i.e. { 1, 1, 1, 1},
% because first measure may be not correct (sreason: set-up time)
do_it_all(Orders, TimeList) :-
  rcv(A, B)@1,
  tasks_gen(Orders, Tasks),
  is_list(Tasks, Ground),
  loop(Ground, Tasks, B, A, TimeList)@0.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% processes defined below construct a Pack of Len characters (ASCII 97)
% when the pack is ready, Done becomes an empty list
gen_pack(Len, Pack, Done) :-
  generate_list(Len, List), is_list(List, Ground),
  convert(Ground, List, Pack, Done).

generate_list(0, List) :- List := [].
generate_list(N, List) :- N > 0 |
   N1 is N - 1, List := [ 97 | List1], generate_list(N1, List1).

is_list([ _ | T ], Ground) :- is_list(T, Ground).
is_list([], Ground) :- Ground := [].

convert([], List, Pack, Done) :-
   list_to_string(List, Pack), is_string(Pack, Done).

is_string(Pack, Done) :- string(Pack) | Done := [].
```

## A.6 Linda communication test program

```
/* This program measures mean transmission times in SCA Linda */
/* The measurement is carried out for two kinds of char packs */
/* short (16-1024 bytes, with step 16 bytes)                  */
/* and long ones (1-64 Kbytes, with step 1 Kbyte)             */
/* A pack is sent and received 500 times                      */
/* Network SCA Linda runs each eval() on different processor, */
/* so it must be run on at least three workstations           */

#include <stdio.h>

#define MAX_PACK 1024 * 64   /* maximum pack size */

/* Tables below define experiment parameters for short and long */
/* packets, respectively:                                       */
/*      - first pack size                                       */
/*      - upper limit for pack size                             */
/*      - step size, by which consecutive packets differ        */
/*      - how many times one packet is sent                     */
int start_sizes[2] = { 16, 1024 };
int end_sizes[2]   = { 1024, 1024 * 64 };
int step_sizes[2]  = { 16, 1024 };
int times[2]       = {  500, 500 };

real_main()
  {
    int i, size;
    int sender(), receiver();

    /* for short and then long packets ... */
    for (i = 0; i < 2; i++)
     /* from lower to upper size limit ... */
     for (size = start_sizes[i]; size <= end_sizes[i]; size += step_sizes[i])
      {
       eval(receiver(times[i]));         /* execute receiver       */
       eval(sender(times[i], size));     /* in parallel with sender */
       in(1); /* == "sender   completed" */
       in(2); /* == "receiver completed" */
      }

  }

/* initiates transmission and measures time                   */
/* Because receiver() is evaled before sender() in real_main() */
/* and sender is filling the pack with 'x'-es, receiver() will */
/* be on time to fetch first packet, so no initial handshake   */
/* protocol is needed                                          */
sender(times, size)
 int times, size;
 {
   char pack[MAX_PACK + 1];
   int i;

   for (i = 0; i < size; i++) pack[i] = 'x';
   pack[size] = (char) 0;
```

```
  printf(" sending : %d bytes %d times\n", size, times);
  start_timer();              /* Linda system timer - start it */
  for (; times; times--)
   {
     out("to_R", pack:size);   /* to - receiver                  */
     in("to_S", ?pack:size);   /* from sender                    */
   }
  timer_split(" pack sent");   /* Linda system timer - get time   */
 print_times();               /* Linda system timer - print time */
  return 1;
}


/* receiver() pingpongs packets back to sender                */
receiver(times, size)
 int times, size;
 {
  char pack[MAX_PACK + 1];

  for (; times; times--)
    {
      in("to_R", ?pack:size);
      out("to_S", pack:size);
    }
  return 2;
 }
```

## Appendix B

# Processor farm fractal code programs

## B.1 PVM program

```
/* PVM - Mandelbrot fractal */

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <sys/time.h>

#define max_x 800
#define max_y 600

#define two 2.0
#define four 4.0
#define range 0.25
#define x_coord -0.25
#define y_coord (0.8 + range)
#define gap (range / max_y)

typedef struct results_structure {
   int    start;
   int    nr;
   char   counts[6000];
} RESULTS;

static pack_len = 800;
static model = 0;
static total_pack;

int * processor;
int me;
int Nproc;
int Nwork;

struct timeval tp0,tp1,tp2;
struct timezone tpz;

/*********************************************************************/

double czas(ptime1 , ptime2)
 struct timeval *ptime1;
 struct timeval *ptime2;
 {
   return ((ptime2->tv_sec - ptime1->tv_sec +
           (ptime2->tv_usec - ptime1->tv_usec)/1000000.0)*1000);
 }

/*********************************************************************/

master()
{
  int         snd_pack, rcv_pack;
  char        scr[max_x][max_y];
  int         start;
  int         x,y;
  int         i,j;
  RESULTS     r;
```

```
printf("pack_len = %d\n",pack_len);

pstatus( &Nproc,&i );
if (model==0) Nwork = Nproc;
else Nwork = Nproc-1;

printf("%d processors\n", Nproc);
printf("%d workers   \n", Nwork);
processor = (int*)calloc(Nwork , sizeof(int));

for(i=0; i< Nwork; i++) {
  if((processor[i] = initiate("m_fr","SUN4"))< 0) {
     printf("initiate error %d\n",processor[i]);
     leave();
     exit(1) ;
  } else  printf("%d nr %d OK\n",i,processor[i]);
}
printf("\nSTART\n");

total_pack = max_x*max_y/pack_len;
if ((max_x*max_y)%pack_len!=0) total_pack++;

initsend();
putnint( &pack_len, 1 );
snd( "m_fr",-1 , 99 );

gettimeofday(&tp1,&tpz);
start = 0;
for (i=0; i<Nwork; i++) {
  initsend();
  putnint( &start, 1 );
  snd( "m_fr", processor[i], 100 );
  start += pack_len;
}

snd_pack = Nwork;
rcv_pack = 0;
while (rcv_pack<total_pack) {
  rcv( 200 );
  getbytes(&r,8+pack_len);
  rcv_pack++;
  if (snd_pack<total_pack) {
    initsend();
    putnint( &start, 1 );
    snd( "m_fr", r.nr, 100 );
    start += pack_len;
    snd_pack++;
  }

  x = r.start % max_x;
  y = r.start / max_x;
  if (y<max_y)
  for (j=0; j<pack_len; j++) {
      scr[x][y] = r.counts[j];
      x++; if (x==max_x) { x=0; y++; if (y==max_y) break;}
  }
}
```

```
  gettimeofday(&tp2,&tpz);
  for (i=0; i<Nwork; i++) terminate("m_fr",processor[i]);
  cfree(processor);
}

/*******************************************************************/

worker()
{
  RESULTS  r;
  float ac, bc, size, a2, b2, a, b;
  int i, j, n, x, y, count;
  int ready;
  int start;

  rcv( 99 );
  getnint(&pack_len,1);

  for (;;) {
    rcv( 100 );
    getnint(&start,1);
    x = start % max_x;
    y = start / max_x;

    for (i = 0; i < pack_len; i++)
    {
      bc = y_coord - y*gap;
      ac = x*gap + x_coord;
      a = ac; b = bc; size = 0.0; count = 1;
      a2 = a*a; b2 = b*b;
      while ((size < four) && (count < 256))
      {
        b = two*a*b + bc;
        a = a2 - b2 + ac;
        a2 = a*a; b2 = b*b;
        size = a2 + b2;
        count++;
      }
      r.counts[i] = (char)(count - 1);
      x++; if (x==max_x) { x=0; y++; }
    }
    r.start = start;
    r.nr = me;
    initsend();
    putbytes(&r,8+pack_len);
    snd("m_fr",0,200);
  }
}

/*******************************************************************/

main ( argc, argv )
int argc;
char *argv[];
{
  if (argc>=2) model = atoi(argv[1]);
  if (argc==3) pack_len = atoi(argv[2]);
```

```
  gettimeofday(&tp0,&tpz);
  me = enroll( "m_fr");
  if (me<0) exit(0);

  if (me==0) master(); else worker();

  if (me==0) {
    printf("\nTime t = %8.2f [sec]",czas(&tp1,&tp2)/1000.0);
    printf("\nFINISH\n");
    leave();
    exit(0);
  }
}
```

## B.2 P-4 program

```
/* P4 - Mandelbrot fractal */

#include "p4.h"

#include <stdio.h>
#include <malloc.h>
#include <sys/time.h>

#define max_x 800
#define max_y 600

#define two 2.0
#define four 4.0
#define range 0.25
#define x_coord -0.25
#define y_coord (0.8 + range)
#define gap (range / max_y)

typedef struct results_structure {
   int    start;
   int    nr;
   char   counts[6000];
} RESULTS;

#define END 1

static pack_len = 800;
static total_pack;
unsigned char scr[max_x][max_y];

int me;
int Nproc;
struct timeval tp1,tp2,tp3;
struct timezone tpz;

int size, type, from;
char *incoming;

/**********************************************************************/

double czas(ptime1 , ptime2)
 struct timeval *ptime1;
 struct timeval *ptime2;
 {
   return ((ptime2->tv_sec - ptime1->tv_sec +
           (ptime2->tv_usec - ptime1->tv_usec)/1000000.0));
 }

/**********************************************************************/

main ( argc, argv )
int argc;
char *argv[];
{
    int flops, netbytes;
```

```
    int x,y;

    if (argc==2) pack_len = atoi(argv[1]);

    gettimeofday(&tp1,&tpz);

    p4_initenv(&argc, argv);
    me = p4_get_my_id();
    if (me == 0) master(); else slave();

    gettimeofday(&tp3,&tpz);

    if (me==0) {
        printf("init \t %6.2f \n", czas(&tp1, &tp2));
     printf("work \t %6.2f \n", czas(&tp2,&tp3));
     printf("total \t %6.2f\n", czas(&tp1,&tp3));

     printf("\nFINISH\n");

    }
    p4_wait_for_end();
}

/*********************************************************************/

master()
{
  int         snd_pack, rcv_pack;
  int         start;
  int         x,y;
  int         i,j;
  RESULTS     *r;

  p4_create_procgroup();
  printf("pack_len = %d\n",pack_len);
  Nproc = p4_num_total_slaves();

  printf("\nSTART\n");
  gettimeofday(&tp2,&tpz);

  total_pack = max_x*max_y/pack_len;
  if ((max_x*max_y)%pack_len!=0) total_pack++;

  start = 0;
  for (i=0; i<Nproc; i++) {
    p4_sendx(99, i+1, &pack_len, sizeof(int), P4INT);
    p4_sendx(100, i+1, &start, sizeof(int), P4INT);
    start += pack_len;
  }

  snd_pack = Nproc;
  rcv_pack = 0;
  while (rcv_pack<total_pack) {
    type = 200;
    from = -1;
    incoming = NULL;
    p4_recv(&type, &from, &incoming, &size);
    r = (RESULTS *)incoming;
```

```
    rcv_pack++;
    if (snd_pack<total_pack) {
      p4_send(100, (*r).nr, &start, sizeof(int));
      start += pack_len;
      snd_pack++;
    }

    x = (*r).start % max_x;
    y = (*r).start / max_x;
    if (y<max_y)
    for (j=0; j<pack_len; j++) {
        scr[x][y] = (*r).counts[j];
        x++; if (x==max_x) { x=0; y++; if (y==max_y) break;}
    }
    p4_msg_free(incoming);
  }
  p4_broadcast(END, &start, sizeof(start));
}

/**********************************************************************/

slave()
{
  RESULTS  r;
  float ac, bc, size, a2, b2, a, b;
  int i, j, n, x, y, count, len;
  int ready;
  int start;
  char *incoming;

  type = 99;
  from = -1;
  incoming = NULL;
  p4_recv(&type, &from, &incoming, &len);
  pack_len = *((int *) incoming);
  p4_msg_free(incoming);

  for (;;) {
    type = -1;
    from = 0;
    incoming = NULL;
    p4_recv(&type, &from, &incoming, &len);
    if (type == END) return;
    start = *((int *) incoming);
    p4_msg_free(incoming);

    x = start % max_x;
    y = start / max_x;

    for (i = 0; i < pack_len; i++)
    {
      bc = y_coord - y*gap;
      ac = x*gap + x_coord;
      a = ac; b = bc; size = 0.0; count = 1;
      a2 = a*a; b2 = b*b;
      while ((size < four) && (count < 256))
      {
        b = two*a*b + bc;
```

