Efficiency Evaluation of Some Parallelization Tools on a Workstation Cluster Using the NAS Parallel Benchmarks

Florian Sukup
Computing Center
Vienna University of Technology

January 94

Contents

Chapter 1

Introduction

Parallel computing becomes increasingly important, because the performance, needed to solve some of todays problems is too expensive or not available on uniprocessor systems. A parallel computing system can be a solution, such systems can mostly solve these problems faster and cheaper. In that case several single components are to compute a problem together.

In a parallel algorithm, which is needed to get a solution, communication has to be performed. Tools must be available to perform this communication in parallel algorithm.

Besides portability, the efficiency of these tools is particularly important when it refers to a parallel computer system with distributed memory.

For that purpose an evaluation project has been started on a cluster of nine workstations using the five most commonly used tools, namely: PVM 2.x [3], PVM 3.x [4], p4 [5], Express [6] and Linda [7]. Four of the five kernels of the NAS parallel benchmarks [1] have been implemented to evaluate the tools mentioned above.

The hardware, a workstation cluster described in section 2.1, was chosen, because workstation clusters became very popular recently and so far, as it is known, no reasonable evaluation has been done.

In the next chapter there will be a description of the hardware and a discussion about the software used for parallelization. In chapter 3 a description and motivation for choosing the NAS parallel benchmarks can be found. The implementation, including the parallel algorithms, is explained in chapter 4. The results of the tool evaluation are presented in chapter 5.

Chapter 2

Hard- and Software for the Benchmarks

2.1 The Cluster

The hardware used for this evaluation were nine IBM RS6000-320H workstations with 25 MHz clock rate and 16 MB memory per workstation. They are connected via a token ring capable of 16 Mbits per second, the operating system is AIX 3.2.3e. One of the nine workstations, which acts as a front end, has 32 MB memory and the home file system, where the user data reside, is NFS-mounted.

2.2 Tools for Parallelization

For parallel computing using dedicated workstation clusters, as used within this project, the message passing programming model can be used. An alternative approach would be the usage of parallelizing compilers, like HPF [11], but so far neither commercial products nor public domain compilers are available. Therefore, in this project it was concentrated on available and supported commercial products or public domain tools, which support the message passing model to perform communication.

In the next sections an overview regarding these tools is given. There will be an example in each section, which shows how to send a part of an array `work_array` beginning with index `begin` and of length `len` from one worker process to another. Several other topics are discussed from the author's point of view.

2.2.1 PVM 2.x

PVM 2.x [3] was developed at the Oak Ridge Laboratory in 1991. PVM 2.x should allow the utilization of a heterogeneous network of parallel and sequential computers as a single computational resource.

In PVM 2.4.2, which is used in this work, a message must be put into a send buffer before sending it. Receiving a message is similar only the other way round:

Messages are received via a receive buffer. See Figure 2.1. This is a difference to other tools, where it is not necessary to put data into and get data from message buffers explicitly. PVM 2.x has only the two elementary send and receive commands for communication, broadcast messages can be send if the destination is set to -1. A barrier command is also available. PVM 2.x is easy to learn and to work with.

PVM 2.x uses daemons to control the PVM processes. (An example of a daemon is *sendmail* which handles all the incoming and outgoing electronic mail on a Unix system.) With starting the PVM daemon on the master machine, one daemon is started on each workstation. Several PVM processes can run on each workstation controlled only by one daemon. Each user needs his own daemons, daemon sharing with other users is not possible.

PVM processes do not communicate directly with each other, but through daemons. The message is copied from the send buffer of one PVM process to the daemon and after the transmission into the receive buffer of the other PVM process. The daemon of PVM 2.x can be started in interactive mode or in background on the master machine. In interactive mode processes can be controlled by several commands: For example, the state of each process can be shown. If the states of all processes are "waiting for a message", the user can recognize a deadlock. In this case or another reason for a reset, all processes can be killed very easily by a simple command and it is not necessary to quit PVM daemon and restart it for another run. This can save a lot of time.

PVM 2.x adds the name of the machine to the workers output, so that the origin of each line of output can be determined. This is very useful for debugging. There is also another possibility to detect errors: Xab [8], which is a tool for run time monitoring of PVM programs.

PVM 2.x did run from the beginning and was stable almost all the time. Therefore, it can be recommended for program development because one can be sure that an occurring error is the programmer's fault and not in PVM 2.x.

2.2.2 PVM 3.x

PVM 3.x [4] was developed at the Oak Ridge Laboratory in 1993. PVM 3.x has the same aims as PVM 2.x, but contains some new features.

PVM 3.x [4] was still under development during this evaluation project. Therefore, it was experimented with several versions up to 3.2.3. The presented results refer to version 3.2.3. In this version there is still the necessity of the explicit use of message buffers as described in section 2.2.1 (see Figure 2.2). But in future versions of PVM 3.x data may be copied directly out of user's memory. The number of commands concerning the communication is increased. In addition to the send and receive commands there is a barrier, multicast and broadcast command. Unfortunately, the broadcast command sends the message to the sender, too. This will be changed in later versions, but in the meantime broadcasting is to be simulated with the multicast command.

The PVM3-daemon is started in the background and afterwards a console process can be started if interactive mode is requested. This is an advantage compared to

```
int *work_array;
int begin, len;
int to,
      /* worker process, the message is to be sent to */
    type;
      /* message type */

initsend();
      /* The send buffer is initialized */
putnint (work_array + begin, len);
      /* Data are put into the send buffer */
snd ("nodeprogram", to, type);
      /* The message is sent */

rcv (type);
      /* The message is received */
getnint (work_array + begin, len);
      /* Data are got out of the receive buffer */
```

Figure 2.1: **Sending and receiving messages in PVM 2.x**

```
int *work_array;
int begin, len;
int to,
      /* worker process, the message is to be sent to */
    type;
      /* message type */

pvm_initsend(PvmDataRaw);
      /* The send buffer is initialized */
putnint (work_array + begin, len, 1);
      /* Data are put into the send buffer */
snd (to, type);
      /* The message is sent */

pvm_recv (-1, type);
      /* The message is received */
pvm_upkint (work_array + begin, len, 1);
      /* Data are got out of the receive buffer */
```

Figure 2.2: **Sending and receiving messages in PVM 3.x**

PVM 2.x, because there is the possibility to change between running processes in the background and controlling the processes interactively. In the console process more features are available than in PVM 2.x, but commands have become more complex. Furthermore one could find PVM 2.x is more clearly arranged than PVM 3.x. In contrast to PVM 2.x the communication may not run via the daemons. The user can switch through a command between direct communication or communication by the way of daemons.

pvm_advise() is the routine to change between direct communication or communication via daemon. The parameter can be "PvmAllowDirect" = 2 if direct links are allowed but not requested or "PvmRouteDirect" = 3 if direct links are requested. In chapter 5 can be seen which way of communication is the better one for each benchmark.

At the beginning PVM 3.x was not stable. For a rather long time it was not possible to run all benchmarks, because especially at the conjugate gradient benchmark programs got stuck at execution. Conclusively stability improved and all benchmarks could be implemented with PVM 3.x.

2.2.3 p4

P4 [5] was developed at Argonne National Laboratory in 1992 for programming a variety of parallel machines.

At this project version 1.3 + patches are used. The user of p4 has a choice of several commands to perform communication between workers. In addition to the send and receive commands, which can be seen in the example in Figure 2.3, there are also broadcast, barrier commands and global operations.

If a p4-code is executed a file is needed, called procgroup, which contains information about the computer system. The number of workers is already determined by this file and it is hardly possible to run the application with a different number of workers. This is a weakness in comparison to the rest of the tools, where the number of workers can be determined during run time. p4 has no daemon that controls the workers. This could be the explanation, that at the beginning processes got lost sometimes, especially if workers run incorrectly. Of course, this also may happen in PVM, but in p4 the probability is quite higher. Another disadvantage is the long setup and finishup time, whenever a program is executed.

As concerning program writing there are also advantages and disadvantages compared to other tools. On the one hand p4 helps the user to debug p4 programs. There is the possibility of tracing the execution of the programs. There are several error levels that can be chosen by the user, the higher the more information is shown about the execution of a program. And the programmer also gets some possibilities for printing debugging information with a special command, if requested depending on a debug level. The p4 process id is added to this debug information. In addition parallel program behavior can be studied with Upshot [9], an X-based graphics tool for viewing log files produced by parallel programs. On the other hand writing a p4 program can become a bit complicated because of the use of the message buffers, which must be allocated first and freed afterwards.

p4 had some problems concerning the execution of programs, too. The programs of the integer sort and the conjugate gradient benchmark got stuck at execution. The originator of the tool had to fix a bug to get the integer sort benchmark run and improve timing enabling to run the conjugate gradient benchmark. p4 was the last tool all of the four benchmarks could be implemented with.

2.2.4 Express

Express [6], Version 3.2.5, was developed by Parasoft Corporation in 1992.

Two programming models are available in Express: The *Host-Node* and the *Cubix* programming model. In the *Host-Node* programming model the program is divided into two pieces, one for the host and one for the nodes. The host, which controls computations on the system, does initialization, cleaning up, time measurement etc. The Nodes do the essential work. In the *Cubix* programming model the entire application is executed on the parallel system. Processes are available to handle the I/O, system and graphical needs of the program. The *Host-Node* programming model is used for implementing the NAS parallel benchmarks.

```
int *work_array;
int begin, len;
int to,
      /* worker process, the message is to be sent to */
    from,
      /* worker process, the message is received from */
    type;
      /* message type */
int *msg;
      /* message buffer */

p4_send (type, to, (char *)(work_array + begin),
    len * sizeof(int));
      /* Data are copied into the send buffer internaly
      and the message is sent */

msg = NULL;
      /* This is necessary to let p4_recv know, that it must
      allocate a message buffer by itself */
p4_recv (&type, &from, (char **)&msg, &len);
      /* The message is received */
len /= sizeof(int);
for (i=0; i<len; i++)
    work_array[begin + i] = msg[i];
      /* Data are assigned from msg into work_array */
p4_msg_free(msg);
      /* The message buffer must be freed by the user */
```

Figure 2.3: **Sending and receiving messages in p4**

The inter-process communication is performed in a message passing programming model. In Express a lot of commands support this model. There are the elementary send and receive commands, which are used in the *Node* to *Node* programming example in Figure 2.4. A broadcast command and global operations are also available, but a barrier command is missing.

Express helps building virtual structures in a special case. If a user specification for a problem has the topology of a Cartesian grid, it can easily be mapped onto any hardware topology with special Express commands.

There is the possibility to debug Express programs with *NDB*, the Express debugger, which is similar to the *dbx* debugger, known on many Unix machines. Breakpoints, printing variables etc. are available features. But in contrast to the tools described above no process-id or similar information is added to the output.

Exinit, which is the Express initialization program, gets its environment from a file called express.cst. There are three parameter concerning the Express message buffers, which are worth to adapt to each application. These parameters influence the Express message buffers. NNBUF is the number of Express message buffers to be allocated in each node. NBSIZE is the size of each Express message buffer in bytes. Messages longer than this will be cut into smaller pieces and sent in packets. SBSIZE is the socket size. It has to be at least 2 * NBSIZE + 80. Dealing with these parameters can cause big differences in performance. The combinations used in benchmarking can be seen in chapter 5.

