INSTITUT FÜR THEORETISCHE TEILCHENPHYSIK
INSTITUT FÜR ANWENDUNGEN DER INFORMATIK
UNIVERSITÄT KARLSRUHE (TH)

TTP99-16
April 1999

Milestone No. 1

zur Forschungsvereinbarung DE-98008 im Rahmen der Europäischen Forschungsförderungsprogrammes von Digital Equipment

zwischen

Digital Equipment International Limited (DEIL)
1 Grand Places
1700 Fribourg
SCHWEIZ

und

Institut für Theoretische Teilchenphysik
Universität Karlsruhe
D-76128 Karlsruhe
Bundesrepublik Deutschland

Contents

Chapter 1

Introduction

The first step of the project was the installation and set-up of a workstation-cluster consisting of 8 DEC α Workstations (600 MHz) connected by Myricom network-hardware and the Parastation II software provided by the Institut für Programmstrukturen und Datenorganistation (Universität Karlsruhe).

Due to unexpected inconsistencies of the hardware configuration and instabilities of the software we had to wait for modifications of the supplying companies and institutes. This has caused a shift of the actual start of the project to September 1998. It also resulted in exhaustive tests and verifications of the performance as well as the stability of the various soft- and hardware components of the cluster that are still ongoing and are part of this report.

As stated in the contract one of the main goals is an extensive parallelization of the algebraic program FORM. Another (smaller) project is the parallelization of a program for processing symmetric patterns. Details plus results of these two activities will be given in the following report.

Chapter 2

Evaluation of Different Hard– and Software-Platforms

The implementation of the parallel programs is based on the message passing libraries MPI(CH) [3] and PVM [4] in order to garuantee portability. Both message passing libraries can make use of specialized device drivers to yield maximum performance. During the first part of our project the following parallel platforms served as testbeds:

- DEC α Workstation cluster running DEC UNIX 4.0D,
  8 processors, 600MHz, MPI(CH) & PVM.
- IBM RS6000 SP2 running AIX 4.2.1,
  168 (in total 256) processors, 120MHz, MPI only.

2.1 The DEC α / ParaStation II Cluster

In the case of the DEC α cluster different network hardware, different IP drivers and different implementations of device drivers for the messages libraries exist and have been used:

- DEC Fast Ethernet (100 MBit/s), IP driver, MPICH.
- DEC Gigabit Ethernet (1 GBit/s), IP driver, MPICH.
- Myrinet (1.26 GBit/s)[5], IP driver, MPICH.
- Myrinet (1.26 GBit/s), ParaStation II[6] software, MPICH.

For a thourough understanding of the parallel systems' behaviour it is of course crucial to compare their performance. For the IP drivers the TCP performance was determined. Moreover we measured the bandwidth and latency performance for the MPI(CH) and PVM libraries under different conditions, which gives a useful impression of the systems stability. It turned out that the (freely available) PVM implementations tend to be unstable. We therefore present the MPI(CH) results only.

2.2 TCP Performance

The TCP (Transfer Control Protocol) stream (data transfer) performance of the IP (Internet Protocol) drivers for the Fast Ethernet, the Gigabit Ethernet and the Myrinet on the DEC α cluster was measured with the standard package netperf [7]. It provides a test script that repeats the measurement until a certain level of accuracy is reached. A couple of parameters can be chosen, namely the buffer sizes on both the sending & receiving side and the size of the transmitted packages. The results are shown in figure 2.1 for send and receive buffer sizes of 56kB. In the case of the Gigabit Ethernet the measurement was done both with standard kernel TCP parameters and tuned (i.e. maximum value) kernel TCP parameters.
It is obvious that the theoretical throughput of 100 MBit/s and 1 GBit/s respectivly is not reached because of the software overhead caused by the IP driver. As for the 100 MBit/s Ethernet a maximum of about 40 Mbit/s can be reached, both the Gigabit Ethernet and the Myrinet give a maximum throughput of approximately 250 Mbit/s only.

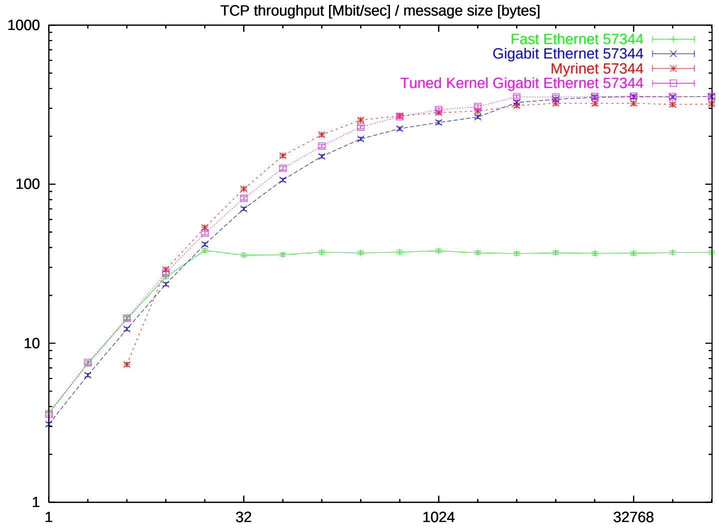

Figure 2.1: TCP performance of different IP drivers on the DEC α cluster. Both the send and receive buffer sizes are 56kB.