```
        a = a2 - b2 + ac;
        a2 = a*a; b2 = b*b;
        size = a2 + b2;
        count++;
      }
      r.counts[i] = (char)(count - 1);
      x++; if (x==max_x) { x=0; y++; }
    }
    r.start = start;
    r.nr = me;
    p4_send(200, 0, &r, 8+pack_len);
  }
}
```

## B.3 SR program

```
# Processors farm (2D version) for mandelbrot computation
# Uses SR compiler version 2.0


resource glob

  const HOST_NAME_LEN := 50

  type farm_info = rec
   (
    xmin, ymin, xmax, ymax: int       # farm boundaries
    cxcell, cycell: int               # cell dimentions

    cxmin, cymin, cxmax, cymax: real  #!!! Mandelbrot set boundaries
    iter: int                         #!!! number of iterations
   )

  #!!! result type:
  type   res_type = char

  optype outproc = (machine, x, y, cx, xy: int; buff: [*]res_type)

end glob




resource worker

  import glob

  op init(farm: farm_info)
  op docell(x, y, cx, cy: int)

  external setup(f: farm_info)
  external calc(x, y, cx, cy: int; res buff: [*]res_type)

body worker(machine: int; o: cap outproc)

  var farm: farm_info
  var x, y, cx, cy: int
  process worker_process
    receive init(farm)  # receive farm information

    setup(farm)
    var buff[farm.cxcell * farm.cycell]: res_type # allocate buffer

    do true ->
      receive docell(x, y, cx, cy)        # wait for job
      calc(x, y, cx, cy, buff)            # calculate field
      send o(machine, x, y, cx, cy, buff) # send results
    od
  end worker_process

end worker
```

```
resource main() # farmer

  import glob
  import worker

  const MAXREM := 64  # maximal number of virtual machines
  const AHEAD  := 0

  # import some useful C functions:
  external atoi(s: string[*]) returns i: int
  external atof(s: string[*]) returns f: real
  external run_info(filename: string[*]; comment: string[*]; action: int)

  op startup()              # starting semaphore, every process (except initial)
                            # waits for it
  op start(machine: int)    # starts virtual machine
  op hungry(machine: int)   # signal that machine has nothig to do
  op out: outproc           # output procedure (receiving results)
  op command_line_parse(i: string[*]) returns r: int
                            # parses command line arguments

  var rcap[MAXREM]: cap worker      # caps for worker resources
  var name[MAXREM]: string[HOST_NAME_LEN]  # their host names
  var tally[MAXREM]: int            # number of cells computed by each
  var nvm: int                      # number of virtual machines
  var xcells, ycells: int           # number of cells in x and y directions
  var farm: farm_info               # farm information
  var cml: string[256]              # copy of command line
  var silent: bool := false         # silent mode off


proc start(machine)
  var o: cap outproc := out
  var mcap: cap vm

  tally[machine] := 0               # zero the output counter
  locate(machine, name[machine])    # set hostname
  mcap := create vm() on machine    # create virtual machine
  if mcap = null ->                 # check for success
    if (not silent) ->
      write(stderr, "Can't create vm on", name[machine])
    fi
    stop
  fi

  rcap[machine] := create worker(machine, o) on mcap  # create worker server process
  send rcap[machine].init(farm)            # initialize it & send maximal field size
  if (not silent) ->                       # and announce it
    write(stderr, "ready on", name[machine])
    flush(stderr)
  fi
  fa j := 0 to AHEAD ->
    send hungry(machine)                             # give it an initial work queue
  af
end start
```

```
process initialization

 var argument: string[HOST_NAME_LEN]
 var sAHEAD: string[20]

  #!!! default values
  farm.xmin   := 0
  farm.ymin   := 0
  farm.xmax   := 799
  farm.ymax   := 599
  farm.cxcell := 800
  farm.cycell := 1
  farm.cxmin  := -0.25
  farm.cxmax  := 0.08
  farm.cymin  := 0.8
  farm.cymax  := 1.05
  farm.iter   := 256

  # read host names
  nvm := 0
  cml := ""
  fa i := 1 to numargs() ->
    getarg(i, argument)
    cml := cml || " " || argument

    if command_line_parse(argument) = 0 ->   # argument isn't a switch,
                                              # so treat it as a host name
      nvm++
      name[nvm] := argument
    fi
  af

  # if no hosts specified in command line we terminate
  if nvm = 0 ->
    if (not silent) ->
      write (stderr, "Machine not specified")
    fi
    stop
  fi

  # calculate number of cells:
  if (farm.xmax - farm.xmin + 1)%farm.cxcell = 0 -> xcells := (farm.xmax - farm.xmin + 1)/farm.cxcell
  [] else -> xcells := (farm.xmax - farm.xmin + 1)/farm.cxcell + 1
  fi

  if (farm.ymax - farm.ymin + 1)%farm.cycell = 0 -> ycells := (farm.ymax - farm.ymin + 1)/farm.cycell
  [] else -> ycells := (farm.ymax - farm.ymin + 1)/farm.cycell + 1
  fi

  run_info("farm.log", cml, 0)
  sprintf(sAHEAD, "AHEAD=%d\n", AHEAD);
  run_info("", sAHEAD, 1)
  # create virtual farmer processes
  fa i := 1 to nvm ->
    send start(i)
  af
```

```
  if (not silent) -> write("Global initialization OK...")
  fi

  # open output file

  # open semaphores (allow dispatcher and output processes to work)
  send startup()
  send startup()
end initialization



process dispatcher
receive startup()

  var machine: int
  var x, y, cx, cy: int

  fa i := 0 to xcells -1 ->
    fa j := 0 to ycells -1 ->
      # calculate cell origin
      x := farm.cxcell*i
      y := farm.cycell*j

      # calculate cell dimentions
      if x + farm.cxcell > farm.xmax + 1 -> cx := farm.xmax + 1 - x
      [] else -> cx := farm.cxcell
      fi

      if y + farm.cycell > farm.ymax + 1 -> cy := farm.ymax + 1 - y
      [] else -> cy := farm.cycell
      fi

      receive hungry(machine)                  # get idle machine
      send rcap[machine].docell(x, y, cx, cy)  # give it some job
    af
  af
  do true ->
    receive hungry(machine)  # if all calculation is done, ignore all requests
  od
end dispatcher


process output
receive startup()

  var machine, x, y, cx, cy, i, j, k: int
  var buff[farm.cxcell*farm.cycell]: res_type
  var mbrotpic[(farm.xmax - farm.xmin + 1) * (farm.ymax - farm.ymin + 1)]: res_type

  # receive calculated data
  fa k := 1 to xcells*ycells ->
    receive out(machine, x, y, cx, cy, buff)
    tally[machine]++
    hungry(machine)
    fa j := 0 to cy - 1 ->
      fa i := 0 to cx - 1 ->
        mbrotpic[(j + y)*(farm.xmax - farm.xmin + 1) + i + x + 1] := buff[j*cx + i + 1]
```

```
      af
    af
  af

  # summary
  fa i := 1 to nvm ->
    if (not silent) ->
      write(stderr, "Machine ", name[i], ": ", tally[i])
    fi
  af
  run_info("", "end of data collecting", 1)
  run_info("", "", 2)
  stop
end output


proc command_line_parse(s) returns r
  if s[1:1] != "-" ->
    r := 0
  [] else ->
    #!!! command line switches:
    if length(s) > 1 and s[2:2] = "i" ->
      # number of iterations
      farm.iter := atoi(s[3:*])

    [] length(s) > 1 and s[2:2] = "l" ->
      # left M-set bound
      farm.cxmin := atof(s[3:*])

    [] length(s) > 1 and s[2:2] = "s" ->
      # silent mode on
      silent := true

    [] length(s) > 1 and s[2:2] = "r" ->
      # right M-set bound
      farm.cxmax := atof(s[3:*])

    [] length(s) > 1 and s[2:2] = "t" ->
      # top M-set bound
      farm.cymax := atof(s[3:*])

    [] length(s) > 1 and s[2:2] = "b" ->
      # bottom M-set bound
      farm.cymin := atof(s[3:*])

    [] length(s) > 1 and s[2:2] = "x" ->
      # x size
      farm.xmax := farm.xmin + atoi(s[3:*]) - 1

    [] length(s) > 1 and s[2:2] = "y" ->
      # left M-set bound
      farm.ymax:= farm.ymin + atoi(s[3:*]) - 1

    [] length(s) > 2 and s[2:3] = "cx" ->
      # x size
      farm.cxcell := atoi(s[4:*])

    [] length(s) > 2 and s[2:3] = "cy" ->
```

```
      # left M-set bound
      farm.cycell := atoi(s[4:*])

    [] length(s) > 1 and s[2:2] = "h" ->
      # help info
      write("usage: a.out [switches] [hosts]")
      write(" switches:")
      write("  -i<number of iterations>")
      write("  -l<left M-set bound>")
      write("  -r<right M-set bound>")
      write("  -l<left M-set bound>")
      write("  -t<top M-set bound>")
      write("  -b<bottom M-set bound>")
      write("  -x<x resolution>")
      write("  -y<y resolution>")
      write("  -cx<x field size>")
      write("  -cy<y field size>")
      write("  -s   turn silent mode on")
      write("  -h   this help")
      write(" Example:")
      write("  a.out -l-2.5 -r0.5 -t1.5 -b-1.5 -x256 -y256 -cx16 -cy16 -i64 uci1 uci2 uci3")
      stop

    [] else ->
      write("Unknown switch: ", s)
    fi
    r := 1
  fi
end command_line_parse
end main


#include <stdio.h>

typedef double real;   /* SAME AS IN SR MODULE */
typedef char res_type; /* SAME AS IN SR MODULE */

real Starta, Startb;   /* START POINT OF FARM */
real Sizea, Sizeb;     /* SIZE OF FARM */
int Iter;              /* NUMBER OF ITERRATIONS */

typedef struct
         {
          int xmin, ymin, xmax, ymax;
          int cxcell, cycell;
          real cxmin, cymin, cxmax, cymax;
          int iter;
         } farm_info; /* SAME AS IN SR MODULE */

setup(f)
farm_info f;
/*---------------------------------------------------------------------------*/
/* READS GLOBAL FARM PARAMETERS                                              */
/*---------------------------------------------------------------------------*/
 {
  static int done = 0;

  if (done) return; /* PREVENT MULTIPLE CALL */
  done = 1;
```

```
  Starta = f.cxmin;
  Startb = f.cymin;
  Sizea  = (f.cxmax - f.cxmin) / (f.xmax - f.xmin + 1);
  Sizeb  = (f.cymax - f.cymin) / (f.ymax - f.ymin + 1);
  Iter   = f.iter;
 }


calc(x, y, cx, cy, buff)
int x, y, cx, cy;
res_type buff[];
/*------------------------------------------------------------------------*/
/* CALCULATES FARM IN RECTANGLE (x..x + cx - 1, y..y + cy - 1)            */
/* STORES RESULTS IN buff[0..cx*cy - 1]                                   */
/*------------------------------------------------------------------------*/
 {
  int i, j, count, fill = 0;
  real ac, bc, a, b, na, a2, b2;
/*
  printf("calculating box: (%f - %f, %f - %f)\n", Starta + x*Sizea,
                                                  Starta + (x + cx - 1)*Sizea,
                                                  Startb + y*Sizeb,
                                                  Startb + (y + cy - 1)*Sizeb);
  fflush(stdout);
*/
  for (j = 0; j < cy; j++)
   {
    bc = Startb + (y + j)*Sizeb;
    for (i = 0; i < cx; i++)
     {
      ac = Starta + (x + i)*Sizea;
      count = 1;
      a2 = a = 0;
      b2 = b = 0;
      while (count < Iter)
       {
        na = a2 - b2 + ac;
        b  = 2*a*b     + bc;
        a = na;
        a2 = a*a;
        b2 = b*b;
        if (a2 + b2 > 4) break;
        count++;
       }
      buff[fill++] = (res_type)(count - 1);
/* printf("%d ", buff[fill - 1]);
fflush(stdout); */
     }
   }
 }