At the beginning there were a lot of problems with Express. Firstly, programs ran very unstable. On the average every sixth to eighth program got stuck. Moreover, as in p4, programs could not run at all and always got stuck. It was necessary to wait for a newer version and then still the Unix sockets were overflowed when executing the program of the conjugate gradient benchmark. It was necessary to contact Parasoft that could fix the bug.

2.2.5 Linda

Linda [7], version 2.5, was developed by the SCIENTIFIC Computing Associates Inc. in 1992. Linda should enable users to create parallel programs which perform well in a wide range of computing environments.

The concept of Linda is a little bit different than the concepts of the other tools. As seen in Figure 2.5, messages are sent via the *tuple space*. Only five commands are available. Therefore, Linda is easy to learn. Linda was stable most of the time, although there is no daemon controlling its processes.

Linda has *Tuplescope* for debugging. In addition to the usual debugger features such as single-step mode and breakpoints, *Tuplescope* can display tuple classes, data in specific tuples, and visual indications of process interaction throughout the course of a program run.

There is a general difficulty in programming broadcast messages in Linda. At one of the NAS parallel benchmarks, the "Solving an Unstructured Sparse Linear System by the Conjugate Gradient Method" benchmark (see section 3.3 and section 4.3), almost only broadcast messages are needed. Therefore, the two options for

```
int *work_array;
int begin, len;
int to,
     /* worker process, the message is to be sent to */
   from,
     /* worker process, the message is received from */
   type;
     /* message type */

exwrite ((char *)(work_array + begin), len * sizeof(int),
   &to, &type);
     /* Data are sent from users memory using only one
     command */

exread ((char *)(work_array + begin), len * sizeof(int),
   &from, &type);
     /* Data are received into users memory using only one
     command */
```

Figure 2.4: **Sending and receiving messages in Express**

```
int *work_array;
int begin, len;
int type;
      /* message type */

out (type, (work_array + begin):len);
      /* Data are put into tuple space */

in (type, ?(work_array + begin):);
      /* Data are got from tuple space */
```

Figure 2.5: **Sending and receiving messages in Linda**

programming broadcast messages should be discussed:

- If there are n workers, the one, which wants to broadcast a message, can put $n-1$ tuples in the tuple space, using *out*. Then each of the other workers can read a tuple, using *in*.
- The broadcasting worker only puts one tuple in the tuple space, using *out* again, but then each of the other workers read this tuple, using *rd*. Now the tuple still remains in the tuple space. In some cases one can leave it there, but it has probably to be removed, because an unused tuple is a waste of memory. Besides, Linda could need more time for finding a tuple, because tuple matching becomes more difficult. As a result, leaving a tuple in the tuple space can reduce efficiency.

  To avoid this, a certain worker, for example the sender, must take the tuple out of the tuple space, using *in*. The only problem is that this worker must be sure that each worker has already read this tuple. Therefore, a barrier is needed before taking out the tuple of the tuple space.

On a network a Linda program can be executed with several options. It was experienced with following options (with the exception of “An Embarrassingly Parallel Benchmark”, where too little communication is performed for such experiments). The description is taken from [7]:

-bcast This option enables the tuple broadcast optimization.

+bcast This option disables the tuple broadcast optimization.

-bcastcache This option specifies the size of the broadcast cache. This is a trade-off between memory consumption and hit rate. The default size is 1 Mbyte. This resource is only used when bcast is true.

-high This option causes all workers to be run at normal priority and causes Linda internodal communication to run at full speed.

+high This option causes all workers to run at a nice-ed priority, (unless specifically overridden on a per node per application basis using the nice resource). It also causes Linda internodal communication to be throttled to avoid flooding the network.

-kaon This option turns on the keep alive mechanism.

+kaon This option turns off the keep alive mechanism.

-redirect This option turns on the tuple redirection optimization.

+redirect This option turns off the tuple redirection optimization.

-udp This option specifies the udp datagram size used by the Linda kernel. The default is 7800 bytes.

All possible combinations where tested, the results can be found in Chapter 5. Only +kaon was used all the time.

Chapter 3

The NAS Parallel Benchmarks

The Numerical Aerodynamic Simulation (NAS) parallel benchmark suite [1] has been presented on the Supercomputing Conference 91 [10] as a part of a NAS program, which is based at NASA Ames Research Center. It consists of five kernels and three simulated application benchmarks. Four of the kernels were implemented on the cluster: "An Embarrassingly Parallel Benchmark" to get used with the tools and to see what is the maximum performance of the cluster. Nearly no communication is necessary between the processes to solve this benchmark problem. The other three implemented benchmarks: "A Simple 3D Multigrid Benchmark", "Solving an Unstructured Sparse Linear System by the Conjugate Gradient Method" and "Parallel Sort Over Small Integers" are very communication intensive. The fifth kernel was not implemented because the scope of this project would have been exceeded.

There are several reasons why these benchmarks are chosen for tool evaluation:

- No other benchmark program or strategy is broadly accepted for today's highly parallel systems as well as for traditional shared memory multiprocessor machines. Other known benchmarks in the field, for example SPLASH [2], are not usable for the whole range of parallel computers.
- Relevant data are available for these benchmarks to compare performance of parallel computers even for completely different architectures.
- The NAS parallel benchmarks are used by several institutions, i.e. the High Performance Fortran (HPF) Forum.
- The NAS parallel benchmarks are "paper and pencil" benchmarks, that means the specification of the problem is only algorithmic. Only the behavior of the program according to input and output is specified. And it is up to the person working on a certain architecture to implement the benchmark in a way that fits to this architecture. Naturally, the problem has to be specified in sufficient detail that a unique solution exists. Therefore, the NAS parallel benchmarks take care of the following facts:
  - New algorithmic and software approaches are frequently required by parallel systems. These algorithms and software often differ generally from

conventional methods in implementing source code on a sequential or vector machine.

  – Benchmarks should not favor any particular parallel architecture, they must be "generic". A benchmark specification must preclude architecture-specific code, such as message passing code.

- These benchmarks cannot be parallelized in a trivial way (except the Embarrassingly Parallel Benchmark). A lot of communication is necessary and shows the differences in efficiency between tools.

- There is a verification test at the end of each of these benchmarks. If the verification test is passed the program certainly solved the problem.

The choice of the programming language is also important. In the NAS parallel benchmarks only C or Fortran 77 is allowed for implementation. These restrictions are made to avoid the usage of assembly code and handy tuned libraries. In this way it can be ensured, that the benchmark runs under the same condition each time it is implemented.

The different sizes of a specific problem are also worth mentioning. The sizes of the current benchmarks were chosen so that implementation is possible on current available supercomputers. Of course, it would not make any sense to solve a small problem on a parallel computer. But there are also smaller problem sizes to run a sample scaled problem. This is a great help implementing the benchmarks in the phase of testing the code. But only the large problem size is allowed for evaluating computer systems.

Four of the five kernels of the NAS parallel benchmarks [1] have been implemented. A short description of these four benchmarks is following now.

3.1 An Embarrassingly Parallel Benchmark

Pairs of Gaussian random deviates are generated according to a specific scheme described below and tabulate the number of pairs in successive square annuli.

$2n$ floating point values, where $n = 2^{28}$, are generated by a specified pseudorandom number generator. Then the first two values, x and y, are taken and they are tested if $x^2 + y^2 \leq 1$. If this inequality does not hold, reject this pair and take the next two pseudorandom generated values, otherwise the pair X and Y is computed from x and y in the following way: $X = x\sqrt{\frac{(-2\log(x^2+y^2))}{x^2+y^2}}$ and $Y = y\sqrt{\frac{(-2\log(x^2+y^2))}{x^2+y^2}}$. Now X and Y are independent Gaussian deviates with mean zero and variance one. Approximately $\frac{n\pi}{4}$ pairs will be constructed in this manner.
Finally, for $0 \leq l \leq 9$ tabulate Q_l as the count of the pairs (X, Y) that lie in the square annulus $l \leq \max(|X|, |Y|) < l + 1$, and output the ten Q_l counts.

The operations to be timed are all operations described above, including tabulation and output.

Approximately 2.668×10^{10} floating point operations have to be performed in this benchmark when $n = 2^{28}$. This calculation is based on 19 floating point operations

z_k	$=$	$M^k r_k$	:		
	if $k > 1$				
		r_{k-1}	$=$	$P r_k$	(restrict residual)
		z_{k-1}	$=$	$M^{k-1} r_{k-1}$	(recursive solve)
		z_k	$=$	$Q z_{k-1}$	(prolongate)
		r_k	$=$	$r_k - A Z_k$	(evaluate residual)
		z_k	$=$	$z_k + S r_k$	(apply smoother)
	else				
		z_1	$=$	$S r_1$	(apply smoother)

Table 3.1: **The V-cycle multigrid operator in the multigrid benchmark**

for each pseudorandom number, 12 for each square root and 25 for each logarithm evaluation.

The memory requirement for this benchmark is minimal, storage is required only for the uniform pseudorandom numbers generated in a single batch.

3.2 A Simple 3D Multigrid Benchmark

Four iterations of the V-cycle multigrid algorithm described below are used to obtain an approximate solution u to the discrete Poisson problem

$$\nabla^2 u = v$$

on a $n \times n \times n$ grid with periodic boundary conditions, where $n = 256$.

v is set to 0 except at twenty points where $v = \pm 1$. The iterative solution has to begin with $u = 0$. Each of the four iterations consists of the following two steps, in which $k = \log_2(n)$, that means $k = 8$ in the full scaled problem:

$$r = v - Au \qquad (\textit{evaluate residual})$$

$$u = u + M^k r \qquad (\textit{apply correction})$$

Here M^k denotes the V-cycle multigrid operator, defined in Table 3.1. A, P, Q and S consists of four coefficients c_{0-3}.
c_0 denotes the central coefficient of the 27-point operator, when these coefficients are arranged as a $3 \times 3 \times 3$ cube. Thus, c_0 is the coefficient that multiplies the value at the gridpoint (i, j, k), while c_1 multiplies the six values at grid points which differ by one in exactly one index, c_2 multiplies the next closest twelve values, those that differ by one in exactly two indices, and c_3 multiplies the eight values located at grid points that differ by one in all three indices.
After four iterations of the V-cycle multigrid algorithm the residual has to be evaluated.

Timing begins before evaluating the residual for the first time and after initializing u and v. Timing ends after evaluating the norm the final residual.

In simple implementations are approximately $n^3 \times 478$ floating point operations and about $(n+2)^3 \times \frac{8}{7}$ 64-bit words of memory required. Thus, the full sized benchmark would require 8×10^9 floating point operations. This number can be reduced to 3.905×10^9 at the cost of greater memory usage. This is the official operation count of this benchmark.

3.3 Solving an Unstructured Sparse Linear System by the Conjugate Gradient Method

In this benchmark, the power method is used to find an estimate of the smallest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzeros.