2.3 MPI Performance

The performance of the MPI(CH) libraries was examined using the Pallas MPI benchmarks (PMB) [8]. Again a script is provided that can be used to measure the throughput and latency of the messages passing operations under different circumstances without further interaction. We present the results of a subset of three basic benchmarks, which are sufficient for our purpose. The plots contain bandwidth and latency curves for all MPI implementations we have used.

The first benchmark is the simplest possible single transfer benchmark, a ping-pong test based on the blocking MPI routines `MPI_send` and `MPI_recv`. One process(or) sends a message of n bytes to another process(or), which immediately sends that message back. In contrast to the other benchmarks there is no concurrency with other messages passing activity during this test. In other words the bandwidth and latency are measured under optimal conditions.

As can be seen from figure 2.2 the bandwidth increases as the message size does. For large message sizes there is a saturation effect. On the DEC α cluster the performance of the different IP drivers directly translates to the MPI(CH) performance. The MPI(CH) layer adds overhead that has no dramatic effect on the Fast Ethernet throughput, but lets the bandwidth of both the Myrinet and the Gigabit Ethernet drop below 200 MBit/s. The ParaStation II software provides a special device driver for MPI(CH) on the Myrinet hardware that does not use the IP protocol and adds much less protocol overhead. With this spezialized software a bandwidth of about 300 MBit/s is achieved. The IBM SP2 reaches a maximum MPI bandwidth of 800 MBit/s.

In figure 2.3 the measured MPI(CH) latency for the ping-pong benchmark is shown. The latency ranges from a few microseconds for small messages sizes up to a second for large message sizes. Using MPI(CH) over the IP protocol results in a minimum latency of more than 200 microseconds, a number that is determined by the protocol overhead. Lower latencies can be achieved with special device drivers only. It turns out that the minimum latency on the ParaStation II (about 20 microseconds) is even smaller than on the IBM SP. Note that for our particular interest the latency at larger message sizes is most important. In this region again the IBM SP2 provides optimal performance.

The following benchmarks represent parallel transfer benchmarks. Again they are run with varying messages length of n bytes. Timings are averaged over multiple samples and (additionally) over the different processes running. In the send-receive benchmark (based on the MPI routine `MPI_sendrecv`) each process(or) in a periodic chain of N processes/ors sends a messages of n bytes to one of its neighbours. The exchange benchmarks