#include <stdio.h>
#include <sys/time.h>


run_info(filename, comment, action)
char *filename, *comment; int action;
```

```
/*--------------------------------------------------------------------------*/
/* WRITES LOG-INFO ON FILE filename, LOG-INFO CONTAINS ANY comment AND TIME  */
/* OF run_info(...) CALL.                                                   */
/*   IF action == 0 TIMER IS RESET,                                         */
/*   IF action == 1 TIMER IS ONLY READ                                      */
/*   IF action == 2 LOG-FILE IS CLOSED                                      */
/* WHEN run_info IS CALLED FIRST (OR AFTER CALL WITH action == 2)           */
/* FILE filename IS OPENED FOR APPENDING, IN OTHER CASES filename IS IGNORED */
/* AND THUS SHOULD BE SET TO NULL.                                          */
/*--------------------------------------------------------------------------*/
 {
  static FILE *fptr = NULL;
  static struct timeval t1, t2;
  struct timezone tz;
  double time;

  if (comment == NULL) comment = "";

  if (fptr == NULL)
   {
    fptr = fopen(filename, "a");
    t1.tv_sec = 0;
   }

  if (fptr == NULL) return;

  switch (action)
   {
    case 0: /* INIT TIMER */
         if (!t1.tv_sec)
          {
           gettimeofday(&t1, &tz);
           fprintf(fptr, "\nTIMER INIT: %s\n", comment);
          }
         else
          {
           gettimeofday(&t2, &tz);
   time = t2.tv_sec - t1.tv_sec + (t2.tv_usec - t1.tv_usec)/1000000.0;
           fprintf(fptr, "TIMER INIT - after %f seconds: %s\n",
                   time, comment);
           t1 = t2;
          }
         break;
    case 1: /* READ TIMER */
         gettimeofday(&t2, &tz);
 time = t2.tv_sec - t1.tv_sec + (t2.tv_usec - t1.tv_usec)/1000000.0;
         fprintf(fptr, "TIMER READ - after %f seconds: %s\n",
                 time, comment);
         break;
    case 2: /* CLOSE TIMER */
         fclose(fptr);
         fptr == NULL;
         break;
   }
 }
```

## B.4 ANSA program

```
Farmer : INTERFACE =

BEGIN

-- Operations

Start : OPERATION [length : INTEGER ; size : INTEGER]
      RETURNS [];

END.


Worker : INTERFACE =

BEGIN

--      Types

        Buffer : TYPE = SEQUENCE OF CHAR;
        HostN  : TYPE = ARRAY 10 OF CHAR;
Data : TYPE = RECORD [
start : INTEGER,
             pack_len : INTEGER
];

Results : TYPE = RECORD [
start : INTEGER,
 name : HostN,
               counts : Buffer
];

-- Operations

Process : OPERATION [ Number : Data ] RETURNS [ Results ];

END.

! USE Farmer
! USE Worker
! DECLARE {lr} : Farmer SERVER
! DECLARE {fs->worker} : Worker CLIENT

ansa_InterfaceRef lr;


#define DATA 0
#define RES 1
#define max_x 800
#define max_y 600

/*int pack_len = 800;*/
int total_pack;
char scr[max_x][max_y];

#include <capsule/ecs.h>
#include <./bcompat.h>
```

```
typedef struct{
ansa_InterfaceRef worker;
} farmerState;

ansa_StatePtr Farmer__Create( worker )
ansa_InterfaceRef worker;
{
farmerState *p;

p = (farmerState *)malloc(sizeof(farmerState));
        if (p==NULL)
         {
          fprintf(stderr,"blad alokacji");
          exit(4);
         }
    p->worker = worker;
return (ansa_StatePtr)p;
}

void Farmer__Destroy(state)
ansa_StatePtr state;
{
 free((char *)state);
}

ansa_InterfaceRef create_farmer(worker)
ansa_InterfaceRef worker;
{
! {lr} :: Farmer$Create(1,worker)
 return lr;
}




extern ansa_EventCount ec[2];
extern ansa_Sequencer sq[2];

void ecs_await();

static void grab_mutex(which)
int which;
{
 ecs_await(ec[which],ecs_ticket(sq[which]));
}

static void free_mutex(which)
int which;
{
 ecs_advance(ec[which]);
}

/*
  Data reading and Results writing routines
*/
```

```
#include "io.c"

static int get(buf,len)
Data *buf;
int len;
{
 int success;
 grab_mutex(DATA);
 success=get_data(buf,len);
 free_mutex(DATA);
 return success;
}

static void put(buf,len)
Results *buf;
int len;
{
 grab_mutex(RES);
 put_data(buf,len);
 free_mutex(RES);
}


int Farmer_Start(_attr,pack_len,BUF_SIZE)
ansa_InterfaceAttr *_attr;
ansa_Integer pack_len;
ansa_Integer BUF_SIZE;
{
 Data buf;
 Results res;
 int in_processing=0,k=0;
 ansa_Voucher a[4];

 farmerState *fs=(farmerState*)stub_state_locate(_attr->attr_dst_ref);


 total_pack = max_x*max_y/pack_len;   /* ilosc wszystkich paczek */

 if ((max_x*max_y)%pack_len!=0) total_pack++;
 while(1){
  int result;
  if(in_processing<BUF_SIZE){
   result=get(&buf,pack_len);
   if(!result)
    break;
! {a[k]} := fs->worker$Process(buf)
   k++;
   in_processing++;
  }
  else{
if(BUF_SIZE==1)
   k=0;
else
   k%=BUF_SIZE;
!{res} <- fs->worker$Redeem(a[k]) Continue transmitTimeout Continue -1784913919
   in_processing--;
   put(&res,pack_len);
  }
```

```
 }
 while(in_processing>0){
   k++;
if(BUF_SIZE==1)
   k=0;
else
   k%=BUF_SIZE;
!{res} <- fs->worker$Redeem(a[k]) Continue transmitTimeout Continue -1784913919
  in_processing--;
  put(&res,pack_len);
 }
}


/* Header Start */
/*
 *  File: client.dpl
 *  Author:
 *  Date Created:
 *
 *  General Description:
 *   Processor farm creator and activator
 */
/* Header End */

/*#include <stdio.h> */
#include </home/tulip/ansa/install/sun4_sos_4.1/include/capsule/ecs.h>
#include <string.h>

! USE Trader
! USE Worker
! DECLARE { *ir } : Worker CLIENT
! USE Farmer
! DECLARE { *fr } : Farmer CLIENT

 char *hosttab[]={
 "tulip",
 "rose",
 "daisy",
 "crocus",
 NULL
};

int host_nb;

ansa_InterfaceRef workers[21];
ansa_InterfaceRef farmers[21];
ansa_Voucher      fvouchers[21];
ansa_EventCount ec[2];
ansa_Sequencer sq[2];

double czas(ptime1 , ptime2)
 struct timeval *ptime1;
 struct timeval *ptime2;
 {
   return ((ptime2->tv_sec - ptime1->tv_sec +
           (ptime2->tv_usec - ptime1->tv_usec)/1000000.0)*1000);
 }
```

```
void body(argc, argv)
int argc;
char *argv[];
{
  ansa_InterfaceRef *ir,*fr;
  ansa_Voucher *fv;
  char **p;
  int i,p_len,buf,task_nb;
  struct timeval tp1,tp2;
  struct timezone tpz;
  FILE *fp;


  ansa_InterfaceRef create_farmer();

  if (argc<5)
   {
    printf(" usage: client  pack_len BUF_SIZE task_nb host_nb \n");
    exit(1);
   }
  else
   {
    p_len=atoi(argv[1]);
    buf= atoi(argv[2]);
    task_nb= atoi(argv[3]);
    host_nb= atoi(argv[4]);

    if (p_len>800*600 || p_len<1)
     {
      printf("zla dlugosc pakietu ;liczba spoza przedzialu <1,240000> \n");
      exit(1);
     }
    else
     printf("dlugosc pakietu = %d \n",p_len);
   }



  ec[0]=ecs_makeEventCount((ansa_Cardinal)0);
  sq[0]=ecs_makeSequencer((ansa_Cardinal)0);
  ec[1]=ecs_makeEventCount((ansa_Cardinal)0);
  sq[1]=ecs_makeSequencer((ansa_Cardinal)0);
  nucleus_tasks(task_nb-1,0);
  gettimeofday(&tp1,&tpz); /* poczatek pomiaru czasu */

  for(i=0,p=hosttab,ir=workers,fr=farmers;i<host_nb;i++,p++,ir++,fr++){
   char props[100];
   sprintf(props,"Node == '%s'",*p);
! {*ir} <- traderRef$Import("Worker", "/ansa/testservices", props )
   *fr=create_farmer(*ir);
  }

  for(i=0,fv=fvouchers,fr=farmers;i<host_nb;i++,fv++,fr++){
!  {*fv} := *fr$Start(p_len,buf)
 /*  printf("Farmer %s started\n",hosttab[i]); */
  }
```

```
  for(i=0,fv=fvouchers,fr=farmers;i<host_nb;i++,fv++,fr++){
!   {} <- *fr$Redeem(*fv) Continue  -1784913919 Continue abnormalReturn
  }

  gettimeofday(&tp2,&tpz);
  printf("\nCzas obliczen t = %8.2f [sec]\n",czas(&tp1,&tp2)/1000.0);
  fp = fopen("result.fr","a");
  fprintf(fp,"Proc %2d  t=%d w=%d  Pack_len %4d  Time%8.2f\n",host_nb,task_nb,buf,p_len,czas(&tp1,&tp2)/1000.0 );
  fclose( fp );

  finish();


  for(i=0,p=hosttab,fr=farmers;i<host_nb;i++,p++,fr++)
  {
!  {} ::    Farmer$Destroy(*fr)
   }

}


/* Header Start */
/*
 *  Copyright 1989 APM Limited (ISA Project)
 *
 *  File: server.dpl
 *  Author:
 *  Date Created:
 *
 *  General Description:
 *    Server for the Checksum service.
 *
 *  $Header: /disc2/local/src/ANSAware3.0Dist/master/src/examples/Cksum/RCS/server.dpl,v 1.11 90/11/05 15:35:50 njh Exp $
 *
 */

/* Header End */

#include <AWstdio.h>

#define PROPSIZE 1024
char propbuf[PROPSIZE];

! MANAGED Worker

! USE Worker
! USE Trader
! DECLARE { ir } : Worker SERVER

ansa_InterfaceRef ir;

void body (argc , argv, envp)
 int argc;
 char *argv[], *envp[];
{
  ansa_InterfaceRef ir;

! {ir}  ::  Worker$Create(16)
```

```
system_init_properties(propbuf, PROPSIZE, argc, argv);
! {} <- traderRef$Export("Worker", "/ansa/testservices",propbuf,ir)
}


ansa_StatePtr Create_Worker_Object(argc, argv, envp, results)
int argc;
char *argv[], *envp[];
InstantiateResult *results;
{

        nucleus_tasks(2,0);
! {ir} :: Worker$Create(4)


results->length = 1;
results->data = &ir;
return (ansa_StatePtr)NULL;
}

ansa_Boolean Destroy_Worker_Object(state)
ansa_StatePtr state;
{
Result regres;
!       {} :: Worker$Destroy(ir)
free((char *)state);
}

#include "work.c"

int Worker_Process( _attr, data, result)
ansa_InterfaceAttr *_attr;
Data data;
Results *result;
{
/* FILE *fp;

  fp=fopen("res.dat","a");
  fprintf(fp,"%d ",data.number);
  fclose(fp);
*/
  work(data,result);
  return 1;
}

#define max_x 800
#define max_y 600

#define two 2.0
#define four 4.0
#define range 0.25
#define x_coord -0.25
#define y_coord (0.8 + range)
#define gap (range / max_y)


void work(data,result)
Data data;
```

```
Results *result;
{

  float ac, bc, size, a2, b2, a, b;
  int i, j, x, y, count;
  char *t;

    t=malloc(data.pack_len*sizeof(char));
    x = data.start % max_x;
    y = data.start / max_x;
       for (i = 0; i < data.pack_len; i++)
    {
      bc = y_coord - y*gap;
      ac = x*gap + x_coord;
      a = ac; b = bc; size = 0.0; count = 1;
      a2 = a*a; b2 = b*b;
      while ((size < four) && (count < 256))
      {
        b = two*a*b + bc;
        a = a2 - b2 + ac;
        a2 = a*a; b2 = b*b;
        size = a2 + b2;
        count++;
      }
        t[i] = (char)(count - 1);

          x++; if (x==max_x) { x=0; y++; }
    }
    result->counts.length=data.pack_len;
    result->counts.data=&(t[0]);
    result->start = data.start;
    gethostname(result->name,10) ;

  }
```

## B.5 Strand program

```
% MANDELBROT FRACTAL COMPUTING - MODULE: farmer.std
% This module supervises the work of workers.

