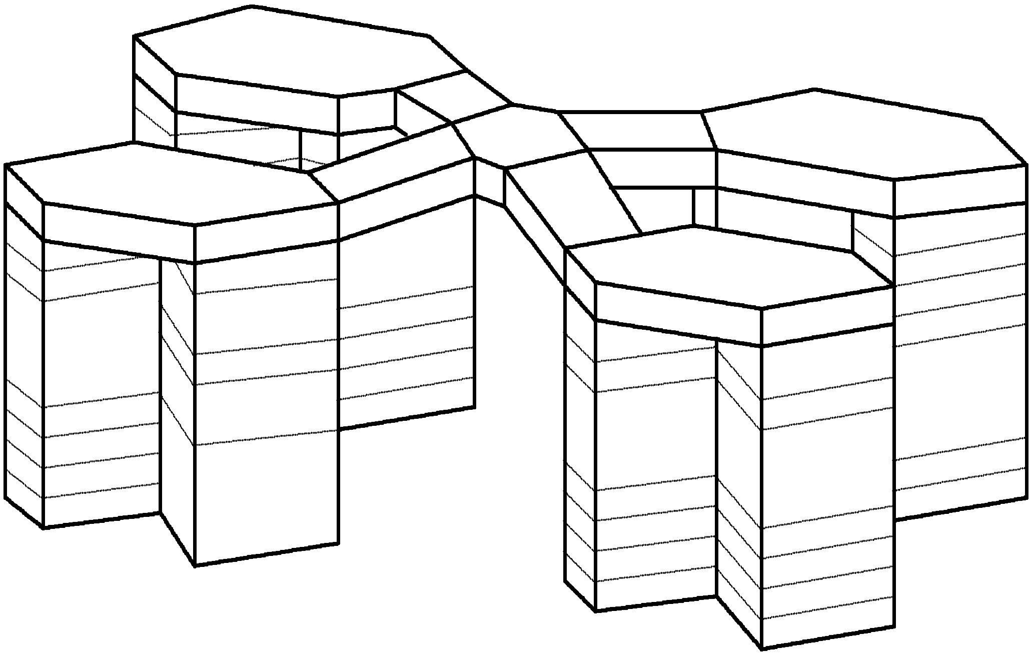
One Year KSR1 at the University of Mannheim



Results & Experiences

Robert Schumacher (Ed.)

RUM 35/93

December 1993

PREFACE

In December 1992 the parallel computer KSR1-32 from Kendall Square Research in Waltham/Mass. was installed at the University Computing Center. The installation was preceded by a very short time of approval after the decision for the KSR1 was made by the University of Mannheim.

Here, true thanks go to the Ministerium für Wissenschaft und Forschung (MWF) in Stuttgart, the Kommission für Rechenanlagen (KfR) of the Deutsche Forschungsgemeinschaft (DFG) in Bad Godesberg as well as to the Wissenschaftsrat in Bonn. I also want to thank the manufacturer KSR as well as Siemens Nixdorf, with whom we have signed the final contract because of SNI's cooperation with KSR. Both companies rapidly drove forward the installation of this supercomputer without much bureaucracy. Never before had we experienced a less than 4 hour installation the way it happened on December 9, 1992 here at the Computing Center.

By the end of 1992 parallel computers had already outgrown the stadium of being solely objects for research in computer science and research institutes. Because of this MWF within the framework of BelWü – the research–net of Baden–Württemberg – had started the initiative Parablü – Parallel Computers in BelWü – with the goal to install parallel computers in several university computing centers in order to gather knowledge of these innovative computer typs promising a great future. Because of this, one of the procuring demands made by MWF was to install different architectures in Baden-Württemberg in order to gather knowledge on a broad level. Presently the following parallel computers are installed at the university computing centers in Baden-Württemberg:

Intel Paragon XP/S5 in Stuttgart, MasPar MP 1216 in Karlsruhe, KSR1–32 in Mannheim and nCube2S in Ulm.

Being one year in use at the University Computing Center in Mannheim we present the following report of the KSR1. This covers the attachement of the computer into the computing center and into the university backbone net as well as the experience during operation. The center of attention will be the experience of users with KSR1. These users come from the University of Mannheim and other universities in Baden–Württemberg – via BelWü –, from some universities from Germany via the science net WIN – as well as from neighboring countries. Little use of the KSR1 was made by the industry. Quite a few projects just started recently so we have no proof of experience yet but they have already found mentioning under ongoing projects.