A symmetric, positive definite, sparse matrix of order 14000 has 1853104 nonzero elements. In the following A is the sparse matrix, lower case Roman letters are vectors, x_j is the j^{th} component of x, and lower case Greek letters are scalars. We denote by $||x||$ the Euclidean norm of a vector x, $||x|| = \sqrt{\sum_{i=1}^{14000} x_i^2}$.
The power method is to be implemented as follows:

$x = [1, 1, \ldots, 1]$;
$\zeta_2 = 0$;
$\zeta_1 = 0$;
$\zeta = 0$;
$it = 0$;
outer iteration:
do 15 times
    $it \leftarrow it + 1$
    Solve the system $Az = x$;
    $\zeta_2 = \zeta_1$;
    $\zeta_1 = \zeta$;
    $\zeta = \frac{1}{\max_i\{|z_i|\}} \quad i \in \{0, 1, \ldots, 14000\}$;
    Apply Aitken extrapolation to the last three iterates ζ, ζ_1 and ζ_2
        to produce an improved approximation ζ'
        with the following formula:
        $\zeta' = \zeta - \frac{(\zeta-\zeta_1)^2}{\zeta - 2\zeta_1 + \zeta_2}$.
    Print it, ζ' and $||rv||$,
        the Euclidean norm of the last CG residual vector
        where $rv = x - r$ (see r in the inner iteration)
    $x = \zeta z$;
od

The solution z to the linear system of equations $Az = x$ is to be implemented as follows:

$z = 0;$
$r = x;$
$\rho = r^T r;$
$p = r;$
inner iteration:
do 25 times

$q = Ap;$
$\alpha = \frac{\rho}{(p^T q)};$
$z = z + \alpha p;$
$\rho_0 = \rho;$
$r = r - \alpha q;$
$\rho = r^T r;$
$\beta = \frac{\rho}{\rho_0};$
$p = r + \beta p;$

od

The reported time must be the time required to compute all 15 iterations and print the results, after the matrix is generated and downloaded into the parallel machine, and after the initialization of the starting vector x.

Approximately 1.508×10^9 floating point operations must be performed, the requirement of storage for the timed portion of this kernel is about 22 Mbytes.

3.4 Parallel Sort Over Small Integers

Sort N keys in parallel. The keys are generated by the sequential key generation algorithm described in [1] and initially must be uniformly distributed in memory.

A sequence of keys, $\{K_i \mid i = 0, 1, \ldots, N-1\}$, will be said to be *sorted* if it is arranged in non-descending order, i.e. $K_i \leq K_{i+1} \leq K_{i+2} \ldots$. The *rank* of a particular key in a sequence is the index value i that the key would have if the sequence of keys were sorted. *Ranking* is the process of arriving at a rank for all the keys in a sequence. The benchmark requires computing the rank of the keys.

The sequence of N keys must be mapped into the memory of the parallel processor in the following way: In a distributed memory system with p distinct memory units, each memory unit initially must store N_p keys in a contiguous address space, where

$$N_p = \frac{N}{p}.$$

If N is not evenly divisible by p, then memory units $\{P_j \mid j = 0, 1, \ldots, p-2\}$ will store N_p keys, and memory unit P_{p-1} will store N_{pp} keys, where now

$$N_p = \left\lfloor \frac{N}{p} + 0.5 \right\rfloor$$

$$N_{pp} = N - (p-1)\, N_p.$$

The original sequence of keys must remain intact such that each time the ranking process is repeated the original sequence of keys exists and the same algorithm for ranking is applied.

Partial verification is conducted for each ranking performed. Partial verification consists of comparing a particular subset of ranks with the reference values.

Full verification is conducted after the last ranking is performed. Full verification requires the following:

1. Rearrange the sequence of keys, $\{K_i \mid i = 0, 1, \ldots, N-1\}$, in the order $\{K_j \mid j = r(0), r(1), \ldots, r(N-1)\}$, where $r(0), r(1), \ldots, r(N-1)$ is the last computed sequence of ranks.

2. For every K_i from $i = 0 \ldots N-2$ test that $K_i \leq K_{i+1}$.

If the result of this test is true, then the keys are in sorted order.

The benchmark consists of the following steps:

1. In a scalar sequential manner and using the key generation algorithm described in [1] generate the sequence of N keys.

2. Using the appropriate memory mapping described above, load the N keys into the memory system.

3. Begin timing.

4. Do, for $i = 1$ to I_{max}

   (a) Modify the sequence of keys bey making the following two changes:

$$K_i \leftarrow i$$

$$K_{i+I_{max}} \leftarrow (B_{max} - i)\,.$$

   (b) Compute the rank of each key.

   (c) Perform the partial verification test.

5. End timing.

6. Perform the full verification test.

The specifications given in Table 3.2 shall be used in the benchmark. The computational cost on a sequential machine consists of $2N + B_{max}$ arithmetic operations (integer add or subtract), and $7N + 4B_{max}$ memory references. No floating point operations are required.

N	2^{23}
B_{max}	2^{19}
I_{max}	10

Table 3.2: **Parameter values to be used for the integer sort benchmark**

Chapter 4

The Implementation on the Cluster

The NAS parallel benchmarks were running on an unloaded cluster. Therefore, load balancing was not necessary. As described in section 2.1, the cluster consists of nine RS6000. Hence, the problems were broken done into nine parts, each part of a problem had the same size. The SPMD Model was used, however, there was an additional master process nearly without computational work.

The implementations described in the next sections could be improved if messages were compressed before sending and decompressed after receiving. Especially in the "Parallel Sort Over Small Integer" benchmark compression can easily be implemented (see section 4.4). Compression was not performed in this implementation.

4.1 The Implementation of "An Embarrassingly Parallel Benchmark"

4.1.1 Parallel Algorithm

In this benchmark 2^{29} pseudorandom floating point values must be generated. These 2^{29} values can be seen as 2^{28} pairs. The master divides the number of pairs by the number of workers and tells every worker how many pairs of the pseudorandom floating point values are to be computed. Afterwards the timing begins. A couple of operations, described in section 3.1, are performed on each pair of these values. Operations on one pair of these floating point values do not depend on operations on other pairs. Hence, every worker can do its work independently of the other workers. No communication is necessary except at the end of the computation. Then each worker sends an array of ten integers as its result to the master. The master receives the results of the workers and prints the sum of the received values. After that timing ends.

4.1.2 Specifics in the Implementation

Since in this project there are nine workstations, the 2^{28} pairs cannot be evenly divided. As a result, eight workers get $\lfloor \frac{2^{28}}{9} \rfloor + 1$ pairs and the ninth worker gets the rest.

4.2 The Implementation of the "A Simple 3D Multigrid Benchmark"

4.2.1 Parallel Algorithm

In this benchmark operations are performed on two three dimensional matrices. These three dimensional matrices are divided into parts. Each worker gets one part of each matrix and computes its part of the solution matrix. As described in 3.2 each gridpoint (i, j, k) of the matrices is manipulated by a 27-point operator. Here the difficulty of a parallel implementation appears: Calculating one gridpoint, the neighbouring gridpoints around are needed. Consequently, outer gridpoints can only be calculated if gridpoints of the neighbour workers are available. In the parallel algorithm each worker makes a slice around its matrices that can be seen as cubes. A slice is a copy of the outer gridpoints of the neighbours. Now manipulation with the 27-point operator becomes possible. When calculating a matrix with size n^3, a storage of a matrix with size $(n+2)^3$ is required.

The slice consists of 6 planes, 12 edges and 8 corners. In Figure 4.1 a part of the slice, which is cut into pieces (planes, edges, corners), can be seen. Each worker gets the whole slice from its 26 neighbours. It is necessary to communicate with the neighbours before each matrix operation. To reduce the effort of communication some optimizations are possible. Both the number of messages and number of bytes, which are transferred, can be reduced:

- The number of messages, which are received/sent, can be reduced from 26 to 6. This is possible if each worker sends and receives its messages in the following way (The x'-plane denotes the received plane from the worker's x-side, thus, the x'-plane of the worker is the x-plane of the slice around its cube or matrix):

  1. Send: The two x-planes to the two neighbours which are at the x-sides. See Figure 4.2.
     Receive: The two planes from these workers. See Figure 4.3
  2. Not only the two y-planes are sent to the two neighbours which are at the y-sides but also the y-edges of the x'-planes that were received before. See Figure 4.4. For the receiving worker these edges are the xy-edges of the workers in its xy-direction.
     Receive: The two planes and the four edges from the workers of the y-sides.
     Figure 4.5 shows that the worker has updated the following parts of its slice: The two x'-planes, the two y'-planes and the four xy'-edges.