-compile(free).
-exports([farmer/3]).

farmer(Stream, Limit, Result) :-  farmer(Stream, 0, Limit, Result).

% farmer/4:
%   1> communication stream is closed - the work has been completed
%   2> a worker wants a new task - but there is none
%   3> a worker wants a new task and he gets it; the task consists
%      of pack number and return argument - ColourPack

farmer([], Limit, Limit, Result) :-  Result := [].
farmer([ request(Task) | Stream ], Limit, Limit, Result) :-
  Task := done,
  farmer(Stream, Limit, Limit, Result).
farmer([ request(Task) | Stream ], PackNum, Limit, Result) :-
  PackNum < Limit,
  Task := { PackNum, ColourPack },
  PackNum1 is PackNum + 1,
  Result := [ ColourPack | Result1 ],
  farmer(Stream, PackNum1, Limit, Result1).



% MANDELBROT FRACTAL COMPUTING - MODULE: manager.std
% This module distributes workers and sets up the communication
% channels between them and the farmer.

-compile(free).
-exports([go/2]).

% go/2 - computes the optimal placement of workers in the network,
%        initiates a farmer and the workers, sets up the communication
%        channels between the farmer and the workers
go(PackSize, Result) :-
   compute_limit(PackSize, Limit),  % compute packs number (Limit)
   topology(Top)!,                  % topology (connected)
   get_workers_num(Top, Limit, Start, WorkersNum),
   farmer:farmer(Stream, Limit, Result),
   merger(InStream, Stream),        % communication farmer<--->workers
   map_workers(Start, WorkersNum, InStream, PackSize).

map_workers(Worker, WorkersNum, InStream, _) :-
    Worker > WorkersNum | InStream := [].
map_workers(Worker, WorkersNum, InStream, PackSize) :-
    otherwise |
    InStream := [ merge(NewStream) | InStream1 ],
    NewStream := [ request(Task) | NewStream1 ],
    Call := { ":", { worker, worker(0, Task, NewStream1, PackSize) }},
    Worker1 is Worker + 1,
    send({ '@', { Call, Worker } }),
    map_workers(Worker1, WorkersNum, InStream1, PackSize).

% Start == a number of the processor from which placing of workers is started.
```

```
% The idea is as follows - if the number of packs is less than the number
% of processors, there is no worker process on the farmer's processor.
% Otherwise, there is one worker on every processor in the network
min(X, Y, WN, S) :- Y >= X | WN := X, S := 0.
min(_, Y, WN, S) :- otherwise | WN := Y, S := 1.

% Limit == number of packs to be computed
% Limit == (480000 / PackSize ) + !! (480000 % PackSize ) (C language notation)
compute_limit(PackSize, Limit) :-
    Mod is 480000 // PackSize,
    Div is 480000 / PackSize,
    compute(Div, Mod, Limit).

compute(Div, 0, Limit) :- Limit := Div.
compute(Div, Mod, Limit) :- Mod > 0 | Limit is Div + 1.



% MANDELBROT FRACTAL COMPUTING - MODULE: worker.std
% This module carries out orders received from farmer.
% This module calls the foreign procedure, c_worker/4.

-compile(free).
-exports([worker/4]).

% worker/4:
%   1> the work has been finished
%   2> PackNum is the number of a pack to be computed - the result is
%      put into ColourPack
%
%   c_worker/4 is a foreign procedure (written in C), incorporated
%              into Strand Abstract Machine (SAM). This procedure
%              does the actual computing of colours. On completion,
%              Done is assigned an integer constant 0.
worker(0, done, InStream, _) :- InStream := [].
worker(0, { PackNum, ColourPack }, InStream, PackSize) :-
   c_worker(PackNum, PackSize, ColourPack, Done),
   InStream := [ request(Task) | InStream1],
   worker(Done, Task, InStream1, PackSize).



/* MANDELBROT FRACTAL - C procedure added to Strand */
#include <stdio.h>

/*  Fractal's Conatsnts */
#define range 0.25
#define two   2.0
#define four  4.0
#define x_coord -0.25
#define y_coord (0.8 + range)
#define gap (range / max_y)
#define max_x 800
#define max_y 600

/* pack_num - a number of the pack      (INPUT ARGUMENT)
   pack_len - length of the pack        (INPUT ARGUMENT)
   result - colours of pack's points    (OUTPUT ARGUMENT)
```

```
   done - synchronizing zero            (OUTPUT ARGUMENT)
*/
 void c_worker(pack_num, pack_len, result, done)
  int pack_num, pack_len, * done;
  char ** result;

{    float ac, bc, size, a2, b2, a, b;
     int x, y, count, points_so_far, i = 0;

     static char * temp;

       /* computing of coordinates of first point of the current pack */
     points_so_far = pack_num * pack_len;
     x = points_so_far % max_x;
     y = points_so_far / max_x;

       /* last pack may be shorter than the others */
     if ( max_x * max_y - points_so_far < pack_len)
       pack_len = max_x * max_y - points_so_far;

       /* result will be stored here */
     if (temp) free(temp);
     temp = (char *) malloc( (pack_len + 1) * sizeof(char));

     temp[pack_len] = (char) 0;

       /* colours computing */

     bc = y_coord - y * gap;
     while (pack_len--)
       {
        ac = x * gap + x_coord;
        a = ac; b = bc; size = 0.0; count = 1;
        a2 = a*a; b2 = b*b;
        while ((size < four) && (count < 256))
          {
           b = two*a*b + bc;
           a = a2 - b2 + ac;
           a2 = a*a; b2 = b*b;
           size = a2 + b2;
           count++;
          }
        x++;
        if (x == max_x) { x = 0; y++; bc = y_coord - y * gap; }
        temp[i++] = (char) ( count - 1 );

       }
        * result = temp;
        * done = 0;
 }
```

## B.6 Linda program

```
/* SCA Linda - Mandelbrot fractal */

/* task format: ((int)line)                           */
/* answer format : ((int)line, (char *) answer)       */

#include <stdio.h>
#define two 2.0
#define four 4.0
#define range 0.25
#define x_coord -0.25
#define y_coord (0.8 + range)
#define gap (range / max_y)
#define max_x 800
#define max_y 600

int y;
char arr[max_y][max_x];


 real_main(argc, argv)
  int argc;
  char  * argv[];
 {
    int i, worker(), workers, worker_on_farmer;

   if (argc != 3)
    {
     printf(" usage: %s <workers_off_farmer> <worker_on_farmer>\n", argv[0]);
     return 0;
   }

  workers = atoi(argv[1]);
  worker_on_farmer = atoi(argv[2]);

  printf(" workers == %d, on farmer == %d\n", workers, worker_on_farmer);

   start_timer();
    for (i = 0; i < max_y; i++) out(i);
    for (i = 0; i < workers; i++) eval("worker", worker());

  if (worker_on_farmer) worker();

/* the scheme for collecting results presented below is not very        */
/* efficient, but suited very well for the case when farmer's processor */
/* runs 'worker()' process before gathering results                     */
    for (y = 0; y < max_y; y++) in(y, ? arr[y]);
   timer_split(" done");

   print_times();

   return 0;
  }
```

```
int worker()
{
   float ac, bc, size, a2, b2, a, b;
   int x, y, count;
   char array[max_x];

do {
     if (!inp(? y)) return 0;

     bc = y_coord - y*gap;
     for (x = 0; x < max_x; x++)
      {
        ac = x*gap + x_coord;
        a = ac; b = bc; size = 0.0; count = 1;
        a2 = a*a; b2 = b*b;
        while ((size < four) && (count < 256))
          {
           b = two*a*b + bc;
           a = a2 - b2 + ac;
           a2 = a*a; b2 = b*b;
           size = a2 + b2;
           count++;
          }
        if (count == 256) count--;
        array[x] = count;
     }
     out(y, array);