As a result at this point it can be stated that of course - and we did not expect it differently - there are still (software) problems and also problems with the reliability of parallel computers; KSR1 is no exception. But before the reader hastily starts criticizing it should be remembered that 18 years ago Cray1, the very first commercially available vector supercomputer came practically without software into the market and it took almost 10 years until the vector computer found acceptance in the industry as well because of mature vectorized Fortran compiler and because of professional application software.

Our one–year experience with the KSR1 and the experience of the users reporting to us give justified hope that the parallel computing technology will be established much faster than formerly the vector technology. The user experience was namely altogether rather positive which might be to a great part due to the ALLCACHE architecture of the KSR1.

We do hope that the experiences from users of economical science, computer science, mathematics, physics, chemistry, engineering science and the industry presented here will contribute to take anxiety out of the use of parallel computers and to encourage the use of this so very important technology.

Mannheim, December 1993 — Hans–Werner Meuer

CONTENTS

The KSR1 in a Computer Center Environment

Robert Schumacher
Computing Center, University of Mannheim,
Postfach 10 34 62, D–68131 Mannheim, Germany
schumacher@rz.uni-mannheim.de

Abstract

As the first computing centers have parallel computers installed and collect experiences, it is of wide interest how the first hardware virtual shared memory computer – the KSR1 – is integrated in a computer center environment. In this article the experiences of one year KSR1 are reported. A detailed analysis of stability and availability is given. It turns out that with an availability of more than 99% and a very user–friendly environment the KSR1 is a highly usable parallel computer for a wide range of applications.

1 Introduction

Parallel computers are leaving their niches from some parallel computing labs and become more and more popular also with computing centers. In nearly all cases of buying a new powerful computer, parallel computers are seriously considered to be an alternative to the present available multiple processor vector (MP) computers. Surprisingly enough this holds not only for computing centers of universities but also for the industry, where the costcutting on one hand and the increase of demand for computer power on the other hand let the managers search other solutions than the traditional 'just buy the next biggest machine'.
However, if you are in the situation that you have some money and you need a new powerful computer now, you are in some trouble. If you buy one of the MP–computers, than probably you are far behind in one years time, in means of peak performance and maybe more important in the price performance. If you buy a true massively parallel (MPP) computer than you face the problems like availability of software, stability of the machine etc. already now, however these problems are not unsolvable and some early investment in a promising future might save a lot of money in a couple of years.
In this situation everybody closely looks at the experiences made with already installed systems. This is the case in the computing center of the university of Mannheim, where a KSR1 with 32 processors was installed in December '92. So it is time to report to the community about our experiences. For a computing center performance is not everything, but also the manpower you need to run the machine as well MTBF figures, availability, stability, maintenance tasks, the time you spent for user–support etc.
This article will focus on the above mentioned subjects, where the following articles will provide the reader with the aspects related to performance, scalability and efficiency of the KSR1 for a wide range of problems, as numerically intensively tasks, computeralgebra, the implementation of communication protocols or the research of macroeconomic systems.
In the following sections we give first an overview about the hardware and software of the system installed in Mannheim. In section 3 we publicize our experiences with stability and availability of the system.