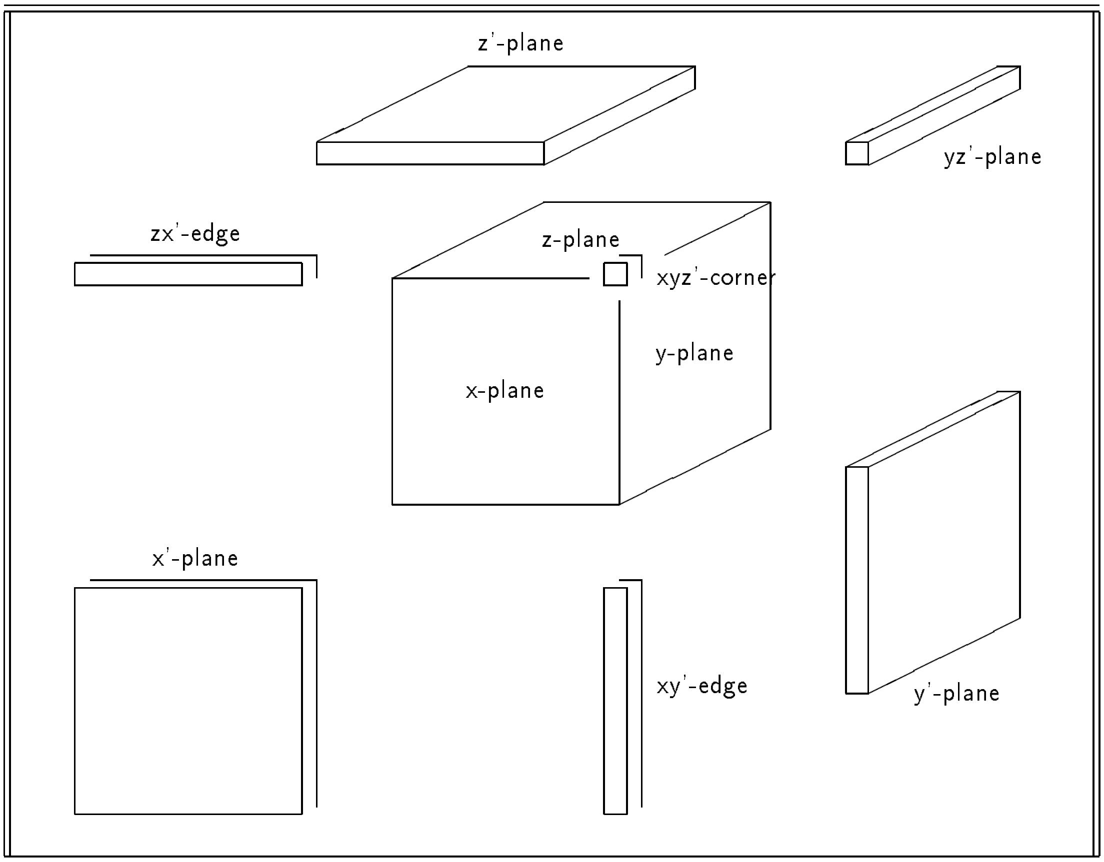

Figure 4.1: **The cube and pieces of its slice**

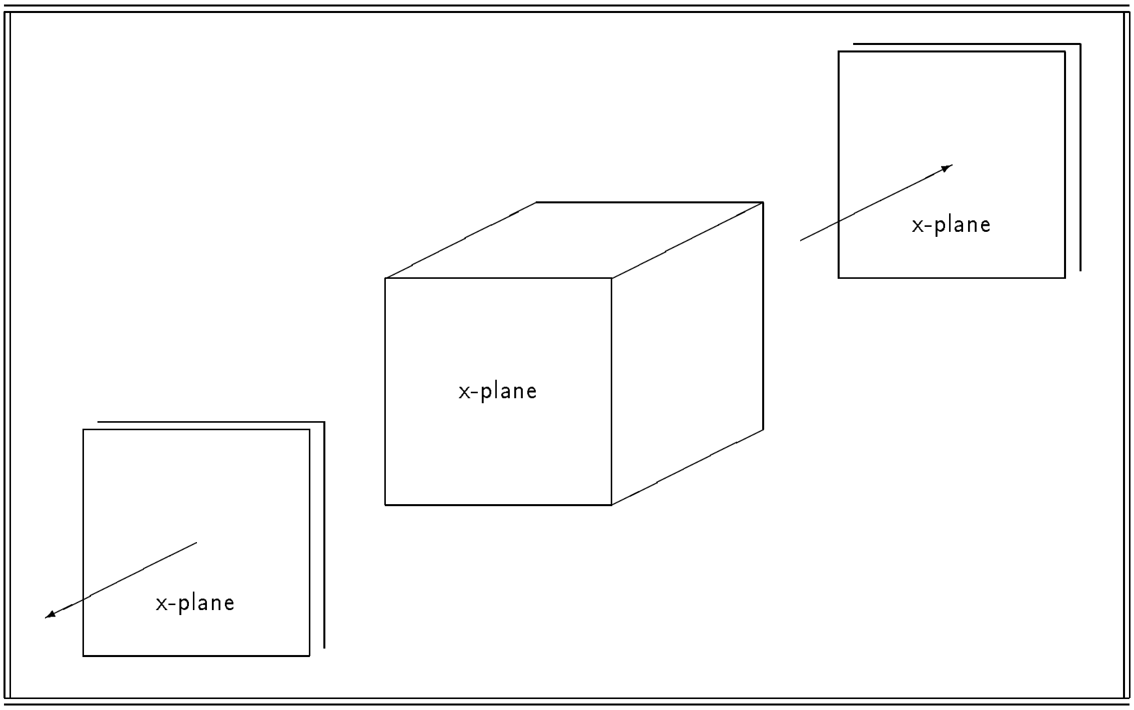

Figure 4.2: **The worker sends its x-planes**

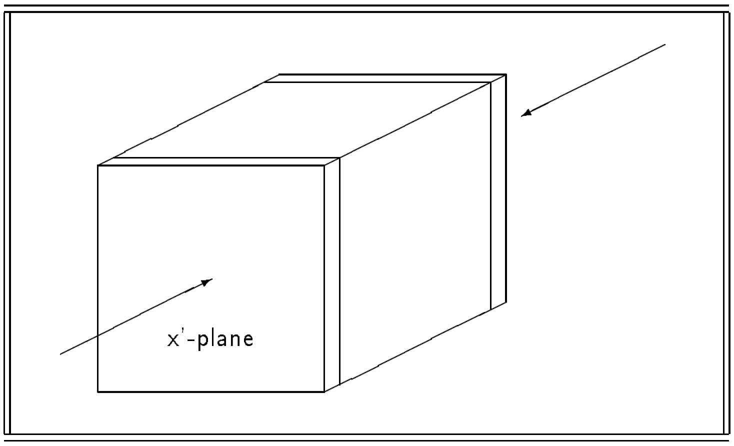

Figure 4.3: **The worker has received its x'-planes**

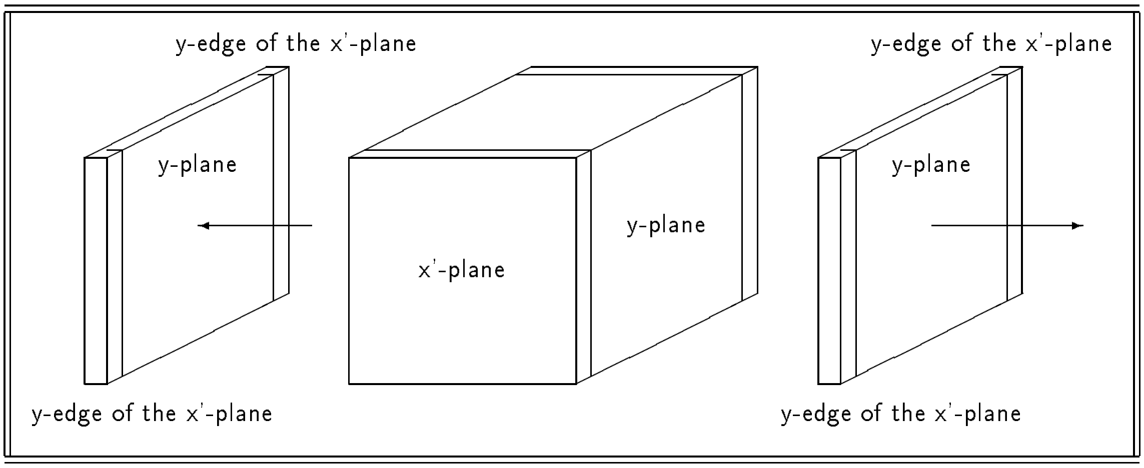

Figure 4.4: **The worker sends its y-planes and parts of its slice**

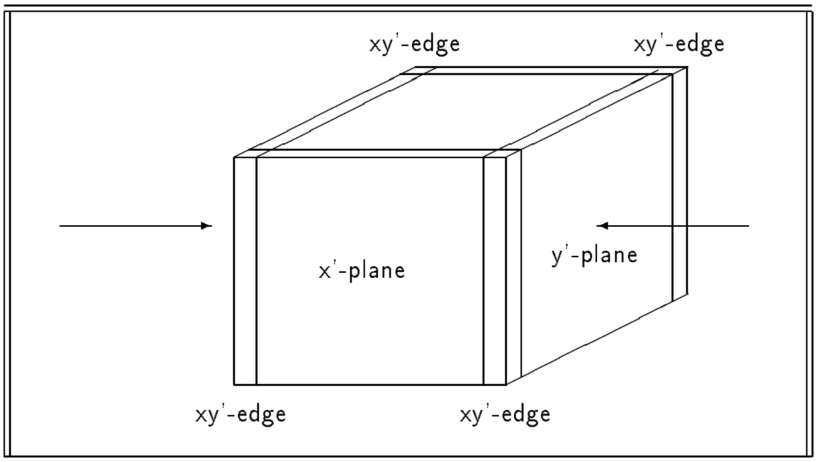

Figure 4.5: **The worker has received its y'-planes and other parts of its slice**

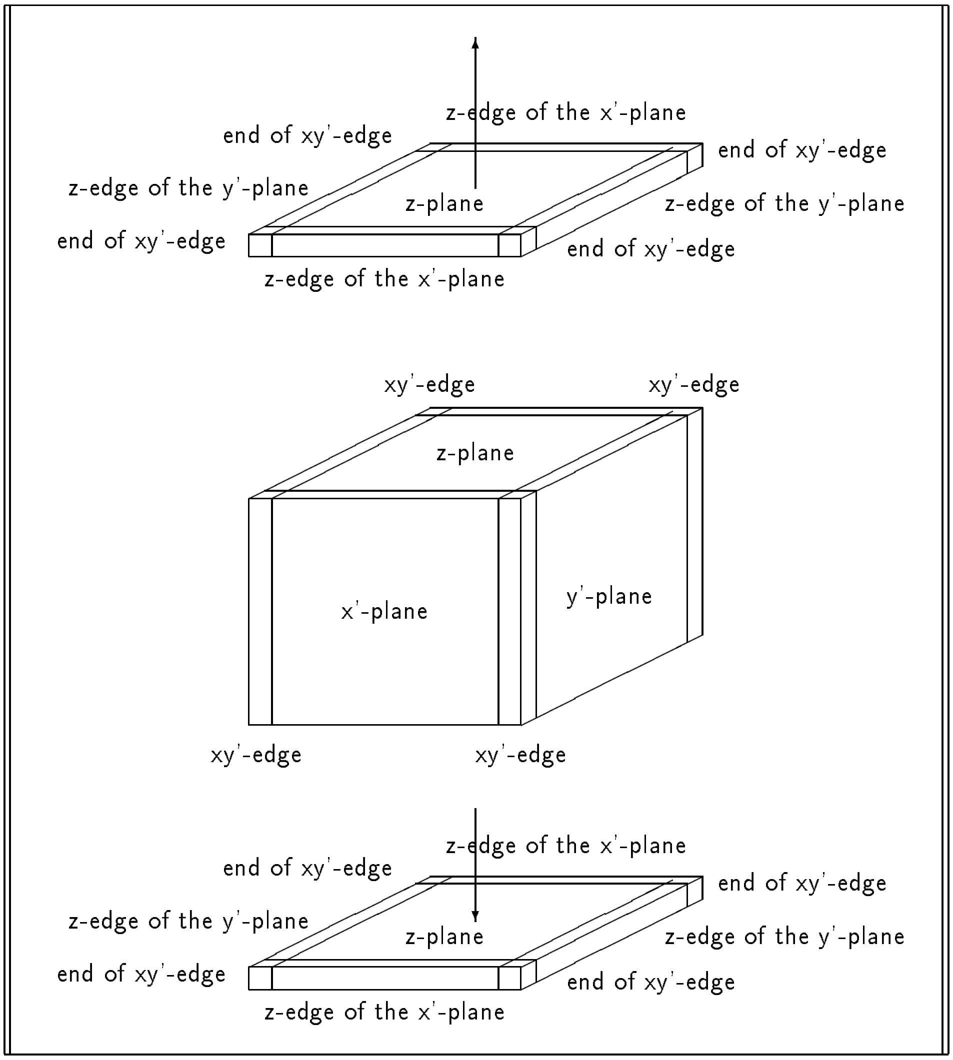

Figure 4.6: **The worker sends its z-planes and parts of its slice**

3. Send: Messages containing the following to the neighbours at the z-sides, see Figure 4.6:
   - z-planes
   - z-edges of the x'-plane: For the receiving worker these are the xz-edges of the workers in the xz-direction.
   - z-edges of the y'-plane: For the receiving worker these are the xy-edges of the workers in the xy-direction.
   - ends of the xy'-edges: For the receiving worker these are the corners of the workers in the xyz-direction.

   After receiving the two messages from the workers at the z-sides the worker has got information from all of its 26 neighbours and the slice around its cube is completed that can be seen in Figure 4.7.