  } while (1);
}
```

# Appendix C

# Transfer data blocks results

- PVM

| Block length [B] | S2-S2 | S2-SLC | SLC-SLC | RS-RS | Block length [B] | S2 | S2-SLC | SLC | RS-RS |
|---|---|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | | Transfer time in [ms] | | | |
| 16 | 3.68 | 5.63 | 6.69 | 8.91 | 528 | 4.67 | 6.94 | 8.17 | 9.92 |
| 32 | 3.74 | 5.67 | 6.77 | 8.90 | 544 | 4.69 | 6.94 | 8.22 | 9.89 |
| 48 | 3.78 | 5.76 | 6.83 | 8.94 | 560 | 4.71 | 6.98 | 8.26 | 9.93 |
| 64 | 3.80 | 5.67 | 6.88 | 8.96 | 576 | 4.74 | 6.95 | 8.28 | 9.94 |
| 80 | 3.88 | 6.00 | 7.03 | 9.11 | 592 | 4.78 | 7.05 | 8.31 | 10.00 |
| 96 | 3.93 | 5.95 | 7.04 | 9.12 | 608 | 4.80 | 7.02 | 8.34 | 10.02 |
| 112 | 3.96 | 5.95 | 7.11 | 9.18 | 624 | 4.83 | 7.11 | 8.38 | 10.02 |
| 128 | 3.98 | 5.98 | 7.16 | 9.19 | 640 | 4.85 | 7.13 | 8.42 | 10.02 |
| 144 | 4.03 | 6.11 | 7.23 | 9.21 | 656 | 4.89 | 7.27 | 8.46 | 10.02 |
| 160 | 4.05 | 6.16 | 7.24 | 9.26 | 672 | 4.92 | 7.22 | 8.52 | 10.02 |
| 176 | 4.06 | 6.12 | 7.26 | 9.28 | 688 | 4.95 | 7.30 | 8.55 | 10.07 |
| 192 | 4.16 | 6.26 | 7.44 | 9.46 | 704 | 5.01 | 7.35 | 8.60 | 10.14 |
| 208 | 4.18 | 6.28 | 7.50 | 9.48 | 720 | 5.01 | 7.42 | 8.79 | 10.18 |
| 224 | 4.20 | 6.25 | 7.52 | 9.53 | 736 | 5.02 | 7.36 | 8.68 | 10.22 |
| 240 | 4.26 | 6.41 | 7.53 | 9.56 | 752 | 5.02 | 7.39 | 8.71 | 10.24 |
| 256 | 4.28 | 6.36 | 7.60 | 9.59 | 768 | 5.04 | 7.35 | 8.75 | 10.24 |
| 272 | 4.32 | 6.45 | 7.69 | 9.60 | 784 | 5.07 | 7.47 | 8.79 | 10.27 |
| 288 | 4.34 | 6.48 | 7.70 | 9.63 | 800 | 5.11 | 7.46 | 8.82 | 10.33 |
| 304 | 4.43 | 6.74 | 7.86 | 9.82 | 816 | 5.14 | 7.53 | 8.88 | 10.37 |
| 320 | 4.45 | 6.58 | 7.91 | 9.84 | 832 | 5.17 | 7.59 | 8.92 | 10.39 |
| 336 | 4.50 | 6.58 | 7.94 | 9.88 | 848 | 5.20 | 7.71 | 8.97 | 10.42 |
| 352 | 4.52 | 6.69 | 7.99 | 9.92 | 864 | 5.23 | 7.65 | 8.99 | 10.45 |
| 368 | 4.56 | 6.67 | 8.05 | 9.95 | 880 | 5.25 | 7.68 | 9.04 | 10.48 |
| 384 | 4.59 | 6.83 | 8.11 | 9.96 | 896 | 5.28 | 7.76 | 9.07 | 10.49 |
| 400 | 4.61 | 6.80 | 8.11 | 10.01 | 912 | 5.31 | 7.81 | 9.13 | 10.54 |
| 416 | 4.70 | 6.92 | 8.28 | 10.15 | 928 | 5.33 | 7.73 | 9.18 | 10.56 |
| 432 | 4.73 | 6.88 | 8.32 | 10.18 | 944 | 5.35 | 7.77 | 9.24 | 10.58 |
| 448 | 4.76 | 7.03 | 8.41 | 10.22 | 960 | 5.39 | 7.89 | 9.26 | 10.61 |
| 464 | 4.78 | 7.01 | 8.40 | 10.26 | 976 | 5.42 | 8.05 | 9.29 | 10.66 |
| 480 | 4.58 | 6.76 | 8.06 | 9.76 | 992 | 5.49 | 8.01 | 9.45 | 10.68 |
| 496 | 4.60 | 6.83 | 8.09 | 9.80 | 1008 | 5.52 | 8.15 | 9.46 | 10.71 |
| 512 | 4.64 | 6.98 | 8.14 | 9.82 | 1024 | 5.56 | 8.13 | 9.52 | 10.72 |

Table C.1: PVM transfer time of small data blocks between various workstations: S2 - SPARCstation2, SLC - SUN Sparc SLC, RS - RS6000

- PVM

| Block length [KB] | S2-S2 | S2-SLC | SLC-SLC | RS-RS | Block length [KB] | S2 | S2-SLC | SLC | RS-RS |
|---|---|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | | Transfer time in [ms] | | | |
| 1 | 5.55 | 8.25 | 9.51 | 10.74 | 33 | 65.57 | 93.04 | 112.92 | 102.50 |
| 2 | 7.57 | 10.65 | 12.50 | 12.87 | 34 | 65.82 | 94.94 | 111.04 | 103.81 |
| 3 | 9.54 | 13.14 | 15.36 | 14.64 | 35 | 67.54 | 96.94 | 113.28 | 105.17 |
| 4 | 13.58 | 18.69 | 21.93 | 21.87 | 36 | 70.76 | 102.17 | 120.39 | 112.26 |
| 5 | 14.49 | 20.09 | 23.69 | 22.76 | 37 | 71.90 | 103.33 | 121.61 | 113.74 |
| 6 | 15.58 | 22.05 | 25.72 | 24.04 | 38 | 73.28 | 106.51 | 123.30 | 115.20 |
| 7 | 17.12 | 24.34 | 27.75 | 25.66 | 39 | 77.11 | 112.01 | 129.50 | 122.54 |
| 8 | 20.86 | 29.37 | 34.30 | 33.54 | 40 | 77.89 | 112.07 | 130.87 | 123.00 |
| 9 | 21.63 | 31.16 | 36.03 | 34.60 | 41 | 78.96 | 114.09 | 132.96 | 125.11 |
| 10 | 22.68 | 32.82 | 37.57 | 36.12 | 42 | 80.51 | 116.00 | 134.96 | 126.54 |
| 11 | 24.22 | 34.97 | 39.09 | 37.85 | 43 | 84.14 | 123.51 | 141.53 | 133.89 |
| 12 | 28.00 | 40.80 | 45.69 | 45.03 | 44 | 85.02 | 123.03 | 143.34 | 135.04 |
| 13 | 28.93 | 41.80 | 47.61 | 45.99 | 45 | 86.13 | 125.61 | 145.05 | 136.46 |
| 14 | 30.08 | 43.37 | 49.84 | 47.15 | 46 | 87.73 | 127.60 | 147.04 | 137.64 |
| 15 | 31.87 | 45.45 | 51.90 | 48.77 | 47 | 91.35 | 131.97 | 153.14 | 145.00 |
| 16 | 35.28 | 50.67 | 60.58 | 56.01 | 48 | 92.12 | 132.51 | 154.87 | 145.32 |
| 17 | 36.22 | 52.11 | 59.82 | 57.47 | 49 | 93.20 | 134.97 | 157.20 | 147.14 |
| 18 | 37.20 | 53.81 | 61.56 | 58.61 | 50 | 94.81 | 137.61 | 159.20 | 149.11 |
| 19 | 38.93 | 56.71 | 63.66 | 60.09 | 51 | 99.08 | 143.36 | 165.61 | 155.84 |
| 20 | 42.19 | 61.30 | 70.03 | 67.22 | 52 | 99.21 | 143.56 | 167.54 | 156.89 |
| 21 | 43.22 | 63.04 | 72.08 | 69.00 | 53 | 106.02 | 145.87 | 169.23 | 158.23 |
| 22 | 44.33 | 64.12 | 73.86 | 69.88 | 54 | 103.22 | 148.72 | 171.46 | 160.19 |
| 23 | 46.01 | 65.93 | 77.28 | 71.06 | 55 | 105.41 | 152.67 | 177.60 | 166.99 |
| 24 | 49.39 | 71.11 | 82.41 | 77.99 | 56 | 106.38 | 157.74 | 179.26 | 168.54 |
| 25 | 50.35 | 72.39 | 84.36 | 80.09 | 57 | 107.28 | 155.80 | 180.99 | 169.81 |
| 26 | 51.35 | 73.85 | 86.70 | 81.06 | 58 | 109.06 | 157.74 | 183.67 | 170.72 |
| 27 | 53.13 | 76.34 | 92.03 | 82.58 | 59 | 112.48 | 163.45 | 189.04 | 178.02 |
| 28 | 56.43 | 81.34 | 98.76 | 89.80 | 60 | 113.54 | 163.89 | 192.27 | 180.68 |
| 29 | 57.47 | 82.64 | 100.59 | 91.32 | 61 | 114.38 | 165.38 | 194.06 | 180.33 |
| 30 | 58.66 | 84.23 | 102.79 | 92.36 | 62 | 116.55 | 167.64 | 195.58 | 181.92 |
| 31 | 60.40 | 86.32 | 105.11 | 93.72 | 63 | 119.27 | 172.72 | 201.61 | 195.12 |
| 32 | 63.74 | 91.50 | 111.39 | 100.40 | 64 | 120.24 | 173.86 | 203.99 | 190.84 |

Table C.2: PVM transfer time of large data blocks between various workstations: S2 - SPARCstation2, SLC - SUN Sparc SLC, RS - RS6000

- P-4

| Block length [B] | S2-S2 | S2-SLC | SLC-SLC | RS-RS | Block length [B] | S2 | S2-SLC | SLC | RS-RS |
|---|---|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | | Transfer time in [ms] | | | |
| 16 | 1.26 | 2.01 | 2.38 | 3.72 | 528 | 1.56 | 2.56 | 2.88 | 3.92 |
| 32 | 1.27 | 2.02 | 2.45 | 3.70 | 544 | 1.58 | 2.57 | 2.90 | 3.90 |
| 48 | 1.27 | 2.04 | 2.44 | 3.71 | 560 | 1.59 | 2.60 | 2.91 | 3.86 |
| 64 | 1.31 | 2.06 | 2.51 | 3.71 | 576 | 1.60 | 2.61 | 2.92 | 3.86 |
| 80 | 1.36 | 2.12 | 2.60 | 3.71 | 592 | 1.65 | 2.61 | 2.93 | 3.86 |
| 96 | 1.36 | 2.14 | 2.61 | 3.73 | 608 | 1.63 | 2.64 | 2.95 | 3.87 |
| 112 | 1.39 | 2.14 | 2.61 | 3.72 | 624 | 1.68 | 2.64 | 2.95 | 3.90 |
| 128 | 1.39 | 2.15 | 2.66 | 3.77 | 640 | 1.67 | 2.75 | 2.96 | 3.92 |
| 144 | 1.40 | 2.18 | 2.65 | 3.78 | 656 | 1.73 | 2.67 | 2.98 | 3.94 |
| 160 | 1.39 | 2.18 | 2.67 | 3.78 | 672 | 1.72 | 2.70 | 2.98 | 3.98 |
| 176 | 1.40 | 2.17 | 2.67 | 3.79 | 688 | 1.74 | 2.71 | 3.01 | 3.99 |
| 192 | 1.40 | 2.20 | 2.67 | 3.79 | 704 | 1.75 | 2.72 | 3.05 | 3.99 |
| 208 | 1.41 | 2.21 | 2.68 | 3.79 | 720 | 1.77 | 2.74 | 3.06 | 4.00 |
| 224 | 1.42 | 2.25 | 2.70 | 3.79 | 736 | 1.79 | 2.77 | 3.05 | 4.01 |
| 240 | 1.42 | 2.30 | 2.72 | 3.80 | 752 | 1.79 | 2.78 | 3.07 | 4.01 |
| 256 | 1.42 | 2.34 | 2.72 | 3.82 | 768 | 1.82 | 2.79 | 3.07 | 4.04 |
| 272 | 1.42 | 2.38 | 2.73 | 3.85 | 784 | 1.83 | 2.81 | 3.11 | 4.07 |
| 288 | 1.45 | 2.38 | 2.76 | 3.85 | 800 | 1.84 | 2.83 | 3.13 | 4.08 |
| 304 | 1.44 | 2.40 | 2.77 | 3.86 | 816 | 1.88 | 2.88 | 3.14 | 4.12 |
| 320 | 1.43 | 2.41 | 2.79 | 3.85 | 832 | 1.89 | 2.89 | 3.13 | 4.14 |