2 Details about the Mannheim installation

2.1 Hardware

The KSR1 system at our computing center consists of 32 processors each with 32 MB local cache to a total of 1 GB main memory or in KSR terms 1 GB of ALLCACHE. The 32 processors are connected by a slotted, pipelined, unidirectional ring, which forms together with the memory the ALLCACHE ENGINE:0 (ACE:0). Up to now, we have a rather small disk space of 10 GB, which is divided into a raidgroup (raid–level 3) with 4 GB useable space for the user data, 2 GB for system data, 1 GB swap space, 1 GB for test purposes and 1 GB as a second root–device. The connection to the outside world is via ethernet within the computing center and FDDI, when coming from the internet. The console – a NeXTstation – is connected through a serial line. The only task you need to do from a console is to boot the machine manually.
The processor is a superscalar RISC proprietary design running at 20 MHz, and executing two instructions each cycle. This yields a peak performance of 40 MFLOPS for a single processor. The dataword is 64 bit, what probably causes the most problems by porting applications to the KSR1, the addressword is 40 bit, which provides a terabyte of address space to the programmer. Also within the board is the local cache and a subcache, which holds for data and for instructions 0.25 MB each. The CPU is made of four chips, namely the cell execution unit (CEU) with 32 40 bit address registers, the floating point unit (FPU) with 64 64 bit registers, the integer processing unit (IPU) with 32 64 bit registers and the external I/O unit (XIU). The XIU provides a 30 MB/s channel to the I/O–boards like the multiple disk adapter (MCD) and others.
But the heart of the machine is the memory system. It is a virtual memory, which is mapped to distributed physical memory done by hardware. However, the physical distribution of memory causes in all cases non–uniform memory access costs, this is of course also true for the KSR1. Most obvious is this when looking at the single processor performance vs problem size. In some of the following articles just this relation is depicted in the form of a stepfunction. You will get the first step, when the problem size exceeds the subcache size, the next, when it exceeds the local cache size and so on. When you start to parallelize your problem the memory access costs effects get complicated, so it is, e.g., possible to get superlinear speedup, in the case the aggregated subcache size is bigger than the problem size. Here the KSR1 behaves principally like a distributed memory machine, the big difference to a message–passing computer is that the synchronization of memory access and the guarantee of data coherency is done by the hardware and has not to be done by the programmer.
In a further step one can connect 2 – 34 ACE:0 to an ACE:1, comprising 1088 processors. Within the ACE:0 the bandwith is 1 GB/s, within the upper level you may have up to 4 GB/s bandwith. Only the members of the ACE:0 are processors, the members of a higher level ring are lower level rings. In [1] is an overview of the technical details available.
One can partition the KSR1 into processor sets. Each processor set consist of a number of processors. On every machine is always at least one processor set: the default set. Also there must be at least one processor in the default set, because when you log in, you are on the default set. Above that, there are no restrictions of building processor sets. The mightyness of this tool is, that you can change access authorizations as well as size of the sets interactively. We do this very often to give a user or a group a set exclusively for measurement purposes, so other users can still work, while performance measurements are done. In figure 1 our current processor set configuration is shown.
The default set is mainly for interactive tasks, the small set for testing purposes and the big set with 20 processors for production. Also all processors, which deal with I/O are in the default set, so if you run your job within a set other than the default set, your application

1 ISS	16 MCE
19 MCF	22
29	30
31 MCD	32

default

24
25
26
27

small

2	3	4	5	6
7	8	9	10	11
12	13	14	15	17
18	20	21	23	28

big

Figure 1: Current processor set configuration of the KSR1 in Mannheim. Indicated is the proc. number and a connection to an I/O board.

won't interfere with other applications doing I/O. The split of the processors into sets does not split the memory, still each processor, independent to which set it belongs, has access to the whole memory.

2.2 Software

The KSR1 runs an extension of the Unix OSF/1 operating system and provides compatibility with Systems V and BSD4.3 operating systems. Because of the virtual memory, each processor knows about the full OS. With the OS comes a bundle of the widespread GNU–software and X–Windows with OSF/Motif, especially this makes the beginner feel familiar with the machine. We have compilers for C, C++, Fortran 77 with extensions and Modula–2. KSR also announced a Fortran 90 compiler. The C++ and the Modula–2 compiler are translators to C. For Fortran 77 exists a preprocessor for automatic parallelization, called KAP. For exploiting the parallel features of this computer from the other languages one has to make explicit calls to the pthread–library. POSIX threads or pthreads are the unit of work for the KSR, they are also the unit for the scheduler. For the processors and the processor sets exist queues, which holds the pthreads waiting for execution on a processor or a processor set. The scheduler does the load balancing by moving the pthreads between the queues.
Besides NAG–Library, LAPACK and the BLAS routines there are tools for debugging, profiling and performance analysis all with a X–Windows interface. To provide portability to message–passing computers and workstation clusters we installed the TCGMSG, PVM and P4 message–passing libraries.
Other software being of interest, but is not available at our installation include the IMSL–library, the PARMACS message–passing library and a bunch of chemical and engineering software packages. In the commercial sector exist ORACLE 7, and the transaction monitor TUXEDO as prominent examples.

3 Stability and availability

As already mentioned in the introduction stability and availability are key figures as well as performance numbers. Parallel computers are not very long at the market, however the vendors of these machines are claiming already a very high level of availability for their systems. So it is not unfair to compare the reality with the promised features.