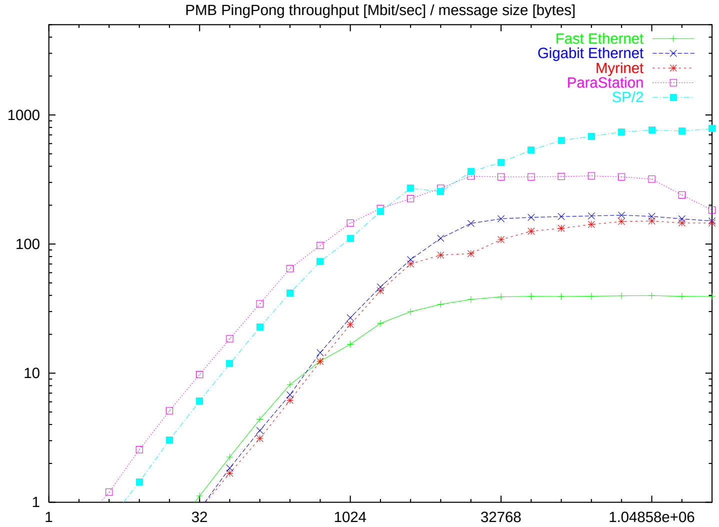

Figure 2.2: MPI(CH) bandwidth results for the PingPong benchmark

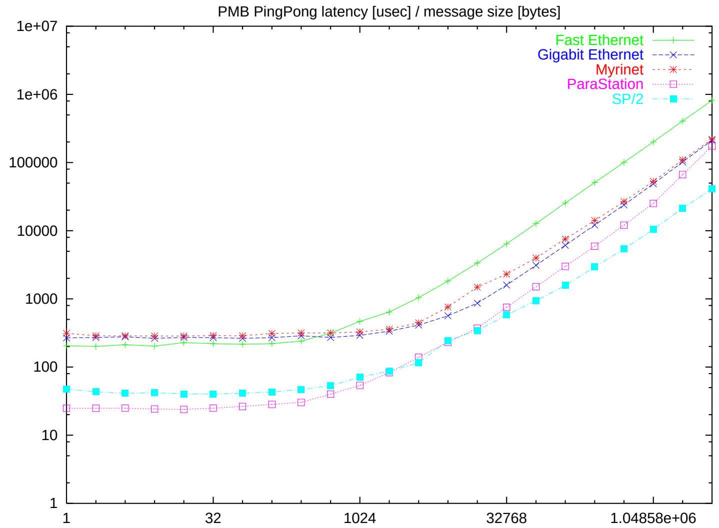

Figure 2.3: MPI(CH) latency results for the PingPong benchmark

use the non-blocking send operation `MPI_Isend` together with a blocking receive operation (`MPI_recv`). The messages are sent in both directions of a periodic chain. In this way a network under activity (i.e. non-optimal conditions) is simulated and the maximum performance will most probably not be reached any more. In the figures 2.4-2.7 the results for $N = 2$ and $N = 4$ are shown. Under heavier traffic the IBM SP2 shows a constant bandwidth and latency whereas the throughput on the ParaStation II decreases and the latency increases considerably. The Pallas MPI benchmarks also provide tests of the collective communication functions in MPI. The results are not presented, because no collective communication is used in the parallelization of FORM at the moment.

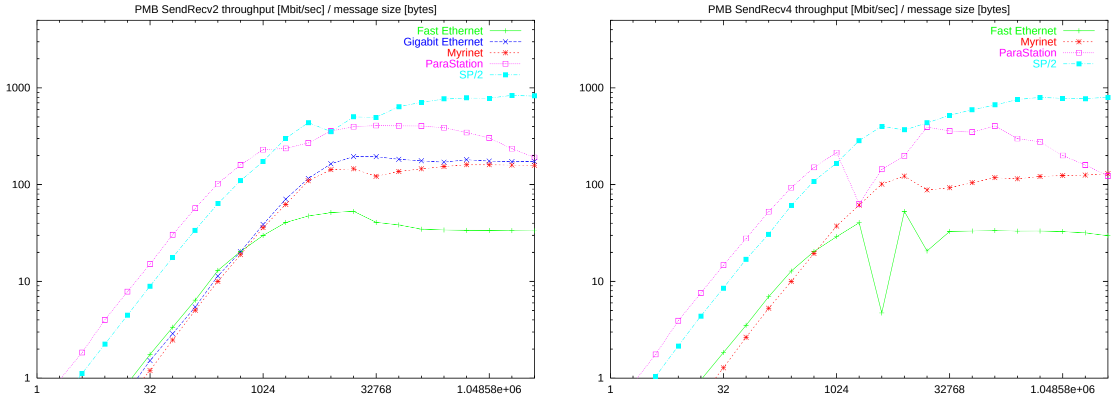

Figure 2.4: MPI(CH) bandwidth results for the SendRecv benchmark (left: 2 processes, right: 4 processes)

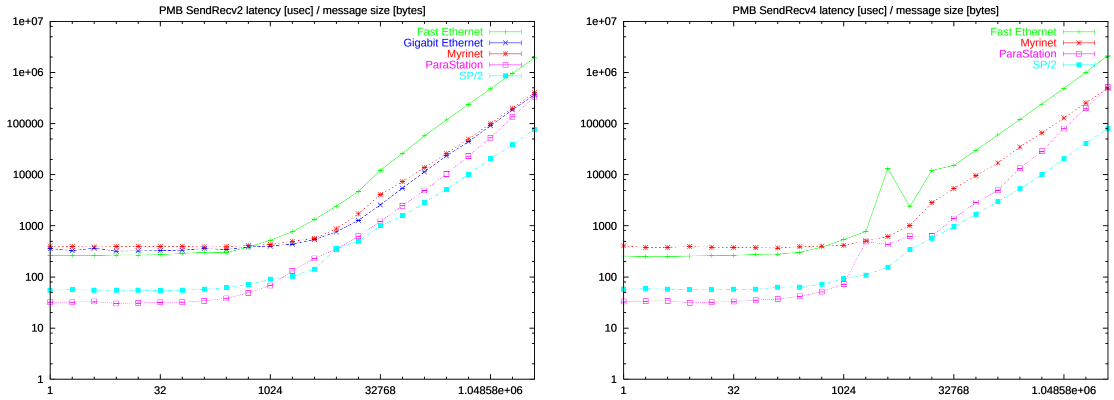

Figure 2.5: MPI(CH) latency results for the SendRecv benchmark (left: 2 processes, right: 4 processes)