| 336 | 1.43 | 2.43 | 2.81 | 3.82 | 848 | 1.91 | 2.92 | 3.15 | 4.16 |
| 352 | 1.47 | 2.49 | 2.82 | 3.85 | 864 | 1.93 | 2.92 | 3.17 | 4.18 |
| 368 | 1.47 | 2.50 | 2.84 | 3.87 | 880 | 1.94 | 2.93 | 3.20 | 4.20 |
| 384 | 1.51 | 2.51 | 2.84 | 3.90 | 896 | 1.96 | 2.95 | 3.22 | 4.21 |
| 400 | 1.51 | 2.51 | 2.86 | 3.94 | 912 | 1.99 | 2.97 | 3.25 | 4.24 |
| 416 | 1.53 | 2.56 | 2.87 | 3.93 | 928 | 1.99 | 2.99 | 3.24 | 4.26 |
| 432 | 1.55 | 2.58 | 2.87 | 3.91 | 944 | 2.02 | 3.00 | 3.24 | 4.28 |
| 448 | 1.57 | 2.60 | 2.86 | 3.91 | 960 | 2.03 | 3.01 | 3.28 | 4.29 |
| 464 | 1.61 | 2.65 | 2.92 | 3.89 | 976 | 2.05 | 3.02 | 3.31 | 4.34 |
| 480 | 1.63 | 2.65 | 2.94 | 3.90 | 992 | 2.06 | 3.03 | 3.31 | 4.36 |
| 496 | 1.64 | 2.67 | 2.97 | 3.94 | 1008 | 2.10 | 3.04 | 3.34 | 4.37 |
| 512 | 1.55 | 2.56 | 2.88 | 3.92 | 1024 | 2.09 | 3.06 | 3.33 | 4.40 |

Table C.3: P4 transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC, RS - RS6000

- P-4

| Block length [KB] | S2-S2 | S2-SLC | SLC-SLC | RS-RS | Block length [KB] | S2 | S2-SLC | SLC | RS-RS |
|---|---|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | | Transfer time in [ms] | | | |
| 1 | 2.17 | 3.12 | 3.34 | 4.39 | 33 | 37.43 | 52.56 | 56.16 | 62.49 |
| 2 | 3.13 | 4.24 | 5.04 | 6.22 | 34 | 38.83 | 53.56 | 57.43 | 64.04 |
| 3 | 4.34 | 5.61 | 6.37 | 8.18 | 35 | 40.13 | 55.70 | 59.40 | 67.04 |
| 4 | 5.47 | 7.93 | 8.55 | 9.50 | 36 | 40.86 | 57.63 | 61.42 | 68.24 |
| 5 | 6.45 | 9.19 | 9.97 | 11.73 | 37 | 41.90 | 58.84 | 62.79 | 70.43 |
| 6 | 7.73 | 10.26 | 11.27 | 13.49 | 38 | 43.13 | 59.84 | 64.41 | 71.77 |
| 7 | 9.33 | 12.41 | 13.50 | 15.39 | 39 | 44.68 | 62.12 | 66.07 | 74.67 |
| 8 | 9.92 | 14.06 | 15.22 | 16.76 | 40 | 45.39 | 63.60 | 67.93 | 75.60 |
| 9 | 10.86 | 15.63 | 16.48 | 18.95 | 41 | 46.45 | 65.00 | 69.56 | 77.63 |
| 10 | 12.24 | 16.35 | 17.71 | 20.86 | 42 | 47.60 | 66.12 | 70.65 | 79.25 |
| 11 | 13.63 | 18.48 | 19.82 | 22.92 | 43 | 49.07 | 68.13 | 72.54 | 82.34 |
| 12 | 14.43 | 20.12 | 21.64 | 24.18 | 44 | 49.85 | 69.95 | 74.66 | 83.68 |
| 13 | 15.30 | 21.43 | 23.00 | 26.26 | 45 | 50.77 | 71.18 | 76.01 | 85.81 |
| 14 | 16.67 | 22.43 | 24.19 | 28.05 | 46 | 52.13 | 72.35 | 77.22 | 86.90 |
| 15 | 18.08 | 24.53 | 26.22 | 30.10 | 47 | 53.66 | 74.48 | 79.13 | 89.94 |
| 16 | 18.83 | 26.30 | 29.03 | 31.39 | 48 | 54.26 | 76.04 | 81.27 | 90.97 |
| 17 | 19.81 | 27.60 | 29.57 | 33.37 | 49 | 55.23 | 77.34 | 82.78 | 93.25 |
| 18 | 21.02 | 29.06 | 30.80 | 35.05 | 50 | 56.56 | 78.62 | 83.96 | 94.94 |
| 19 | 22.54 | 30.86 | 32.88 | 37.33 | 51 | 57.89 | 80.59 | 85.90 | 98.02 |
| 20 | 23.75 | 32.46 | 34.85 | 38.50 | 52 | 58.72 | 82.31 | 88.19 | 99.43 |
| 21 | 24.39 | 33.70 | 36.17 | 40.64 | 53 | 59.78 | 84.50 | 89.51 | 101.31 |
| 22 | 25.48 | 34.87 | 37.51 | 42.29 | 54 | 60.94 | 84.90 | 90.53 | 102.13 |
| 23 | 26.89 | 37.11 | 39.58 | 44.66 | 55 | 62.46 | 86.85 | 92.64 | 105.54 |
| 24 | 27.65 | 39.17 | 41.72 | 45.86 | 56 | 63.03 | 88.48 | 94.63 | 107.18 |
| 25 | 28.63 | 43.27 | 42.96 | 47.88 | 57 | 64.14 | 90.24 | 96.13 | 109.45 |
| 26 | 30.00 | 48.52 | 44.21 | 49.67 | 58 | 65.39 | 90.87 | 97.20 | 110.51 |
| 27 | 31.38 | 61.20 | 46.31 | 52.17 | 59 | 66.77 | 93.07 | 99.48 | 112.98 |
| 28 | 32.03 | 45.21 | 48.19 | 53.76 | 60 | 67.46 | 95.56 | 101.20 | 114.44 |
| 29 | 33.10 | 46.44 | 49.53 | 57.48 | 61 | 68.60 | 96.04 | 102.80 | 117.18 |
| 30 | 34.31 | 47.66 | 51.14 | 57.72 | 62 | 69.88 | 96.99 | 104.02 | 118.50 |
| 31 | 35.74 | 50.10 | 53.01 | 59.44 | 63 | 71.25 | 99.92 | 105.99 | 121.51 |
| 32 | 36.45 | 51.39 | 54.80 | 60.59 | 64 | 71.90 | 100.80 | 107.94 | 122.22 |

Table C.4: P4 transfer time of large data blocks between various workstations: S2 - SPARCstation2, SLC - SUN Sparc SLC, RS - RS6000

- ANSA

| Block length [B] | S2-S2 | S2-SLC | SLC-SLC | Block length [B] | S2 | S2-SLC | SLC |
|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | Transfer time in [ms] | | |
| 16 | 1.61 | 2.91 | 3.09 | 528 | 2.62 | 3.83 | 4.57 |
| 32 | 1.66 | 2.95 | 3.05 | 544 | 2.64 | 3.77 | 4.62 |
| 48 | 1.67 | 2.98 | 3.09 | 560 | 2.69 | 3.80 | 4.66 |
| 64 | 1.71 | 3.01 | 3.16 | 576 | 2.68 | 3.77 | 4.72 |
| 80 | 1.73 | 3.04 | 3.19 | 592 | 2.69 | 3.83 | 4.77 |
| 96 | 1.87 | 3.05 | 3.26 | 608 | 2.72 | 3.86 | 4.82 |
| 112 | 1.94 | 3.07 | 3.31 | 624 | 2.74 | 3.87 | 4.89 |
| 128 | 1.87 | 3.10 | 3.37 | 640 | 2.78 | 3.94 | 4.97 |
| 144 | 1.95 | 3.14 | 3.43 | 656 | 2.82 | 3.97 | 4.97 |
| 160 | 1.85 | 3.20 | 3.46 | 672 | 2.83 | 3.96 | 4.97 |
| 176 | 1.95 | 3.19 | 3.53 | 688 | 2.86 | 4.00 | 5.02 |
| 192 | 2.08 | 3.26 | 3.56 | 704 | 2.87 | 4.03 | 5.07 |
| 208 | 2.05 | 3.28 | 3.63 | 720 | 2.90 | 4.05 | 5.13 |
| 224 | 2.31 | 3.30 | 3.68 | 736 | 2.90 | 4.08 | 5.17 |
| 240 | 2.27 | 3.33 | 3.72 | 752 | 2.93 | 4.10 | 5.22 |
| 256 | 2.36 | 3.36 | 3.79 | 768 | 2.94 | 4.11 | 5.26 |
| 272 | 2.27 | 3.51 | 3.83 | 784 | 2.94 | 4.29 | 5.32 |
| 288 | 2.37 | 3.46 | 3.89 | 800 | 2.98 | 4.21 | 5.37 |
| 304 | 2.36 | 3.45 | 3.96 | 816 | 2.99 | 4.20 | 5.42 |
| 320 | 2.42 | 3.50 | 3.99 | 832 | 3.02 | 4.23 | 5.47 |
| 336 | 2.48 | 3.50 | 4.03 | 848 | 3.05 | 4.28 | 5.52 |
| 352 | 2.41 | 3.56 | 4.10 | 864 | 3.03 | 4.28 | 5.57 |
| 368 | 2.47 | 3.54 | 4.15 | 880 | 3.08 | 4.30 | 5.62 |
| 384 | 2.47 | 3.58 | 4.20 | 896 | 3.08 | 4.34 | 5.67 |
| 400 | 2.59 | 3.63 | 4.26 | 912 | 3.09 | 4.38 | 5.73 |
| 416 | 2.63 | 3.65 | 4.31 | 928 | 3.12 | 4.38 | 5.77 |
| 432 | 2.60 | 3.62 | 4.31 | 944 | 3.16 | 4.41 | 5.83 |
| 448 | 2.52 | 3.58 | 4.32 | 960 | 3.18 | 4.46 | 5.87 |
| 464 | 2.60 | 3.62 | 4.37 | 976 | 3.22 | 4.50 | 5.94 |
| 480 | 2.59 | 3.67 | 4.43 | 992 | 3.22 | 4.57 | 5.99 |
| 496 | 2.60 | 3.67 | 4.47 | 1008 | 3.24 | 4.52 | 6.05 |
| 512 | 2.63 | 3.72 | 4.52 | 1024 | 3.27 | 4.57 | 6.10 |

Table C.5: ANSA transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

- ANSA

| Block length [KB] | S2-S2 | S2-SLC | SLC-SLC | Block length [KB] | S2 | S2-SLC | SLC |
|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | Transfer time in [ms] | | |
| 1 | 3.27 | 4.58 | 6.13 | 33 | 105.04 | 111.40 | 110.51 |
| 2 | 5.05 | 7.44 | 8.93 | 34 | 111.46 | 118.57 | 116.83 |
| 3 | 5.88 | 9.71 | 11.08 | 35 | 112.53 | 124.55 | 125.22 |
| 4 | 12.62 | 15.00 | 15.03 | 36 | 114.55 | 129.76 | 125.18 |
| 5 | 15.73 | 20.14 | 20.01 | 37 | 120.85 | 128.59 | 130.28 |
| 6 | 15.81 | 20.02 | 20.03 | 38 | 123.02 | 132.29 | 130.54 |
| 7 | 20.60 | 25.09 | 25.02 | 39 | 123.77 | 132.70 | 135.28 |
| 8 | 23.78 | 25.70 | 30.04 | 40 | 124.93 | 135.85 | 135.38 |
| 9 | 26.86 | 32.18 | 30.05 | 41 | 132.98 | 142.45 | 140.40 |
| 10 | 26.70 | 30.87 | 30.35 | 42 | 134.27 | 146.07 | 142.61 |
| 11 | 31.82 | 40.12 | 40.05 | 43 | 135.37 | 147.29 | 145.49 |
| 12 | 33.32 | 40.35 | 45.07 | 44 | 143.54 | 154.34 | 155.24 |
| 13 | 37.18 | 42.12 | 45.08 | 45 | 146.32 | 154.99 | 155.45 |
| 14 | 44.46 | 48.48 | 50.06 | 46 | 149.13 | 157.83 | 160.32 |
| 15 | 44.71 | 53.75 | 55.07 | 47 | 153.54 | 163.79 | 162.58 |
| 16 | 47.78 | 53.02 | 55.07 | 48 | 155.29 | 164.23 | 165.39 |
| 17 | 53.07 | 60.46 | 60.13 | 49 | 157.42 | 168.24 | 170.10 |
| 18 | 56.73 | 64.08 | 65.08 | 50 | 158.74 | 169.15 | 170.42 |
| 19 | 59.03 | 65.46 | 70.14 | 51 | 165.38 | 175.16 | 175.38 |
| 20 | 57.63 | 67.65 | 70.15 | 52 | 168.48 | 178.99 | 180.00 |