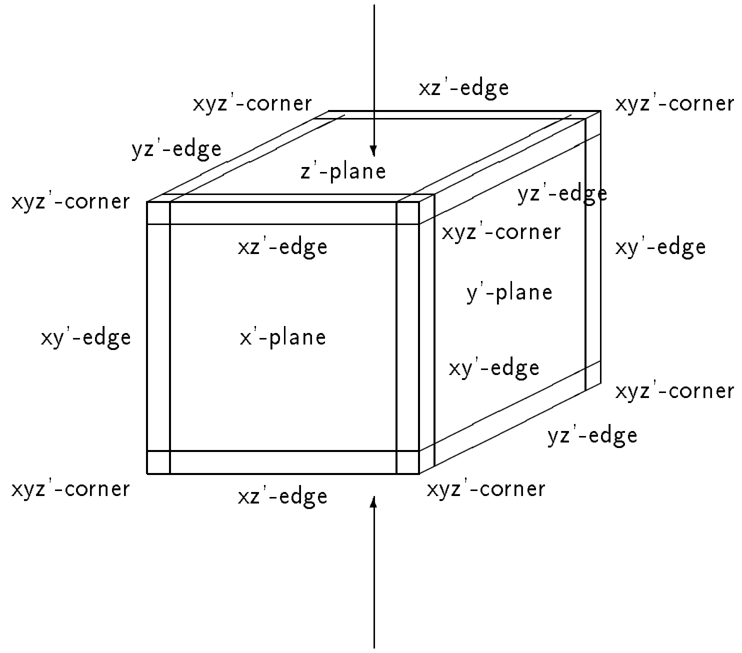

Figure 4.7: **The worker has received its z'-planes and the rest of its slice**

- Two operations (restrict residual) and (prolongate) in Table 3.1 do not need a whole slice around the cube, because the cube is transformed to a smaller one or other way round. Therefore, communication, which is necessary before these two operations can be performed, is reduced to 50% on the average.

4.2.2 Specifics and Differences in implementation

The memory requirements for this benchmark are more than 260 Mbytes. Even if this number was divided by the number of workers, which is nine, each worker would need 29 Mbytes. To avoid swapping problem size was reduced from 256^3 to 128^3, which is the next smaller problem size. The number of operations increases proportional to problem size. In this example the big problem size affords eight times more operations than the implemented one.

The parallel implementation of the benchmark, which was used in this project, is not optimized concerning the number of floating point operations. Consequently, the full scaled problem would require 8×10^9 floating point operations as described in section 3.2. But in Table 5.4 in section 5.2 an eighth of the official operation count (3.905×10^9) was used to calculate MFLOPS.

At this project the problem is divided into nine pieces. Figure 4.8 shows the way it was done. The parts A to I are 128 high, but the base sized as following: A, B, D and E: 43×43; C and F: 43×42; G and H: 42×43; I 42×42. The numbers are

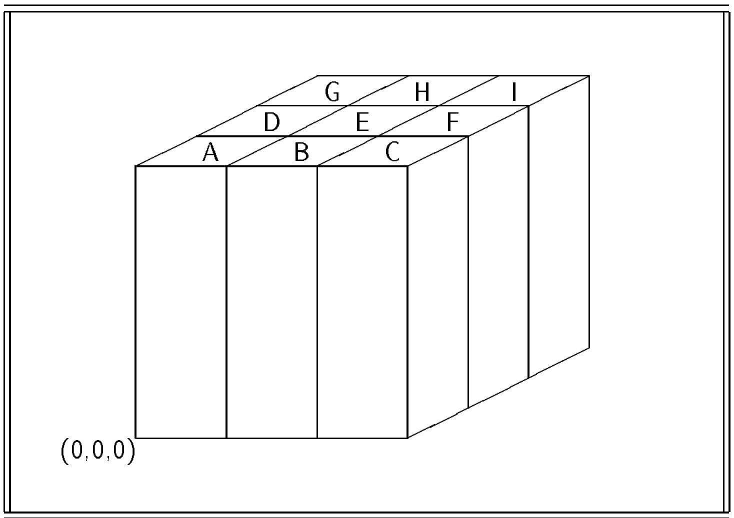

Figure 4.8: **The partitioning of the multigrid cube in this project**

calculated in the following way:

$$43 = \left\lfloor \frac{128}{3} \right\rfloor + 1 \qquad 42 = \left\lfloor \frac{128}{3} \right\rfloor$$

There is another possibility of partitioning the problem: The cube could be cut into nine layers and each layer can be dispatched to a worker. Although the number of messages would decrease, the number of bytes, which were transferred, would increase. Tests showed that this alternative took more time to solve the benchmark problem.

4.3 The Implementation of "Solving an Unstructured Sparse Linear System by the Conjugate Gradient Method"

4.3.1 Parallel Algorithm

The parallel algorithm of this benchmark has a rather fine granularity compared with the other benchmarks. The given sparse matrix is described by three vectors: A vector of nonzeros and two vectors which show the coordinates of the nonzeros. First this matrix is divided by nine, which is the number of workers, that each worker gets nearly the same number of nonzeros. Now the algorithm, described in section 3.3 on page 17, is parallelized in the following way: Each worker initializes its part of x with 1. Afterwards timing begins and the outer iteration can be started. After having computed its part of the solution vector z, the worker broadcasts it to the other workers and receives the other parts of z. Now each worker has stored the whole vector z. Thus, it can compute its part of the residual of the vector z with its part of the given sparse matrix. Then each worker sums up the squares of the members of its part of the residual vector and sends it together with the maximum of the members of its part of the vector z to the master. The master sums up the first value of the received messages and computes the square root. The result, the Euclidean Norm, is printed together with ζ. ζ is the reciprocal value of the maximum of the second values of the received messages. ζ is broadcasted to all workers, too. Each worker receives ζ from the master and the workers part of x can be recomputed for the next iteration.

In the description above the computation of the workers part of the solution vector z is missing. The conjugate gradient method must be used, which is described in section 3.3 on page 17. After initialization of the workers part of z, r and p ρ has to be computed. Therefore, each worker broadcasts the sum of the squares of each element of its part of r to all other workers. The sum over all received values is ρ. Then the workers enter the inner iteration of the benchmark. The matrix-vector multiplication $q = Ap$ needs the whole vector p. Thus, each worker broadcasts its part of p to all other workers. The next operation is $\alpha = \frac{\rho}{p^T q}$. Consequently, the product of the workers part of the vectors p and q are broadcasted to all other workers. α results from the division of ρ and the sum of all received values. $z = z + \alpha q$, $\rho_0 = \rho$ and $r = r - \alpha q$ can be performed without any communication. For $\rho = r^T r$ the sum of the squares of the members of one workers part of r is broadcasted to the other workers. The sum of the received values is ρ. Then each worker can perform the operations $\beta = \frac{\rho}{\rho_0}$ and $p = r + \beta p$ locally.

4.3.2 Characteristics and Tool Specifics in Implementation

In the case of the nine workstations the nonzeros are dispatched to the workers 1-9 as follows:

Worker	Number of nonzeros
1	205837
2	206061
3	205945
4	205877
5	205980
6	205868
7	205818
8	205799
9	205919

Each worker only gets complete rows of the sparse matrix. This is the reason why the number of nonzeros cannot be divided straight forward, since adding the next row means too many, leaving it means too few nonzeros.

Global operations are available in p4 and Express. These global operations are used in the implementation of this benchmark in the following situation: Each worker broadcasts a single value to all other workers and receives the values broadcasted by the other workers. Afterwards an operation, for example the sum or maximum, is performed on these values. In p4 and Express broadcasting and receiving messages is not necessary, only the operation has to be performed as a global operation, for example a global sum or global maximum. If the programmer is used to these global operations, the effort for writing parallel programs decreases.

In the concept of Linda there is no global operation or broadcast command. One has to use *in*, *rd* or *out* to simulate a broadcast message. In section 2.2.5 two options of sending broadcast messages are discussed. The way of broadcasting messages without using *rd* takes more than 50% more time than using *rd*. Therefore, the results in section 5.3 represents the run time of the implementation using *rd*. The need of a barrier, as described in section 2.2.5, does not cause any problems because there are three barriers when implementing the conjugate gradient method anyway.

4.4 The Implementation of the "Parallel Sort Over Small Integer"

4.4.1 Parallel Algorithm

Before explaining the parallel algorithm, it is necessary to get familiar with the sequential sort or ranking algorithm: As described in section 3.4 ranking and not sorting keys is required in this benchmark. The bucket sort is used to rank N integers in the range of 0 to B_{max}. The input of this algorithm is the array ($keys$) of the N keys to be ranked. The output is also an array ($rank$) of N integers, that contains the ranks of the keys. The rank of the i'th member of the input array can be determined as the i'th member of the output array. In this algorithm a work array (key_den) of size B_{max} is needed. It takes the following steps to rank the keys:

1. The whole work array key_den is set to zero.

2. For each key ($keys\,[i]$):
   $key\_den\,[keys\,[i]]$ is to be incremented by one.

   The result of this step is an array that contains the frequency of each key. The member of this array with the index i determines the frequency of keys with value i.

3. For i is 1 to B_{max}:
   $key\_den\,[i] \leftarrow key\_den\,[i] + key\_den\,[i-1]$

   After this step the i'th member of the array contains the sum of the frequency of keys with value i and the frequencies of the keys with value less i. As a result, the value of the last member of the array, $B_{max} - 1$, must be N, the number of keys to be ranked.

4. To compute each rank ($rank\,[i]$):
   $key\_den\,[keys\,[i]]$ is to be decremented by one.
   $rank\,[i] \leftarrow key\_den\,[keys\,[i]]$.

A key with the value i gets a rank in the range from the $i-1$st member to the i'th member of the work array. If $i = 0$ then the $i-1$st member of the work array is to be replaced by zero. An example: The 0'th member of the array is 4, the 1st member is 6 and so on. Hence, there are four keys with value 0. These keys get the rank 0 to 3. There are two keys with value 1 and their ranks are 4 and 5 and so on.

No keys are moved and no comparison is done at the bucket sort algorithm. The effort of sorting increases proportional to the number of keys, but also proportional to B_{max}. Thus, this algorithm cannot be used for sorting words or something similar.

In the parallel version of the bucket sort the N keys are distributed to the workers and each worker has to give each key a rank. Of course, this rank must be the same as all keys are ranked together on a single machine. The input of this parallel ranking is an array containing the keys distributed to the worker. The output array, which has the same size, contains the ranks of the keys. Again, each worker needs a work array *key_den* of size B_{max}. After dispatching the keys timing begins. To perform the parallel bucket sort each worker has do the following steps:

1. The whole work array *key_den* is set to zero.

2. For each key ($keys\,[i]$) held by this worker:
   Increment $key\_den\,[keys\,[i]]$ by one.

   The result of this step is an array that contains the frequency of each key held by this worker. The member of this array with the index i determines the frequency of keys with value i.

3. The work array *key_den* is divided by the number of workers.
   The members of the parts 0 to $j-1$ of *key_den* are summed up and this sum and the j'th part of *key_den* is sent to the j'th worker.