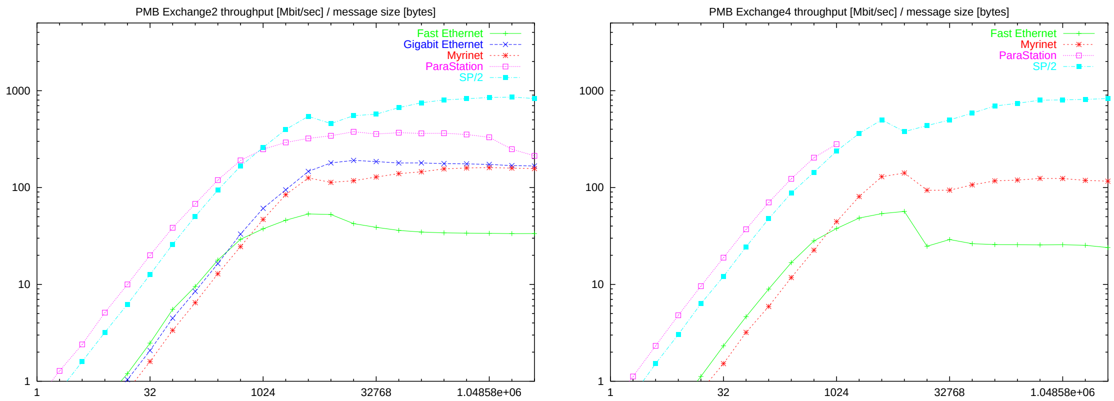

Figure 2.6: MPI(CH) bandwidth results for the Exchange benchmark (left: 2 processes, right: 4 processes)

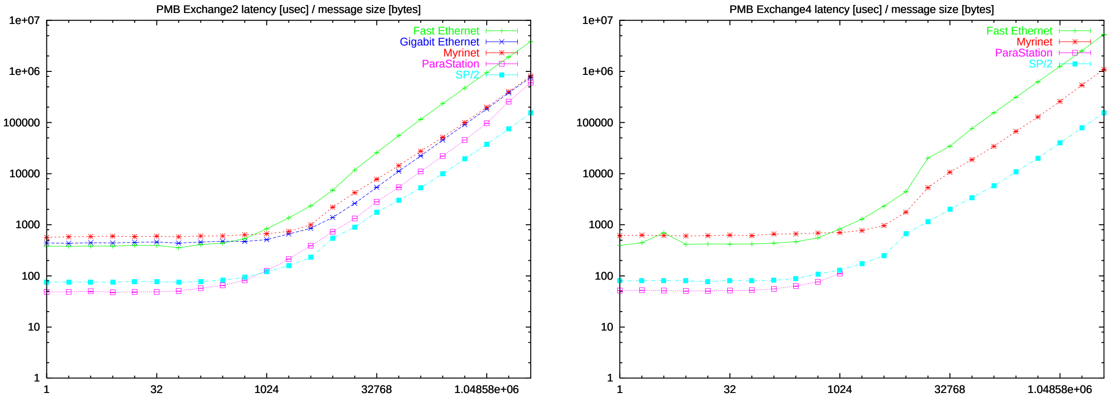

Figure 2.7: MPI(CH) latency results for the Exchange benchmark (left: 2 processes, right: 4 processes)

Chapter 3

Parallelizing the Symbolic Manipulation Program FORM

3.1 Introduction to FORM

FORM [1] is a program for symbolic manipulation of algebraic expressions specialized to handle very large expressions of millions of terms in an efficient and reliable way. Although it supplies a programming language that allows the formulation of a wide area of symbolic algorithms it is mainly used for computations in high energy physics, for which it supplies some special data types like Dirac matrices. Especially in the field of perturbative calculations of higher order corrections to quantum field theoretical quantities FORM has become a de-facto standard.

3.2 The Sequential Version of FORM

3.2.1 The Mechanisms behind the Sequential Version

Since FORM is designed for solving large problems, it is designed to be used in an non-interactive way, by supplying a program that will be executed and the result and typically also some statistics are printed to screen or redirected to a file. A very simple example program reads:

```
Symbols x,a,b;

Local E = x^3;

id x = a + b;
print;
.end
```

FORM is a type-oriented language, so that a program starts with the declaration of variables that are to be used, in this case the symbols a and b. Then the expression(s) that should be dealt with have to be defined, here E is defined as a local expression. In FORM an expression is a sum (sequence) of single terms, which is always completely expanded to yield a unique representation. The rest of a FORM program contains statements that state what is to be done with an expression, in this case the identification (replacement) of all occurences of x in E as the sum of a and b and to print the expression. A program is divided into so called modules that are terminated with "dot"-instructions that cause the execution of the module. The example above is very simple one: it consists of only one module and will produce the following output:

```
    FORM version 3.-1(Nov 30 1998). Run at: Thu Feb  4 14:59:51 1999
    Symbols x,a,b;

    Local E = x^3;

    id x = a + b;
    print;
    .end

Time =       0.00 sec    Generated terms =          4
                E        Terms in output =          4
                         Bytes used      =         68
```

```
E =
   3*a*b^2 + 3*a^2*b + a^3 + b^3;
```

FORM runs a program module by module. The execution of each module is divided into three parts:

- **Compilation:** the textfile is translated into internal code.
- **Generation:** for each term of the input expressions the statements of the module are executed. In general a lot of terms are generated for each input term.
- **Sorting:** all the output terms that have been generated must be sorted and equivalent terms have to be summed up.