| 21 | 66.84 | 70.56 | 72.99 | 53 | 169.83 | 180.64 | 180.45 |
| 22 | 68.75 | 76.19 | 75.13 | 54 | 176.56 | 187.72 | 185.46 |
| 23 | 71.00 | 77.40 | 75.22 | 55 | 179.28 | 190.28 | 190.38 |
| 24 | 78.40 | 82.20 | 85.20 | 56 | 180.54 | 191.42 | 191.34 |
| 25 | 79.75 | 85.46 | 88.63 | 57 | 188.89 | 197.40 | 196.45 |
| 26 | 81.40 | 90.40 | 90.23 | 58 | 189.18 | 204.39 | 218.09 |
| 27 | 89.25 | 94.82 | 91.55 | 59 | 191.18 | 203.02 | 216.12 |
| 28 | 91.14 | 96.03 | 96.72 | 60 | 191.94 | 203.82 | 241.95 |
| 29 | 92.24 | 100.48 | 98.43 | 61 | 201.01 | 209.75 | 235.19 |
| 30 | 92.88 | 100.86 | 103.45 | 62 | 202.05 | 213.97 | 232.56 |
| 31 | 103.97 | 108.49 | 108.69 | 63 | 202.99 | 218.19 | 236.93 |
| 32 | 104.89 | 108.70 | 110.25 | 64 | 213.09 | 224.27 | 236.54 |

Table C.6: ANSA transfer time of large data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

- SR

| Block length [B] | S2-S2 | S2-SLC | SLC-SLC | Block length [B] | S2 | S2-SLC | SLC |
|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | Transfer time in [ms] | | |
| 16 | 5.12 | 9.78 | 6.23 | 528 | 10.02 | 10.02 | 12.39 |
| 32 | 5.01 | 9.89 | 6.33 | 544 | 10.00 | 10.01 | 12.50 |
| 48 | 5.10 | 9.97 | 6.30 | 560 | 10.00 | 10.06 | 12.54 |
| 64 | 5.01 | 9.91 | 5.99 | 576 | 10.04 | 10.05 | 12.38 |
| 80 | 5.04 | 10.04 | 5.98 | 592 | 9.97 | 10.02 | 10.90 |
| 96 | 5.04 | 9.97 | 7.35 | 608 | 10.00 | 10.05 | 10.33 |
| 112 | 5.06 | 10.01 | 11.64 | 624 | 10.04 | 10.01 | 10.16 |
| 128 | 5.01 | 10.03 | 11.66 | 640 | 9.99 | 10.01 | 10.14 |
| 144 | 5.01 | 10.01 | 11.94 | 656 | 9.99 | 10.05 | 10.17 |
| 160 | 5.01 | 10.01 | 11.93 | 672 | 10.05 | 10.02 | 10.09 |
| 176 | 5.01 | 10.05 | 12.09 | 688 | 9.99 | 10.02 | 10.10 |
| 192 | 5.04 | 10.01 | 11.89 | 704 | 9.99 | 10.06 | 10.13 |
| 208 | 5.05 | 10.03 | 11.90 | 720 | 10.06 | 10.04 | 10.09 |
| 224 | 5.00 | 10.05 | 11.90 | 736 | 10.00 | 10.04 | 10.13 |
| 240 | 5.01 | 10.03 | 11.98 | 752 | 9.99 | 10.04 | 10.89 |
| 256 | 5.00 | 10.03 | 11.97 | 768 | 10.02 | 10.01 | 10.01 |
| 272 | 5.00 | 10.06 | 11.81 | 784 | 10.00 | 10.04 | 10.25 |
| 288 | 5.02 | 10.02 | 12.13 | 800 | 7.83 | 10.05 | 10.06 |
| 304 | 5.05 | 10.01 | 12.01 | 816 | 5.35 | 10.00 | 10.22 |
| 320 | 5.00 | 10.06 | 11.95 | 832 | 5.32 | 10.02 | 10.09 |
| 336 | 5.01 | 10.01 | 12.42 | 848 | 5.33 | 10.04 | 10.09 |
| 352 | 5.01 | 10.03 | 11.97 | 864 | 5.35 | 10.03 | 10.03 |
| 368 | 5.01 | 10.05 | 11.99 | 880 | 5.34 | 10.04 | 10.08 |
| 384 | 5.05 | 10.00 | 12.41 | 896 | 5.08 | 10.05 | 10.07 |
| 400 | 5.07 | 10.02 | 12.63 | 912 | 5.06 | 10.00 | 10.03 |
| 416 | 5.00 | 10.06 | 12.55 | 928 | 5.08 | 10.04 | 10.06 |
| 432 | 5.00 | 10.02 | 12.64 | 944 | 5.02 | 10.04 | 10.22 |
| 448 | 5.01 | 10.02 | 12.51 | 960 | 5.03 | 10.03 | 10.04 |
| 464 | 8.76 | 10.07 | 12.44 | 976 | 5.01 | 10.01 | 10.10 |
| 480 | 8.04 | 10.04 | 12.59 | 992 | 5.01 | 10.07 | 10.03 |
| 496 | 9.77 | 10.03 | 12.49 | 1008 | 5.03 | 10.03 | 10.09 |
| 512 | 9.96 | 10.08 | 12.58 | 1024 | 5.06 | 10.05 | 10.20 |

Table C.7: SR transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

- SR

| Block length [KB] | S2-S2 | S2-SLC | SLC-SLC | Block length [KB] | S2 | S2-SLC | SLC |
|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | Transfer time in [ms] | | |
| 1 | 5.05 | 10.05 | 10.51 | 33 | 54.29 | 148.18 | 83.62 |
| 2 | 8.19 | 15.00 | 12.62 | 34 | 55.29 | 108.98 | 81.78 |
| 3 | 10.07 | 15.03 | 13.61 | 35 | 55.16 | 117.18 | 83.51 |
| 4 | 10.02 | 20.28 | 15.56 | 36 | 55.50 | 196.37 | 85.15 |
| 5 | 10.06 | 22.38 | 21.32 | 37 | 60.73 | 126.58 | 91.39 |
| 6 | 12.66 | 25.14 | 20.11 | 38 | 60.63 | 120.95 | 91.74 |
| 7 | 15.01 | 25.28 | 22.49 | 39 | 61.16 | 148.58 | 92.03 |
| 8 | 100.13 | 93.44 | 25.74 | 40 | 65.20 | 159.01 | 100.49 |
| 9 | 100.37 | 99.12 | 28.27 | 41 | 65.38 | 129.23 | 100.46 |
| 10 | 19.22 | 35.63 | 32.11 | 42 | 65.50 | 147.51 | 102.23 |
| 11 | 20.00 | 55.29 | 33.76 | 43 | 66.65 | 185.00 | 110.02 |
| 12 | 25.06 | 53.94 | 35.28 | 44 | 69.95 | 175.69 | 109.01 |
| 13 | 25.08 | 46.53 | 39.55 | 45 | 70.76 | 149.65 | 108.37 |
| 14 | 25.50 | 49.40 | 40.12 | 46 | 71.79 | 163.64 | 108.52 |
| 15 | 25.26 | 92.14 | 41.64 | 47 | 75.60 | 160.92 | 115.25 |
| 16 | 27.74 | 99.58 | 56.84 | 48 | 75.10 | 153.17 | 115.27 |
| 17 | 30.13 | 55.61 | 44.60 | 49 | 75.97 | 157.79 | 124.34 |
| 18 | 30.20 | 86.09 | 48.49 | 50 | 78.37 | 187.98 | 120.91 |
| 19 | 30.93 | 86.98 | 52.88 | 51 | 78.55 | 165.50 | 121.16 |
| 20 | 30.89 | 66.82 | 51.79 | 52 | 80.60 | 163.36 | 123.39 |
| 21 | 35.46 | 71.74 | 52.24 | 53 | 81.07 | 197.68 | 123.18 |
| 22 | 38.86 | 100.08 | 57.33 | 54 | 82.11 | 177.98 | 127.81 |
| 23 | 40.72 | 100.00 | 62.08 | 55 | 86.11 | 171.64 | 130.26 |
| 24 | 40.05 | 77.16 | 61.29 | 56 | 86.38 | 188.44 | 128.42 |
| 25 | 42.01 | 100.00 | 63.04 | 57 | 86.03 | 200.64 | 135.72 |
| 26 | 45.11 | 101.18 | 73.19 | 58 | 89.74 | 178.91 | 136.04 |
| 27 | 44.54 | 86.72 | 71.81 | 59 | 90.33 | 191.75 | 136.77 |
| 28 | 45.01 | 90.50 | 69.69 | 60 | 91.85 | 201.66 | 140.14 |
| 29 | 46.84 | 101.74 | 69.89 | 61 | 94.78 | 196.54 | 141.19 |
| 30 | 50.04 | 101.98 | 73.85 | 62 | 95.89 | 204.36 | 144.08 |
| 31 | 50.87 | 99.96 | 78.53 | 63 | 98.07 | 204.01 | 149.31 |
| 32 | 50.18 | 105.22 | 76.65 | 64 | 98.86 | 203.58 | 150.73 |

Table C.8: SR transfer time of large data blocks between various workstations: S2 - SPARCstation2, SLC - SUN Sparc SLC.

- Strand 88

| Block length [B] | S2-S2 | S2-SLC | SLC-SLC | Block length [B] | S2 | S2-SLC | SLC |
|---|---|---|---|---|---|---|---|
| | Transfer time in [$\mu$s] | | | | Transfer time in [$\mu$s] | | |
| 16 | 30.04 | 30.08 | 30.12 | 528 | 30.05 | 30.05 | 30.01 |
| 32 | 30.01 | 32.08 | 30.13 | 544 | 56.01 | 30.18 | 38.33 |
| 48 | 30.01 | 30.27 | 30.05 | 560 | 30.06 | 30.02 | 30.05 |
| 64 | 30.01 | 30.04 | 30.11 | 576 | 30.05 | 30.05 | 30.01 |
| 80 | 42.76 | 30.05 | 30.25 | 592 | 30.05 | 30.17 | 48.86 |
| 96 | 30.01 | 30.05 | 30.15 | 608 | 30.05 | 45.22 | 30.02 |
| 112 | 30.05 | 30.02 | 30.38 | 624 | 30.05 | 30.01 | 30.09 |
| 128 | 30.00 | 30.04 | 30.18 | 640 | 30.09 | 30.19 | 43.22 |
| 144 | 30.00 | 30.03 | 31.20 | 656 | 30.05 | 30.29 | 30.00 |
| 160 | 30.00 | 30.13 | 30.45 | 672 | 30.05 | 30.04 | 30.07 |
| 176 | 30.12 | 30.31 | 31.26 | 688 | 30.05 | 35.58 | 31.05 |
| 192 | 30.06 | 30.04 | 30.20 | 704 | 31.07 | 30.07 | 30.07 |
| 208 | 30.04 | 30.04 | 30.08 | 720 | 40.49 | 30.24 | 35.61 |
| 224 | 30.04 | 30.05 | 30.04 | 736 | 30.06 | 30.01 | 30.04 |
| 240 | 30.11 | 30.48 | 30.10 | 752 | 30.01 | 30.02 | 36.44 |
| 256 | 30.06 | 30.01 | 30.16 | 768 | 30.04 | 30.22 | 31.72 |
| 272 | 30.06 | 30.03 | 30.78 | 784 | 40.72 | 30.01 | 30.04 |
| 288 | 30.05 | 30.25 | 32.22 | 800 | 30.04 | 30.23 | 40.69 |
| 304 | 30.05 | 30.06 | 30.48 | 816 | 30.10 | 30.00 | 30.01 |
| 320 | 32.01 | 30.08 | 30.04 | 832 | 30.04 | 30.18 | 31.11 |
| 336 | 58.01 | 30.07 | 30.02 | 848 | 30.04 | 30.03 | 30.04 |
| 352 | 30.00 | 30.07 | 30.04 | 864 | 30.02 | 30.23 | 30.56 |
| 368 | 30.05 | 30.18 | 43.86 | 880 | 30.04 | 35.26 | 30.08 |
| 384 | 30.05 | 30.02 | 30.01 | 896 | 30.13 | 31.23 | 30.10 |
| 400 | 30.05 | 30.02 | 30.01 | 912 | 30.01 | 30.23 | 30.22 |
| 416 | 30.03 | 48.56 | 30.01 | 928 | 30.04 | 30.04 | 30.04 |