4. The sums described in step 3 and the arrays sent by the other workers are received. All received sums are summed up to *sum*.
   sum is the absolute rank of the last or one of the last keys, if the key is not unique. The least key is only the last key this worker has to deal with in this step. Absolute means, that the rank is not the rank of the keys of this worker, but it is the rank of all keys.

5. With each member of *key_den*:
   With *key_den* of each worker j:
   $sum \leftarrow sum + key\_den\,(j)\,[i]$.
   $key\_den\,(j)\,[i] \leftarrow sum$.

   ($key\_den\,(j)\,[i]$ denotes the i'th member of the j'th worker's part of *key_den*.) Like in step 3 of the sequential algorithm each member of the arrays is the sum of the members before. In difference to the sequential algorithm there is more than one array. To calculate the $i+1$st member of one array, all members with index i must be calculated first.

6. The updated parts of key_den are sent back to their original workers.

7. All updated parts are received.

8. To compute each rank $rank\,[i]$:
   $key\_den\,[keys\,[i]]$ is decremented by one.
   $rank\,[i] \leftarrow key\_den\,[keys\,[i]]$.

   This step is analogous to step 4 in the sequential algorithm, of course each worker only has to compute the ranks of its part of keys.

It is easy to see that step 3, 4, 6 and 7 are in addition to the sequential version. These steps represent the main part of the overhead. The other overhead is due to step 5, where the calculations are more extensive than in the sequential version.

There is an unused possibility of optimization in this benchmark, that seems to be implemented easily: Messages in step 3, 4 and 6, 7 can be compressed in a simple way: The greatest integer in the arrays, which are transmitted, is $N - 1 = 2^{23} - 1$. Therefore, only 23 of the 32 bits of an integer are used and one of four integers can be broken apart in three pieces, which can be hidden in the other three integers. This only means a compression of 25%, but costs of computation are low.

4.4.2 Characteristics and Differences in Implementation

The Implementation of this benchmark consists of three parts: The code of the master, of the worker and a subroutine, which performs the parallel ranking. This subroutine is called by the worker.

The benchmark is implemented on a cluster of nine RS6000, consequently, the problem is despatched to nine workers. Approximately 12 Mbytes were needed by each worker including send and receive buffers used by the tools. 16 Mbytes would be available, but AIX does not give that much memory to a single process. Hence, a worker process would swap and tool evaluation would become invalid. Therefore the number of keys were reduced from 2^{23} to 2^{21}. This does not influence the communication, because the size of the messages only depends on the range of the keys (B_{max}). Only the computations performed by each worker are reduced in the same order as keys are done.

Chapter 5

Results

In this chapter the results of the implementation of the four benchmarks are presented. Each benchmark was executed several times using each of the different tools. The execution time of each run was measured with the C-routine *gettimeofday*. The routine was called at the start of timing a first and at the end of timing a second time. The execution time resulted from the difference of the two return values.

The average of the execution times of each tool and benchmark was calculated. A summary of the average execution times can be seen in Table 5.1. In the next sections more details of implementations of the benchmarks are discussed. The last section in this chapter contains a summary.

Table 5.2, 5.4, 5.5 and 5.7 will show details of executing the implemented benchmarks. In the following each row of these tables will shortly be discussed:

$\bar{t}$ (sec): Average time of solving the benchmark

σ (sec): Standard deviation

MFLOPS: Number of floating point operations in millions per second

Messages/sec: Messages sent by each worker per second during the section of the benchmark where time is measured; the number of messages is divided by $\bar{t}$ to get the message rate

Bytes/sec: Bytes sent by each worker per second during the section of the benchmark where time is measured; the number of bytes is divided by $\bar{t}$ to get the data rate

Tool	PVM 2.x	PVM 3.x	p4	Express	Linda
Embarrassigly Parallel	584	586	589	589	586
Multigrid (128^3)	22.5	34.8	24	24.2	44.7
Conjungate Gradient	442	685	623	277	322
Integer Sort ($2^{21} : 2^{19}$)	175	208	277	175	179

Table 5.1: **Average time of execution in seconds**

 # Messages: Number of messages sent by each worker during the section of the benchmark where time is measured

Bytes: Bytes sent by each worker during the section of the benchmark where time is measured

The range of seconds in diagrams in Figures 5.2-5.4 is proportional the same in each benchmark. For example, if the average of the lowest and highest value of seconds in a diagram is twice as high as in another, the absolute range of seconds will be twice as high, too, but the relative range of seconds will be the same in both diagrams.

The columns in Figures 5.1-5.4 mean percentage of runs whose run times are less than or equal to the value below, but greater than the value of the column before. There are exception at the first and the second column. The column above the first value of seconds will always be zero, the second column contains percentage of runs, whose run times are less than or equal to its value below, but greater than or equal to the value of the first column.

As already described in section 2.2.4 there are three parameter which are used in the startup time of Express. The benchmarks, except "An Embarrassingly Parallel Benchmark" which does not need that much memory for Express message buffers, are done with the following combinations:

1. NNBUF == 2250, NBSIZE == 2000, SBSIZE == 5000
2. NNBUF == 475, NBSIZE == 9500, SBSIZE == 20000
3. NNBUF == 300, NBSIZE == 15000, SBSIZE == 31000
4. NNBUF == 150, NBSIZE == 30000, SBSIZE == 61000

61000 for SBSIZE in combination 4 is nearly the maximum size. The other values, which are assigned to the rest of the parameters, are chosen arbitrarily. The whole amount of memory that is available for Express message buffers is the product of NNBUF and NBSIZE. This is much more than needed. However, in reality AIX only allocates memory which is really used. Hence, the definition of oversized Express message buffers does not represent any problems. Sizing the Express buffers too small can cause deadlocks: If NNBUF is too small a deadlock can occur if too many messages arrive, see [6]. But also NBSIZE can be too small, especially if messages are long. Then the program gets stuck as the experience of this project shows.

The four benchmarks are implemented using the language C and where compiled and linked with the optimizing (-O) option.

5.1 The Results of "An Embarrassingly Parallel Benchmark"

In this benchmark nearly no communication has to be done. Only one message, containing 40 bytes, is sent by each worker. Therefore, the results of this benchmark

Tool	PVM 2.x	PVM 3.x	p4	Express	Linda
$\bar{t}$ (sec)	584	586	589	589	586
σ (sec)	0.4	0.4	0.1	0.6	0.4
MFLOPS	38	37.9	37.7	37.7	37.9
Messages/sec	0.0017	0.0017	0.0017	0.0017	0.0017
Bytes/sec	0.07	0.07	0.07	0.07	0.07
# Messages	1				
Bytes	40				

Table 5.2: **The results of the embarrassigly parallel benchmark**

do not differ much from tool to tool. As seen in Table 5.2 PVM 2.x is the fastest, followed by PVM 3.x, Linda, p4 and Express. A possible reason for the differences could be the number of processes that are needed by each tool. PVM needs two, Linda three, p4 four and Express five processes on each worker.

At this benchmark the Express code was executed using following message buffer sizes: NNBUF == 250, NBSIZE == 2000, SBSIZE == 5000. Because of the little communication it did not seem to be worth to test other Express message buffer combinations or Linda options. The Linda code was executed with the +kaon option. The PVM 3.x code was executed with *pvm_advise(2)*. Figure 5.1 shows the distribution of the single runs.

Table 5.3 compares the results of the benchmark implemented on the workstation cluster to other computer systems, whose results have been published by NASA [12]. In this benchmark, where speedup is linear, performance of floating point operations of the workstation cluster does not seem to be high enough. A reason is the implementation of the intrinsic functions; log and $\sqrt{\ }$ is needed in this benchmark.

5.2 The Results of "A Simple 3D Multigrid Benchmark"

As described in section 4.2.2 a smaller problem size (128^3) is used to avoid swapping. The number of operations is only an eighth of the big (256^3) problem. The number of messages sent by each worker is reduced to 86%, the number of bytes sent are only 27% of the big problem.

As it can be seen in Table 5.4 the message rate is rather high compared to the other benchmarks. In difference to the other benchmarks each worker sends its messages to certain workers and not to all workers. This and the high message rate seem to be the reason why Linda shows weakness in performance solving this benchmark. PVM 2.x, p4 and Express took rather the same time for solving the problem. PVM 3.x needed more time than PVM 2.x, hence, the new release means a decrease in performance concerning this benchmark.

The best combination of buffers in Express was combination 3: NNBUF == 300,

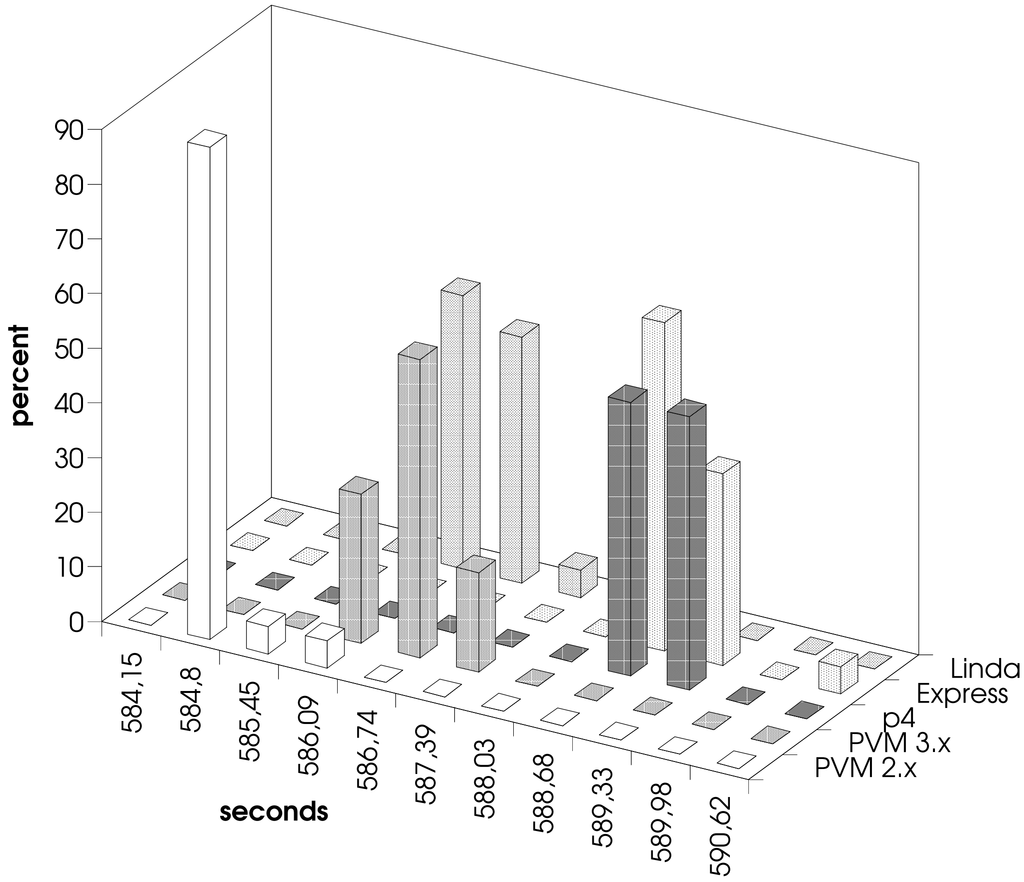

Figure 5.1: **Distribution of the single runs in the embarrassingly parallel benchmark**

Computer System	Problem Size	No. Proc.	Time (sec.)	Ratio to Y-MP/1
Y-MP	2^{28}	1	126.2	1.00
		8	15.9	7.95
Y-MP EL		1	550.5	0.23
		4	141.2	0.89
iSPC/860		32	102.7	1.23
		64	51.4	2.46
CM-5		16	42.4	2.98
		32	21.5	5.88
MP-1		4K	248.0	0.51
MP-2		16K	22.4	5.63
KSR-1		32	69.8	1.81
		64	34.9	3.62
RS6000		9	584	0.22

Table 5.3: **The embarrassingly parallel benchmark: Results of other computer systems in comparison to the RS6000 cluster using PVM 2.x**

NBSIZE == 15000, SBSIZE == 31000, but executions using combination 2 took about the same time as executions using combination 3. Linda's best combination of options are: +kaon -high +redirect. A bigger -udp size did not affect the run time. In this benchmark the better results, executing the PVM 3.x code, were obtained using *pvm_advise(2)* rather than *pvm_advise(3)*. Figure 5.2 shows the distribution of the single runs.