The special properties of FORM result from the fact that it only allows local operations on single terms, like replacing parts of it or multiplying something to it. Together with a sophisticated pattern matcher this seemingly strong limitation allows the formulation of general and efficient algorithms. The limitation to local operations makes it possible to handle expressions as "streams" of terms, that can be read in sequentially from a file and be worked on one at a time. The generation of terms is done in a way that the output terms drop out term by term also. These output terms are stored in two intermediate buffers and a temporary sortfile in a staged procedure: When the small buffer gets full, its contents gets sorted and is copied to the larger buffer, freeing the small buffer for the next patch of output terms. If the large buffer gets full, its contents gets again sorted and is copied to a temporary file, freeing the large buffer. At the end of the module all of the now existing sorted patches sitting in the two buffers and the sortfile have to be merged stage by stage to one single sorted output stream of terms which is written to a file and used as input source of terms for the next module.

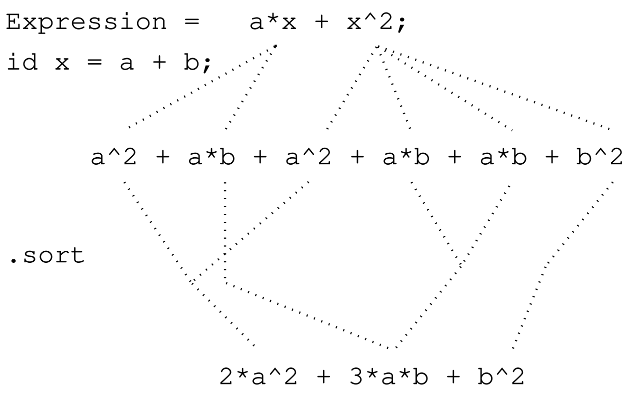

Figure 3.1: The expansion tree of a simple module

For the sorting of the first buffer, the small buffer, a slightly modified merge sort algorithm is used, merging sorted patches of terms in the other stages of sorting is done with a tree of losers [2]. This results in a tree-like structure for the generation of terms as well as for the sorting, as shown in figure 3.1.

3.2.2 The 64-Bit Sequential Version of FORM on DEC UNIX

Before any effort was made to parallelize FORM, the first step was to analyze and optimize the sequential code. FORM was ported to DEC UNIX and the the internal used word length was changed from 16 bits (which results in 32-bit arithmetic operations) to 32 bits (64-bit arithmetics), thereby using the full register width of the α-processors. Afterwards this new 64-bit version of FORM 3 was thouroghly tested and the performance was compared to the the old 32-bit FORM 2.3. The speedup is almost a factor 2 for small problems; for a large size realistic application (the calculation of a quite complicated 3-loop diagram) the run time dropped from 6665 seconds to 4105 seconds by using the larger word size only. Another speedup of about 1.5 was achieved simply by experimenting with the DEC C compiler optimizations.

3.2.3 Performance Analysis of the Sequential Version

The next step before parallelization was profiling FORM in typical applications in order to determine the functions that consume most of the run time. The following is the output of the Digital Unix profiling tool prof when running a yet not very large FORM application program. It shows that the sequential program has indeed been very well optimized within the over 10 years of its existence and the CPU usage is well leveled between functions involved in the generating (marked with a G) and in the sorting of terms (S).

```
%time     seconds  cum %   cum sec  procedure

 20.2    501.6201   20.2    501.62 Normalize           G
 13.4    331.3018   33.6    832.92 Compare             S
  8.5    210.6748   42.1   1043.60 PutOut              S
  5.7    141.9512   47.8   1185.55 StoreTerm           S
  3.4     83.7432   51.2   1269.29 TestSub             G
  3.0     73.2539   54.1   1342.54 MergePatches        S
  2.9     72.4668   57.0   1415.01 InsertTerm          G
  0.9     21.7793   87.5   2171.72 EndSort             S
```

The time needed for compiling the program text into the FORM-internal representation is not dependent on the size of the problem. In realistic applications it is usually neglectible.

3.3 A Prototype for a Parallel Version of FORM

As has been already mentioned, FORM is specialized in working with very large expressions, typical jobs run for weeks and can need Gigabytes of temporary diskspace. So the idea of parallelizing FORM is obvious and interesting not only because distributing the CPU usage would speed the program up but also because a distribution of the large temporary data could make even larger problems accessable.

One of the suppositions for the parallelization of FORM has been not to limit the approach by using a too specialized hardware but instead to use standard message passing libraries (MPI(CH) [3] and PVM [4]) for the parallelization that are available on a wide class of architectures and should be portable to new and more powerful systems. Since there are a lot of well tested and useful algebraic packages written in FORM, another aim is to allow this code to be reused with no or minor changes only.