| 432 | 30.05 | 30.17 | 30.62 | 944 | 30.04 | 30.26 | 30.22 |
| 448 | 30.05 | 30.00 | 49.64 | 960 | 44.26 | 30.28 | 30.21 |
| 464 | 30.05 | 30.02 | 30.01 | 976 | 30.02 | 30.04 | 30.04 |
| 480 | 30.01 | 33.48 | 30.04 | 992 | 30.01 | 30.21 | 30.23 |
| 496 | 30.03 | 30.17 | 37.12 | 1008 | 30.05 | 30.01 | 30.02 |
| 512 | 30.05 | 30.01 | 43.64 | 1024 | 30.03 | 30.16 | 30.23 |

Table C.9: Strand 88 transfer time of small data blocks between various workstations: S2 - SPARCstation2, SLC - SUN Sparc SLC.

- Strand 88

| Block length [KB] | S2-S2 | S2-SLC | SLC-SLC | Block length [KB] | S2 | S2-SLC | SLC |
|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | Transfer time in [ms] | | |
| 1 | 30.04 | 33.80 | 30.03 | 33 | 92.57 | 108.81 | 103.95 |
| 2 | 30.12 | 30.75 | 31.25 | 34 | 90.70 | 109.71 | 105.19 |
| 3 | 30.06 | 35.39 | 35.24 | 35 | 91.54 | 112.46 | 106.39 |
| 4 | 49.97 | 50.42 | 50.37 | 36 | 95.42 | 115.82 | 107.69 |
| 5 | 50.30 | 50.42 | 50.44 | 37 | 95.57 | 118.36 | 108.11 |
| 6 | 50.24 | 50.65 | 50.59 | 38 | 97.90 | 118.21 | 107.97 |
| 7 | 60.02 | 65.46 | 65.37 | 39 | 100.62 | 121.18 | 110.56 |
| 8 | 60.14 | 68.31 | 65.60 | 40 | 98.51 | 120.63 | 115.87 |
| 9 | 65.12 | 70.48 | 65.83 | 41 | 98.54 | 123.58 | 117.27 |
| 10 | 65.13 | 70.68 | 73.08 | 42 | 100.48 | 123.44 | 117.26 |
| 11 | 65.11 | 76.30 | 75.05 | 43 | 100.70 | 127.62 | 117.98 |
| 12 | 65.18 | 75.23 | 75.93 | 44 | 105.36 | 129.18 | 118.62 |
| 13 | 70.22 | 75.39 | 75.92 | 45 | 105.60 | 130.92 | 120.61 |
| 14 | 70.07 | 81.25 | 75.92 | 46 | 105.60 | 131.02 | 122.98 |
| 15 | 70.26 | 80.64 | 76.14 | 47 | 105.95 | 134.22 | 123.76 |
| 16 | 73.03 | 79.47 | 76.42 | 48 | 106.10 | 133.85 | 123.70 |
| 17 | 75.24 | 83.35 | 79.90 | 49 | 106.48 | 139.21 | 126.82 |
| 18 | 75.34 | 84.33 | 81.38 | 50 | 110.71 | 139.35 | 128.35 |
| 19 | 71.87 | 85.25 | 81.66 | 51 | 110.80 | 141.99 | 131.60 |
| 20 | 78.00 | 86.88 | 82.31 | 52 | 112.96 | 141.65 | 132.88 |
| 21 | 75.36 | 88.49 | 86.22 | 53 | 113.53 | 143.27 | 132.92 |
| 22 | 75.32 | 91.80 | 90.90 | 54 | 114.71 | 145.13 | 133.93 |
| 23 | 80.18 | 92.04 | 89.94 | 55 | 117.10 | 148.15 | 135.10 |
| 24 | 80.32 | 92.38 | 91.64 | 56 | 116.00 | 148.37 | 137.68 |
| 25 | 80.55 | 97.04 | 91.83 | 57 | 116.97 | 150.84 | 137.96 |
| 26 | 83.02 | 97.48 | 91.99 | 58 | 120.92 | 150.94 | 139.12 |
| 27 | 85.30 | 101.04 | 92.25 | 59 | 117.75 | 155.31 | 140.65 |
| 28 | 85.35 | 102.25 | 96.08 | 60 | 122.11 | 155.72 | 143.19 |
| 29 | 88.75 | 102.65 | 100.27 | 61 | 120.96 | 157.53 | 142.80 |
| 30 | 90.02 | 106.97 | 101.17 | 62 | 122.00 | 157.97 | 145.36 |
| 31 | 86.54 | 106.11 | 102.03 | 63 | 125.93 | 160.85 | 146.70 |
| 32 | 92.05 | 109.15 | 102.09 | 64 | 124.88 | 163.60 | 146.20 |

Table C.10: Srand 88 ransfer time of large data blocks between various workstations: S2 - SPARCstation2, SLC - SUN Sparc SLC.

- Linda

| Block length [B] | S2-S2 | S2-SLC | SLC-SLC | Block length [B] | S2 | S2-SLC | SLC |
|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | Transfer time in [ms] | | |
| 16 | 2.61 | 3.43 | 5.12 | 528 | 3.50 | 4.46 | 5.46 |
| 32 | 2.62 | 3.49 | 4.92 | 544 | 3.52 | 4.51 | 5.47 |
| 48 | 2.68 | 3.74 | 5.00 | 560 | 3.57 | 4.50 | 5.57 |
| 64 | 2.72 | 3.63 | 4.59 | 576 | 3.61 | 4.58 | 5.59 |
| 80 | 2.73 | 3.87 | 4.69 | 592 | 3.65 | 4.57 | 5.58 |
| 96 | 2.78 | 3.81 | 4.83 | 608 | 3.70 | 4.61 | 5.64 |
| 112 | 2.81 | 3.66 | 4.69 | 624 | 3.68 | 4.63 | 5.70 |
| 128 | 2.87 | 3.72 | 4.76 | 640 | 3.72 | 4.67 | 5.72 |
| 144 | 2.88 | 4.03 | 4.65 | 656 | 3.76 | 4.67 | 5.76 |
| 160 | 2.96 | 4.67 | 4.78 | 672 | 3.79 | 4.72 | 5.78 |
| 176 | 3.01 | 4.21 | 4.79 | 688 | 3.81 | 4.77 | 5.78 |
| 192 | 2.99 | 4.10 | 4.81 | 704 | 3.85 | 4.77 | 5.86 |
| 208 | 3.02 | 3.91 | 4.88 | 720 | 3.88 | 4.81 | 5.85 |
| 224 | 3.03 | 3.90 | 4.95 | 736 | 3.90 | 4.86 | 5.96 |
| 240 | 3.09 | 3.94 | 4.91 | 752 | 3.92 | 4.85 | 5.95 |
| 256 | 3.13 | 4.04 | 5.03 | 768 | 3.93 | 4.96 | 6.07 |
| 272 | 3.17 | 4.03 | 5.01 | 784 | 3.97 | 5.04 | 6.13 |
| 288 | 3.20 | 4.04 | 5.01 | 800 | 4.01 | 4.97 | 6.05 |
| 304 | 3.24 | 4.13 | 5.54 | 816 | 4.04 | 4.99 | 6.07 |
| 320 | 3.28 | 4.21 | 5.21 | 832 | 4.08 | 5.03 | 6.15 |
| 336 | 3.27 | 4.37 | 5.08 | 848 | 4.15 | 5.00 | 6.12 |
| 352 | 3.33 | 5.20 | 5.17 | 864 | 4.15 | 5.07 | 6.16 |
| 368 | 3.32 | 4.68 | 5.14 | 880 | 4.17 | 5.09 | 6.22 |
| 384 | 3.34 | 4.30 | 5.28 | 896 | 4.18 | 5.07 | 6.52 |
| 400 | 3.38 | 4.57 | 5.32 | 912 | 4.20 | 5.14 | 6.26 |
| 416 | 3.41 | 4.48 | 5.34 | 928 | 4.25 | 5.21 | 6.33 |
| 432 | 3.45 | 4.48 | 5.37 | 944 | 4.30 | 5.21 | 6.33 |
| 448 | 3.38 | 4.29 | 5.28 | 960 | 4.35 | 5.28 | 6.41 |
| 464 | 3.41 | 4.33 | 5.29 | 976 | 4.36 | 5.34 | 6.43 |
| 480 | 3.42 | 4.38 | 5.35 | 992 | 4.40 | 5.34 | 6.46 |
| 496 | 3.47 | 4.38 | 5.39 | 1008 | 4.41 | 5.37 | 6.54 |
| 512 | 3.48 | 4.42 | 5.65 | 1024 | 4.45 | 5.42 | 6.57 |

Table C.11: Linda transfer time of small data blocks between various workstations: S2 - SPARC-station2, SLC - SUN Sparc SLC.

- Linda

| Block length [KB] | S2-S2 | S2-SLC | SLC-SLC | Block length [KB] | S2 | S2-SLC | SLC |
|---|---|---|---|---|---|---|---|
| | Transfer time in [ms] | | | | Transfer time in [ms] | | |
| 1 | 4.46 | 5.39 | 6.58 | 33 | 53.46 | 62.62 | 74.26 |
| 2 | 6.46 | 7.81 | 9.05 | 34 | 54.93 | 64.27 | 76.22 |
| 3 | 8.48 | 10.00 | 11.62 | 35 | 56.28 | 65.72 | 78.72 |
| 4 | 10.21 | 11.90 | 13.50 | 36 | 57.59 | 67.36 | 81.06 |
| 5 | 12.23 | 14.04 | 15.77 | 37 | 59.04 | 69.23 | 81.43 |
| 6 | 14.12 | 16.33 | 18.25 | 38 | 60.50 | 70.96 | 83.64 |
| 7 | 15.10 | 18.40 | 22.35 | 39 | 62.53 | 72.83 | 86.19 |
| 8 | 16.77 | 20.68 | 25.03 | 40 | 63.69 | 74.23 | 88.09 |
| 9 | 18.22 | 22.23 | 27.19 | 41 | 65.07 | 75.89 | 90.14 |
| 10 | 19.74 | 23.96 | 29.17 | 42 | 66.53 | 77.47 | 91.49 |
| 11 | 21.22 | 25.58 | 30.92 | 43 | 67.97 | 79.21 | 93.52 |
| 12 | 22.56 | 27.21 | 32.93 | 44 | 69.42 | 80.82 | 95.62 |
| 13 | 24.03 | 28.80 | 34.65 | 45 | 70.65 | 82.32 | 97.30 |
| 14 | 25.48 | 30.40 | 36.51 | 46 | 72.63 | 84.76 | 100.20 |
| 15 | 26.82 | 32.09 | 38.38 | 47 | 74.01 | 86.08 | 101.97 |
| 16 | 28.54 | 34.14 | 41.00 | 48 | 75.16 | 87.78 | 104.14 |
| 17 | 30.05 | 35.65 | 43.01 | 49 | 76.90 | 89.47 | 105.78 |
| 18 | 31.31 | 37.26 | 44.62 | 50 | 78.24 | 91.11 | 107.56 |
| 19 | 32.67 | 38.86 | 46.36 | 51 | 79.78 | 92.71 | 109.52 |
| 20 | 34.19 | 40.51 | 48.38 | 52 | 81.34 | 94.60 | 111.31 |
| 21 | 35.73 | 42.30 | 50.14 | 53 | 82.79 | 96.41 | 113.32 |
| 22 | 37.07 | 43.72 | 51.86 | 54 | 84.31 | 97.92 | 115.80 |
| 23 | 38.90 | 46.17 | 54.86 | 55 | 85.57 | 99.71 | 120.83 |
| 24 | 40.17 | 47.49 | 56.69 | 56 | 87.02 | 101.33 | 121.29 |
| 25 | 41.73 | 49.11 | 58.85 | 57 | 88.49 | 102.70 | |
| 26 | 43.20 | 50.84 | 60.54 | 58 | 90.13 | 104.51 | |
| 27 | 44.63 | 52.38 | 62.83 | 59 | 91.31 | 106.03 | |
| 28 | 46.07 | 54.15 | 64.16 | 60 | 93.00 | 107.69 | 126.72 |
| 29 | 47.57 | 55.76 | 66.42 | 61 | 94.84 | 110.01 | 129.91 |
| 30 | 48.73 | 57.24 | 68.18 | 62 | 96.02 | 111.35 | 131.75 |
| 31 | 50.57 | 59.45 | 70.89 | 63 | 97.53 | 113.13 | 133.76 |
| 32 | 52.18 | 61.08 | 72.73 | 64 | 99.09 | 114.89 | 135.41 |

Table C.12: Linda transfer time of large data blocks between various workstations: S2 - SPARCstation2, SLC - SUN Sparc SLC.