Tool	PVM 2.x	PVM 3.x	p4	Express	Linda
$\bar{t}$ (sec)	22.5	34.8	24	24.2	44.7
σ (sec)	0.09	1.5	0.4	1.05	1.32
MFLOPS[1]	44.6	28.8	41.7	41.5	22.4
Messages/sec	22.3	14.4	20.9	20.7	11.2
Kbytes/sec	162.5	105.1	152.4	151.1	81.8
# Messages	501				
Mbytes	3.57				

Table 5.4: **The results of the multigrid** (128^3) **benchmark**

[1] As described in section 4.2.2 the official and not the real operation count was used to calculate MFLOPS

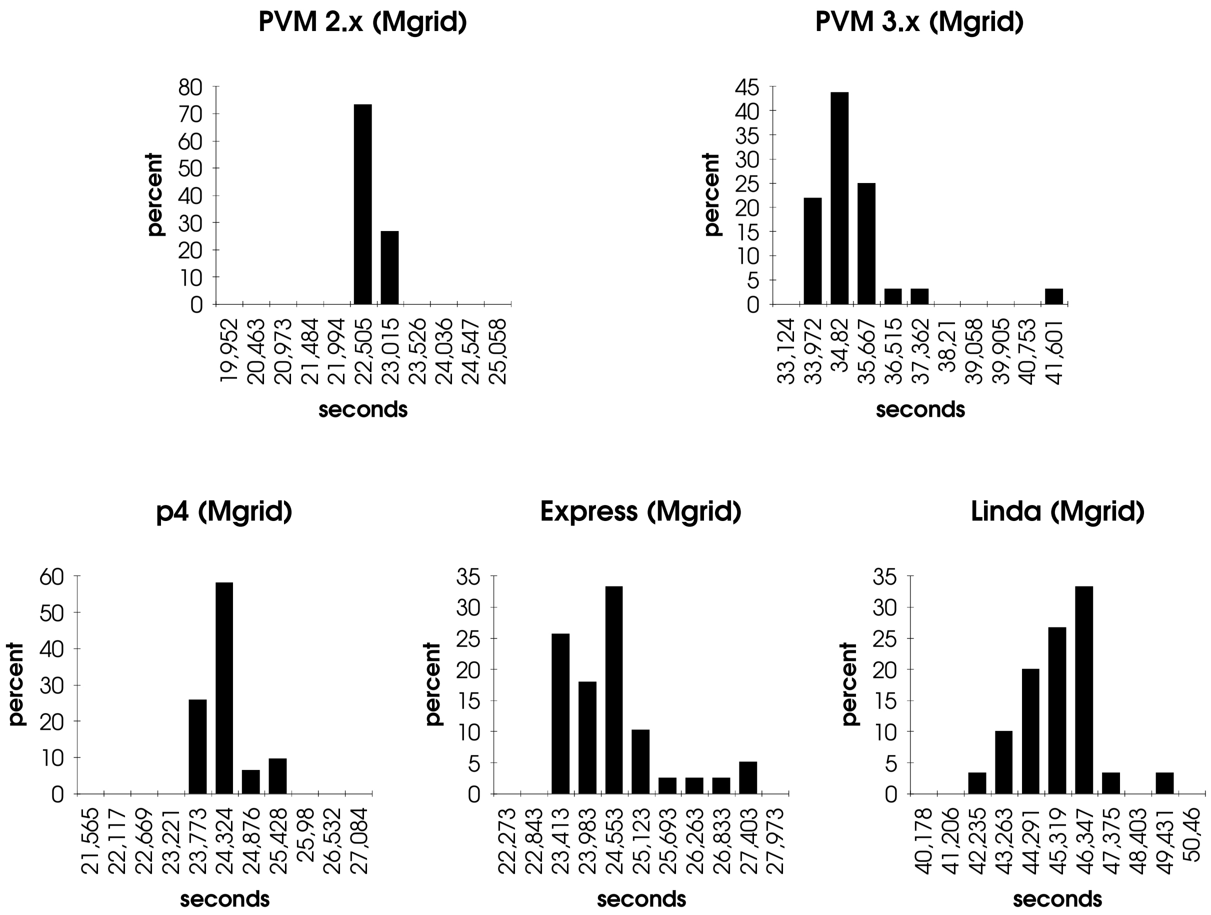

Figure 5.2: **Distribution of the single runs in the multigrid benchmark**

Tool	PVM 2.x	PVM 3.x	p4	Express	Linda
$\bar{t}$ (sec)	442	685	623	277	320
σ (sec)	21.6	10.4	10.9	0.7	2.9
MFLOPS	3.4	2.2	2.4	5.4	4.7
messages/sec	2.65	1.71	1.88	4.23	3.66
Kbytes/sec	10.7	6.9	7.6	17	14.7
# messages	1170				
Mbytes	4.6				

Table 5.5: **The results of the conjungate gradient benchmark**

5.3 The Results of "Solving an Unstructured Sparse Linear System by the Conjugate Gradient Method"

In this benchmark almost only broadcast messages are used. This seems to be the reason for the few Mflops, the low message and data rate. Table 5.5 shows results of five tools. In this benchmark PVM 3.x and p4 do not seem to be efficient. Express is more than two times faster, followed by Linda, which also has a good result.

In the beginning PVM 3.x, p4 and Express could not run this benchmark. Programs got stuck when they were executed. First PVM 3.1.4 was stable enough for a valid execution. p4 (see section 2.2.3) needs some patches for version 1.3 and Express bugs (see section 2.2.4) also had to be fixed.

In this benchmark the following combination 2 of Express buffers turned out to be the best: NNBUF == 475, NBSIZE == 9500, SBSIZE == 20000. Combination 3 seems nearly to be equivalent. Linda's best options are: +kaon +bcast +high +redirect. Especially the option +bcast seems to be important, because all runs using -bcast (and -bcastcache) are a lot slower. *pvm_advise(3)* was used to execute the PVM 3.x code, *pvm_advise(2)* produced worse results. Figure 5.3 shows the distributions of the single runs.

The memory requirements are low in the Conjugate Gradient benchmark, therefore, the big problem size was used. A comparison to other computer systems is possible. Table 5.6 shows the results of other computer systems, which were published by NASA [12].

5.4 The Results of "Parallel Sort Over Small Integer"

As described in section 4.4.2, full problem size makes tool evaluation invalid. Consequently, the number of keys to be ranked are reduced. This does not influence the size or number of the messages. Only the computations each worker has to do are

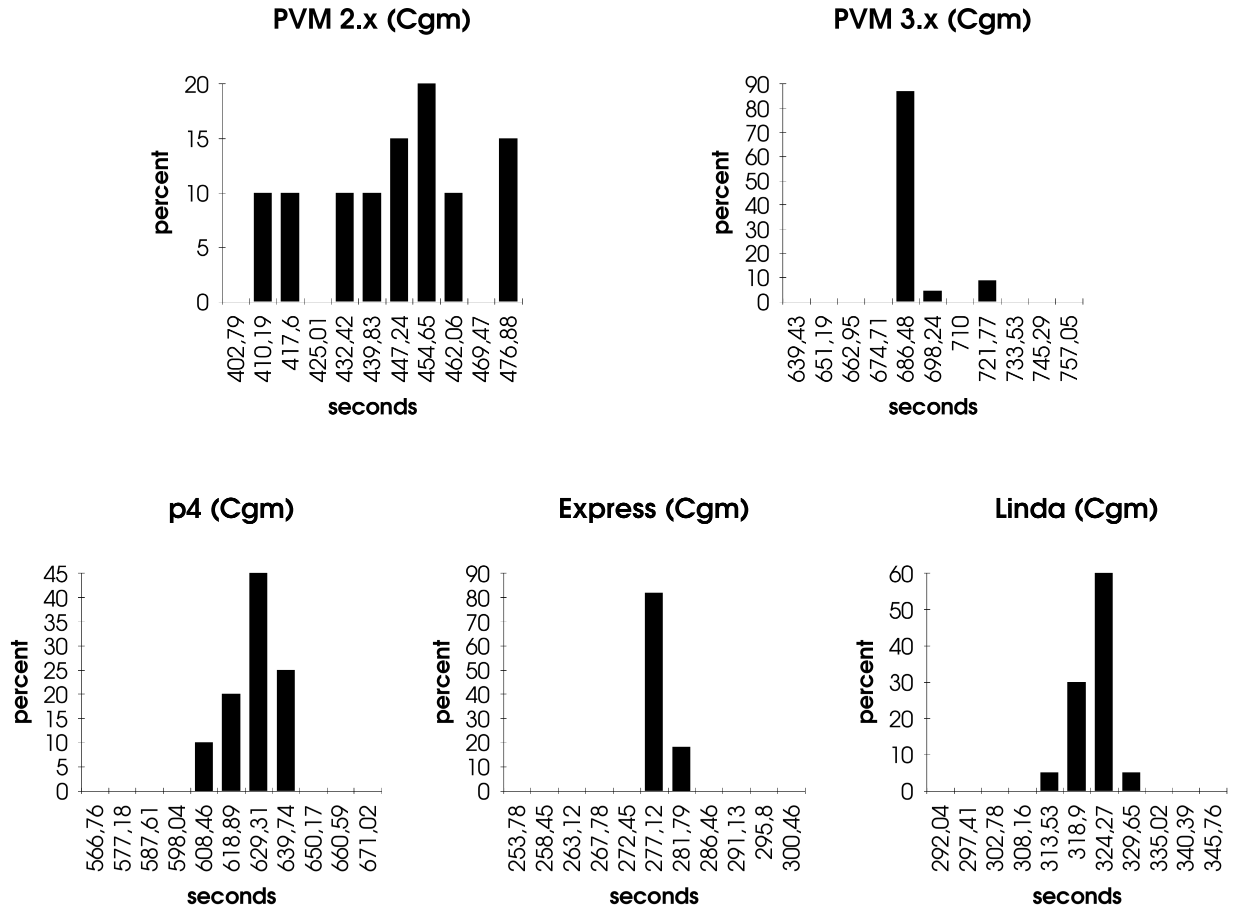

Figure 5.3: **Distribution of the single runs in the conjugate gradient benchmark**

Computer System	Problem Size	No. Proc.	Time (sec.)	Ratio to Y-MP/1
Y-MP	2×10^6	1	11.92	1.00
		8	2.38	5.01
Y-MP EL		1	65.4	0.18
		4	23.9	0.50
iSPC/860		128	8.6	1.38
MP-1		4K	64.5	0.18
MP-2		16K	11.0	1.08
KSR-1		32	21.7	0.55
RS6000		9	442	0.03

Table 5.6: **The conjungate gradient benchmark: Results of other computer systems in comparison to the RS6000 cluster using PVM 2.x**

Tool	PVM 2.x	PVM 3.x	p4	Express	Linda
$\bar{t}$ (sec)	175	208	277	175	179
σ (sec)	0.5	1.4	0.4	0.1	0.8
Messages/sec	0.9	0.8	0.6	0.9	0.9
Kbytes/sec	208	175	131	208	204
# Messages	160				
Mbytes	35.6				

Table 5.7: **The results of the integer sort** ($N = 2^{21}, B_{max} = 2^{19}$) **benchmark**

decreased.