The limitation of performing only local operations makes FORM very well suited for parallelization and the concept of parallization is straightforward: distribute the input terms among the available processors, let each of them perform the local operations on its input terms and generate and sort the arising output terms. At the end of a module the sorted streams of terms from all processors have to be merged to one final output stream again. The compiling of the program-text to the internal representation was considered to not be worth parallelizing. This concept indicates to using a master-slave structure for the parallelization, where the master would store the expressions and distribute and recollect all the terms of each expression. For the implementation of this raw concept we used a four step strategy:

- one process(or) generates terms, a second process(or) sorts the output terms
- instead of only one process arbitrary many processes perform the sorting.
- the input terms are distributed and the term generation is also done in parallel
- final optimizations: avoid or handle worst cases, load leveling, fault tolerance

This approach has several advantages, the most important being that having working versions in every stage gives us a good idea of how good the parallelization is and the possibility of realistic tests even at a very early stage.

3.3.1 The 2-Processor Model

This first step of the parallelization turned out to be very useful, since it not only gave a deep insight of how changes to the source code of FORM have to be made without affecting the efficiency of the well optimized sequential code. It also served as a check of whether and how the concept could or could not lead to a decent speedup. It could be shown that for parallelizing software on a cluster of very fast workstations the importance of avoiding communication overhead can not be overstated, especially concerning the latency of network-communications. It was clear after these experiments that all the communication had to be done in a buffered way, since sending around single terms increased the run times of the two-processor code up to a factor 20. With the buffered version the runtime could be limited to about 1.5 times of that of the sequential code with two workstation connected by a 10 Mbit/s ethernet using the PVM and MPI(CH) libraries. This was considered to be fast enough since in this case due to only minimal workload overlap we merely were adding the communication overhead. This version was also used to learn and optimize all the options and possibilities of the two supported message passing libraries MPI(CH) and PVM.

3.3.2 Parallel Sorting

The second step was to distribute the output terms among arbitrary many processors and do the sorting in parallel. Since this part of the sorting relies strongly on communication between the processors, it most probably sets the limits of parallel speedup.

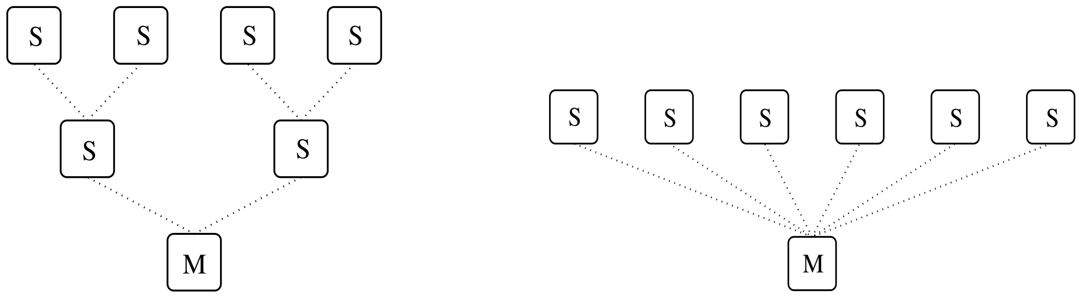

Figure 3.2: Two trees for merging terms from different processors

A first try was to map the "tree of losers" used in FORM to merge sorted patches onto the processors as showed in figure 3.2 left. While it would distribute the workload in an optimal way this approach adds too much communication overhead.

This is why in the end a much simpler communication structure was used (figure 3.2 right), where all slaves send their sorted terms to the master process and this process uses a local "tree of losers" to merge the output streams of the slave-processes. Additional effort was made to overlap the work on the master process with the sorting done on the slaves, which caused a much deeper interference with the sequential code, but finally resulted in a very fast and stable implementation that was tested not only on our own parallel cluster using different network soft/hardware but also on the IBM SP2 parallel computer of the Rechenzentrum der Universität Karlsruhe.

Of course the scaling of this approach with a large number of processors is a possible limitation. Tests on the 256 node IBM SP2 parallel computer showed that, at least for such a specialized network-hardware, the performance does not dramatically break down up to 32 processors. If problems should occur with more processors or a different architecture, an intermediate layer of "foremen" could be used to circumvent this possible bottleneck.