In figure 2 the number of crashes and shutdowns we had each month are shown. We had a record in March with 2 crashes a day due to hardware problems. Surely the number of crashes could have been diminished with better diagnostic tools, so the machine passed all diagnostics but crashed five minutes later. So it was a try and error procedure to locate the fault unit. This was the only hardware failure we had in the reported period. In the meantime the fault

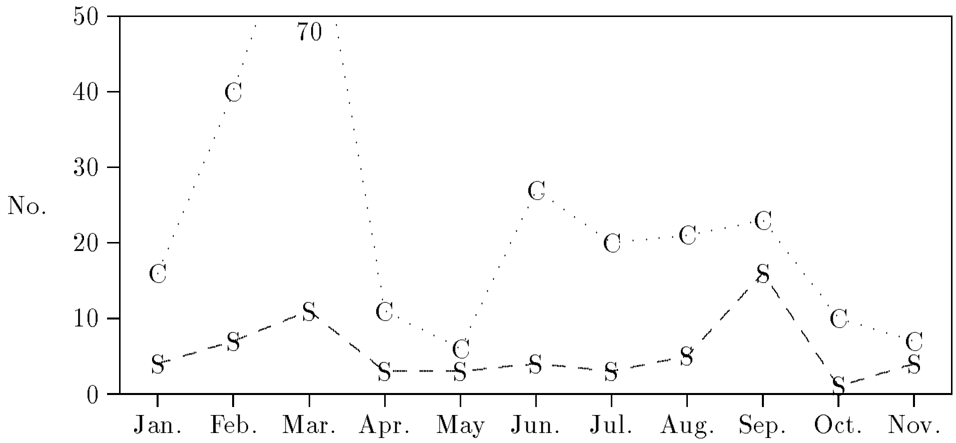


Figure 2: Crashes and Shutdowns Jan. 93 – Nov. 93

detection and diagnostic software improved significantly. In most cases of a crash a dump is written to disk, which allows a post–mortem analysis of the fault.
The situation relaxed in April and May, where we were down to 6 crashes a month, what lead to a MBTF $>$ 5 days. The situation got worse again in June, when some projects started. So we had ca. 20 crashes a month until September. Late September with the OS release 1.1.4 again it relaxed and in November we recorded 7 crashes.
The crashes are in nearly all cases due to user written programs, which put to much load on the machine, or make use of critical low–level instructions. In the last release of the operating system KSR has improved the handling of these situations, as can be seen also from the pictures. The software coming with the operating system, as compilers or the above mentioned tools, to my knowledge never crashed the system, however there were and still are some bugs in the software, what causes runtime errors.
To have a better feeling for the availability of the machine, in figure 3 the number of days the system crashed during a month is shown. For the user it is a big difference, when a system crashes on 10 days a month or has 10 crashes at one day of a month, which he probably wouldn't notice. The number of crashdays is continously falling since June, down to 4 days in November. So the user can expect that his application can run a week on our KSR1. The longest period the system ran without a crash were 15 days, when we did a maintenance shutdown.
The availability of the system is depicted in figure 4, it is also shown how long we run which OS level. Numbers $>$ 99% indicate that it does not take long to boot a KSR1. The most time consuming tasks when a crash happens, are to do the crash dump and to perform the `fsck`. For our system this takes $<$ 20 min. However it may happen that the system crashes and does not reboot automatically, than the reboot has to be done manually. This might cause longer downtimes, these are not included in the figure. Included is the downtime for maintenance, which is ca. 1 h a month.
The maintenance tasks for the KSR1 are typically the same you would have to perform as for a Unix–Workstation. Even to change a piece of hardware, it takes normally less than one hour to replace it, whether it is a disk, a processor board or an I/O adapter. The first impression