As seen in Table 5.7 the average times of execution of PVM 2.x, Express and Linda are rather the same. The execution of the p4 code takes a lot of time. The execution time of PVM 3.x is between p4 and the others.

There were problems with p4 and Express in the beginning and they could not run this benchmark, the programs got stuck at execution. The originators had to fix the bugs which caused these problems. Afterwards they could run this benchmark.

At this benchmark the combination 3 of the Express message buffers (NNBUF == 300, NBSIZE == 15000, SBSIZE == 31000) seems to be the best. The Linda code is executed with following options: +kaon -high, ±redirect and another udp-size do not have much influence on the results. In PVM 3.x there was a big difference in using *pvm_advise(2)* to *pvm_advise(3)*. *pvm_advise(3)* was much faster at this benchmark. Figure 5.4 shows the distribution of the single runs.

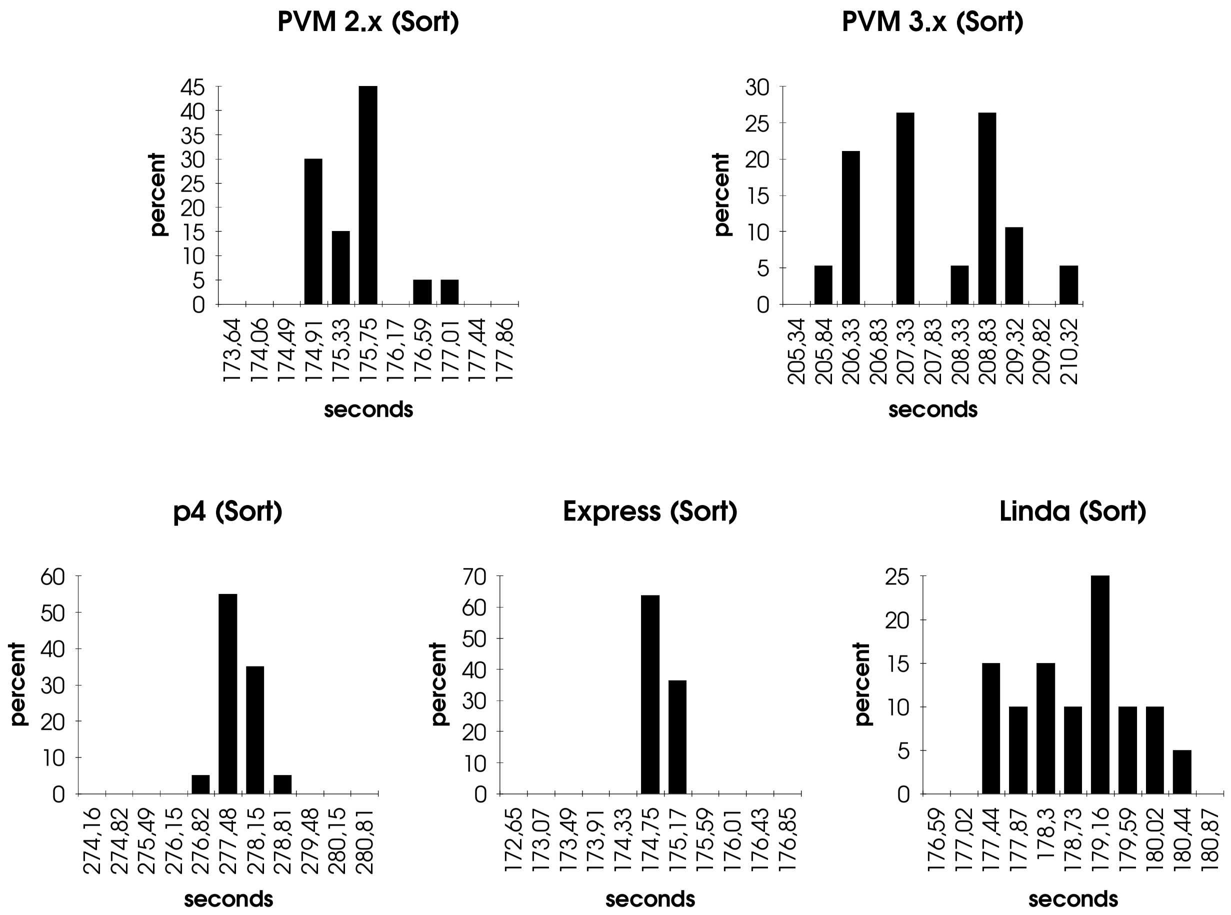

Figure 5.4: **Distribution of the single runs in the integer sort benchmark**

5.5 Summary of the Results of the Implemented Benchmarks

Express shows good results in all implemented benchmarks. The results of PVM 2.x are the best except of the "Conjugate Gradient" benchmark. Linda is also quite good, but the result of the "Multigrid" benchmark is the worst of all evaluated tools. p4 and PVM 3.x seem to be the least efficient tools. On the one hand p4 has one of the best results at the "Multigrid" benchmark, on the other hand p4 produces the worst result of the "Integer Sort" benchmark. Indeed p4 starts up communication on the fly that means that connections between worker processes are built not before they are needed. Therefore, the building up of connections happens during timing, but especially in the "Integer Sort" benchmark the connections are established after each worker has sent the first eight messages. Thus, there must be other reasons for the bad result.

For ranking of the tools, which is done in Table 5.8, nt_k, which is the sum of the norm times $nt_{k,b}$, where $k \in \{\text{PVM 2.x}, \text{PVM 3.x}, \text{p4}, \text{Express}, \text{Linda}\}$ and $b \in \{\text{embar}, \text{mgrid}, \text{cgm}, \text{sort}\}$, is computed for each tool:

$$nt_{k,b} = \frac{t_{k,b}}{\min_k \{t_{k,b}\}} \qquad nt_k = \sum_b nt_{k,b} \quad ,$$

where $t_{k,b}$ is the average time of execution. The other entries in Table 5.8 are personal judgments of the author. "Configuration of the tool" means the effort for configuration, which is necessary by each user before using the tool the first time. "Startup and finishup time" is the time, which is needed by the program each time of execution. This is particularly important during debugging and testing, where the program has to be executed several times. "Handling and program writing" includes the effort and difficulties of using commands and starting the program. Each tool has a debugger but sometimes the debugger can be inappropriate. "Debugging without debugger" tells how much the tools supports debugging without debugger. This can also be interesting if the error seems too easy to detect for learning or even starting the debugger. "Easy to learn" addresses the effort to learn the tool.

Tool	Express	PVM 2.x	Linda	p4	PVM 3.x
Rank	1	2	3	4	5
$nt_{k,\mathrm{embar}}$	1.009	1	1.003	1.009	1.003
$nt_{k,\mathrm{mgrid}}$	1.076	1	1.987	1.067	1.547
$nt_{k,\mathrm{cgm}}$	1	1.596	1.162	2.249	2.473
$nt_{k,\mathrm{sort}}$	1	1	1.023	1.583	1.189
nt_k	4.085	4.596	5.175	5.908	6.2212
Configuration of the tool	$\circ$	$+$	$+$	$\surd$	$+$
Startup and finishup time	$\circ$	$+$	$\circ$	$-$	$+$
Handling and prog. writing	$\circ$	$+$	$+$	$\circ$	$\surd$
Debugging without debugger	$\circ$	$\surd$	$\circ$	$+$	$\surd$
easy to learn	$-$	$\surd$	$+$	$\circ$	$\circ$

$+$: very convenient, $\surd$: convenient, $\circ$: satisfactory, $-$: inconvenient

Table 5.8: **Final ranking of the tools**

Bibliography

[1] D. Bailey, J.Barton, T. Lasinski, H. Simon,
The NAS Parallel Benchmarks,
Report RNR-91-002 Revision 2, Numerical Aerodynamic Simulation (NAS) Systems Division, NASA Ames Research Center, Moffett Field
August 1991

[2] Jaswinder Pal Singh, Wolf-Dietrich Weber and Anoop Gupta,
SPLASH: Stanford Parallel Applications for Shared-Memory,
Computer Systems Laboratory, Stanford University, April 1991.

[3] A. Beguelin, J. Dongarra, Al Geist, R. Manchek, V. Sunderam,
A USERS' GUIDE TO PVM PARALLEL VIRTUAL MACHINE,
ORNL/TM-11826, Oak Ridge National Laboratory, Dec 1991.

[4] A. Beguelin, J. Dongarra, Al Geist, W. Jiang, R. Manchek, V. Sunderam,
PVM 3 USER'S GUIDE AND REFERENCE MANUAL,
ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[5] R. Butler, E. Lusk,
User's Guide to the p4 Programming System,
ANL-92/17, ARGONNE NATIONAL LABORATORY, Oct 1992.

[6] Parasoft Corporation,
Express C: User's Guide 3.0, Reference 3.0, Release Notes 3.2,
Pasadena, 1992.

[7] Scientific Computing Associates Inc.,
C-Linda User's Guide & Reference Manual,
New Haven, 1992.

[8] A. L. Beguelin,
Xab: A Tool for Monitoring PVM Programs,
Pittsburgh Supercomputer Center, Carnegie Mellon University, Pittsburgh.

[9] V. Herrarte, E. Lusk,
Studying Parallel Behavior with Upshot.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browing, R. L. Carter, L. Dagnum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon,

V. Venkatakrishnan and S. K. Weeratunga,
The NAS Parallel Benchmarks - Summary and Preliminary Results,
Proceedings of Supercomputing 91, pp. 158 - 173, Nov 1991.

[11] High Performance Fortran Forum,
High Performance Fortran Language Specification Version 1.0,
Rice University, Houston Texas, May 1993.

[12] D. H. Bailey, E. Barszcz, L. Dagum and H. D. Simon,
NAS Parallel Benchmark Results,
RNR Technical Report RNR-92-002, Dec 1992

List of Figures

List of Tables