Since we are only testing the sorting, any problem that produces a sufficient number of intermediate terms can be used to test the parallel speedup. We used a very simple program, that expanded the expression $(a_1 + \cdots + a_n)^2$ and then replaced a_1 by $-(a_4 + \cdots + a_n)$ which results in a short, easy to check result and, by

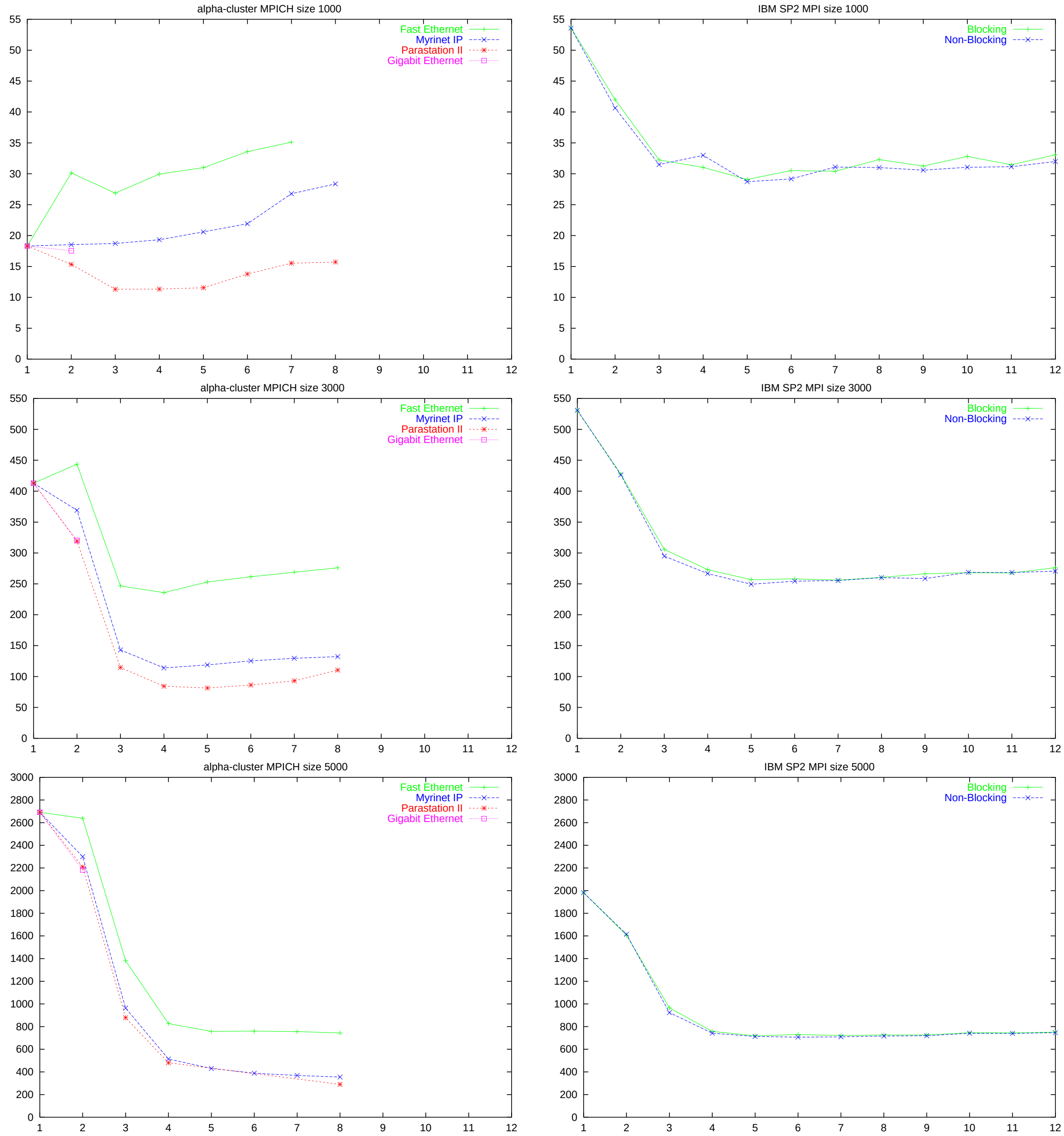

Figure 3.3: Parallel sorting on the DEC α cluster (left) and on the IBM SP2 (right). The results (wall clock runtimes / number of processors) for different problem sizes (n=1000, n=3000 and n=5000) are shown. For the DEC α cluster the different hard- and software combinations are compared. On the SP2 a comparison of blocking vs. non-blocking communication is made.

choosing different values of n can be scaled in an easy way. The runtimes we could achieve with this version on the DEC α cluster with different combinations of communication soft/hardware are shown in figure 3.3 for a small ($n = 1000$, 55 MB) a medium ($n = 3000$, $\sim$ 500 MB) and a large ($n = 5000$, $\sim$ 1400 MB) problem (the sizes in MB are for the 64-bit architecture of the alphas, they are roughly half as large for 32-bit systems, a more efficient compression to reduce disk access is work in progress). They show that only for sufficiently large problems a speedup can be achieved without sophisticated network soft/hardware.