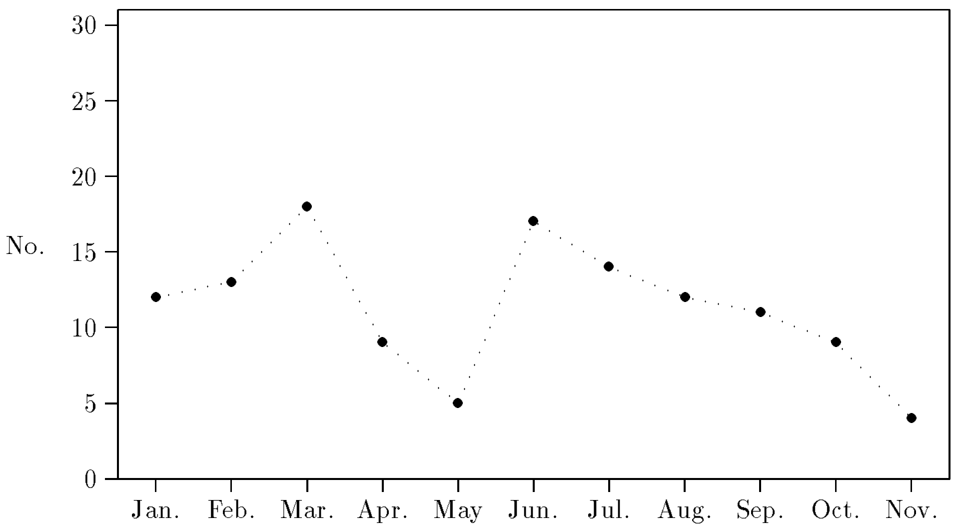


Figure 3: Crashdays Jan. 93 – Nov. 93

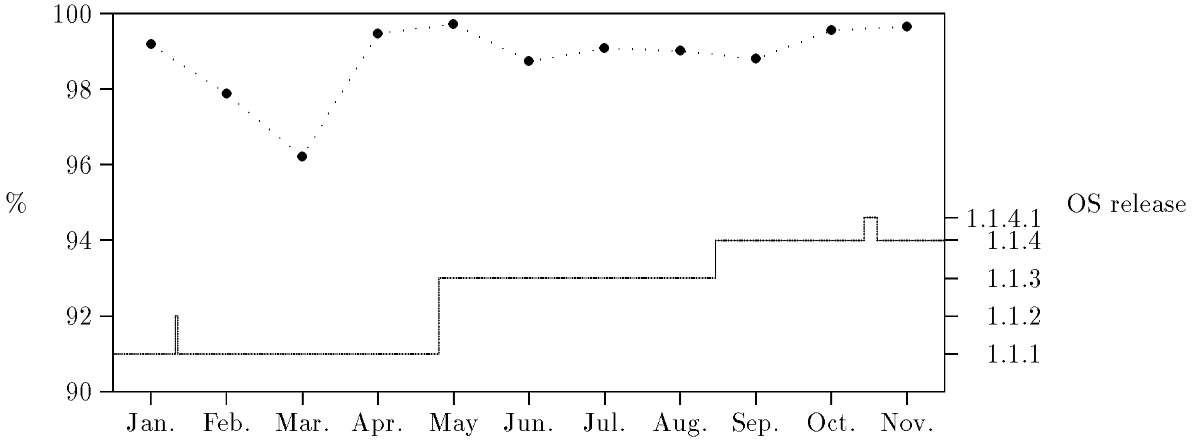


Figure 4: Availability Jan. 93 – Nov. 93

of the KSR1 when the system was installed exactly one year ago was that it took only 4 hours from unloading the truck until the first prompt from the operating system. In August the computing room was moved to another floor within the building, so the system was unbuilt moved to the new location and rebuilt again. The whole action took about 6 hours for 2 persons. When comparing MTBF and and availability with other computers, one should also have these numbers in mind.
To give also an impression on the manpower which is necessary to run the machine, the number of persons involved is 3. These 3 people do everything from system management, user–support, this mainly scientific support, project–support, especially within the university, programming training classes and lectures as well as some own research.

4 Conclusion

The KSR is a very user–friendly computer, provides the user with a full Unix environment and also good performance. Sure the number of crashes should come down even more, but compared to the mainframe – the computing center is also running a BS2000 computer – the KSR has a much higher functionality and performance. The progress, that is made towards higher stability is evident and should not cause any problems in the future. A critical point, mentioned also often in the following articles is the scheduling, multitasking often slows the machine down. This field still is a subject of research, what is true for other parallel computer also.
The acceptance of the KSR1 is very high, our machine is used for a wide range of problems, as this report shows, and still the number of projects is increasing.

References

[1] Kendall Square Research. *Technical Summary,* Waltham 1992.

Implementation of Estelle Specifications on the KSR1

Stefan Fischer
University of Mannheim, Praktische Informatik IV,
P.O. Box 10 34 62, D-68131 Mannheim, Germany
fischer@pi4.informatik.uni-mannheim.de

1 Introduction

Existing communication protocol suites such as the ISO/OSI [7] or the INTERNET protocols [2] were designed with relatively slow networks in mind. The end systems were fast enough to process complex protocols because the data transmission time was long compared to the time needed for protocol execution.