MPI also supports nonblocking unbuffered send/receive functions. This should minimize the overhead and still enable latency hiding at the cost of having to implement the buffering ourselves. The parallel sorting was implemented in a way that can take advantage of these specialized message passing functions by using a set of cyclic buffers which eliminates any unnecessary overhead and also minimizing the time consumed for synchronizing. Choosing the number of cyclic buffers to be one results in using the send/receive function in blocking mode. At least on the IBM SP2 there are basically no differences in the results of the two versions (figure 3.3 right). We expect a larger speedup from the nonblocking message passing for the fully parallelized version of the program running on the ParaStation II hard- and software that does the communication independently of the CPU. It is also quite interesting to see that obviously the master process can generate enough terms to keep about 4 processes busy only, almost independently of the problem size. The difference is that the cost of additional overhead for more than 4 processes is larger for the small problems, which of course is no surprise. It was know from the profiling of the sequential code that a large part of the runtime is spent in the generation routines, so the speedup we could see for very large problems is about the optimum we could expect to reach at this intermediate stage.

The next step towards a working prototype is parallel generating: the distribution of input terms among all processes before generation of the output terms. This part of the parallelization is work in progress.

Chapter 4

Parallel Creation of Symmetric Patterns

It is well known that one can create symmetric patterns or objects by iterating special mappings e.g of a plane onto a plane [9]. Repeating this mapping many times (e.g 10 to 100 million iterations) leads to the formation of symmetric patterns called attractors. These attractors show interesting internal structures depending on various parameters of the mapping and the number of iterations. A program was written to study these effects and the large number of iterations suggested a parallelization of the program. The speedup results of the parallel version - using our DEC α cluster with Parastation II software - can be seen in Fig. 4.1. We want to extend this activity to other programs that can be used to study symmetries and mappings.

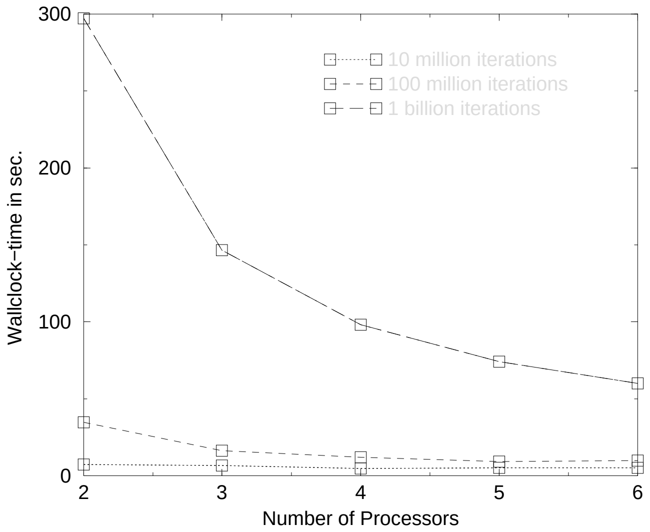

Figure 4.1: Runtimes of the parallel creation of symmetric patterns

Chapter 5

Summary & Outlook

As mentioned in the project description the aim of the first milestone has been achieved. The parallel cluster has been set up and the Parastation software is installed and running. Concerning FORM a parallel version has been implemented, which performs the sorting mechanism in parallel. In addition a smaller activity with an application of parallel image processing has been tested. The activities for the near future could be characterized with the following aims:

- Optimizing and stabilizing the performance of the cluster of 8 DEC α workstations, mainly by tuning and improving the Parastation II software.
- Implementing the parallel generation of terms in FORM
- Installing and improving a parallel Monte-Carlo program (VEGAS)

Finally one should mention that the 8 workstations of the DEC α cluster have often been used as single devices by other members of the "Forschergruppe Quantenfeldtheorie, Computeralgebra und Monte-Carlo". This demonstrates one of our important goals, that simultaneous parallel and single user mode is possible on our configuration. The following members of the University have been involved in this project:

- Prof. Dr. J. Kühn (project leader)
- Prof. Dr. H.M. Staudenmaier (project leader)
- Dr. D. Fliegner (scientific research assistant)
- A. Rétey (PhD cand.)
- H. Kaiser (diploma stud.)

Bibliography

[1] J.A.M. Vermaseren, *Symbolic Manipulation with FORM*, published by CAN (Computer Algebra Nederland), Kruislaan 413, 1098 SJ Amsterdam, 1991, ISBN 90-74116-01-9

[2] D.Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997.

[3] MPI(CH) homepage: http://www.mcs.anl.gov/mpi/

[4] PVM homepage: http://www.epm.ornl.gov/pvm/

[5] Myricom homepage: http://www.myri.com.

[6] ParaStation homepage: http://wwwipd.ira.uka.de/ParaStation/PSM/

[7] Netperf homepage: http://www.netperf.org/nerperf/NetperfPage.html

[8] Pallas MPI Benchmarks: http://www.pallas.de/PMB2/

[9] M.Field + M.Golubitsky, *Chaotische Symmetrien*, Birkhaeuser Verlag, 1993, ISBN 3-7643-2844-4