Given the fast transmission media based on fiber optics that are now available, current end systems are too slow for high performance communication. Communication software has become the major bottleneck in high speed networks [10, 1]. Efficient implementation of the protocol stack is of crucial importance for the networking future.

For specification purposes, formal description techniques are now widely used. They improve the correctness of specifications by avoiding ambiguities and by enabling formal verification. In addition, they allow semiautomatic code generation.

This technique has several advantages: The code can be maintained more easily since the system is specified in an abstract, problem–oriented language. It is also much easier to port an implementation to another system. But one of the major problems is the performance of implementations produced automatically from a formal specification.

In our project we use the formal description technique Estelle [8]. Estelle specifications mainly consist of a hierarchically ordered set of extended finite state machines, so–called modules, which are communicating by message exchange via directional channels with queues on each end.

Existing code generators were made to easily get rapid prototypes for simulation purposes. They produce higher language code (e.g. C or C++) from Estelle specifications. Executable specifications lead to a better understanding of protocol behaviour. Since performance aspects are not essential for such a simulation, existing Estelle tools are designed for validation rather than for the generation of efficient code for high performance implementations.

A considerable amount of the runtime of automatically generated implementations is spent in the runtime system. This gives rise to hopes that much more efficient implementations can be created by code generators if the runtime system can be improved, especially by the use of parallelism.

Thus, the goal of our project was to develop a code generator that accepts an Estelle specification as input and produces a parallel program as output.

To use a parallel protocol implementation efficiently, it is important to have a multiprocessor system. For our project, we selected a KSR1 [5] equipped with 32 processors and running the operating system OSF/1. This machine is available at the University of Mannheim since December 1992. The most important characteristics for our project were, in case of the machine, the virtual shared memory architecture, which reduced the complexity of the code generator, and, in case of the operating system, the availability of threads which leads to a better performance by reducing the context switching overhead.

2 Structure of Implementations

In this project we did not implement a fully new Estelle compiler. We used the PetDingo system [9] and exchanged the runtime system. A detailed description of the new system may be found in [4] and [3].
The structure of the generated code (C++) remains the same. In the old runtime system there existed a scheduler which called one Estelle module after the other while observing a certain Estelle semantics. In the new system, each module is run by one thread thus allowing the use of multiple processors. The above mentioned Estelle semantics is observed by the use of explicit module synchronization. There is no central scheduler anymore.
For the synchronization we use OSF/1 lock and condition variables. When a module wants to synchronize with another one, it changes the state of a shared memory variable and signals this change using a condition variable. The partner module will be waked up and read the new state of the shared memory variable. It will then execute its own cycle until it is ready. Parallelism is possible, as one module may synchronize with more than one other module at a time, if Estelle semantics allows it.

3 Performance Results

For measuring the speedup of parallel implementations of a protocol in comparison to a sequential one, we set up the following scenario: the basic building block is a connection between an initiator and a responder. They are each sitting above a quite generalized protocol stack of a dynanmic height that might be adjusted from 1 to 5. Each module in this stack takes a message from its input queue, does some processing and puts the message to his output queue. Both stacks are connected by a transport pipe that delivers messages from one stack to the other. A supervisor module may create 1 to 5 of these connections. Thus, we are able to measure both the influence of processor–per–layer and processor–per–connection parallelism (for a deeper description of these terms, see [6]).
For our measurements we used 20 of the 32 processors of the KSR1. The results may be seen in Fig.1. For this protocol, we obtain a remarkable speedup of about 10 to 11 with 5 connections and a protocol stack height of 5. It has to be noted that the full speedup (20 in this case because of 20 processors) will never be reachable in the implementation of Estelle specifications. This is due to Estelle semantics which explicitly forbids full parallelism between all modules. The most important barrier for a further speedup is the rule that modules on different levels in the same tree of the module hierarchy (for a more precise description of this concept see [8]) may never run in parallel. The synchronization rules of Estelle specifications are quite complex; ongoing research in our group deals with a further increase of the speedup.
It is obvious that the increase of the number of connections leads to an increase of performance. An interesting point is the structure of the single curves, especially when using 3, 4 or 5 connections. The curve rises steeply when using only a small stack height and flattens when the height is increased. The reason for this is that, at this point, the number of threads becomes greater than the number of available processors. The processor utilization becomes better, but also the the sum of synchronization times between threads increases.

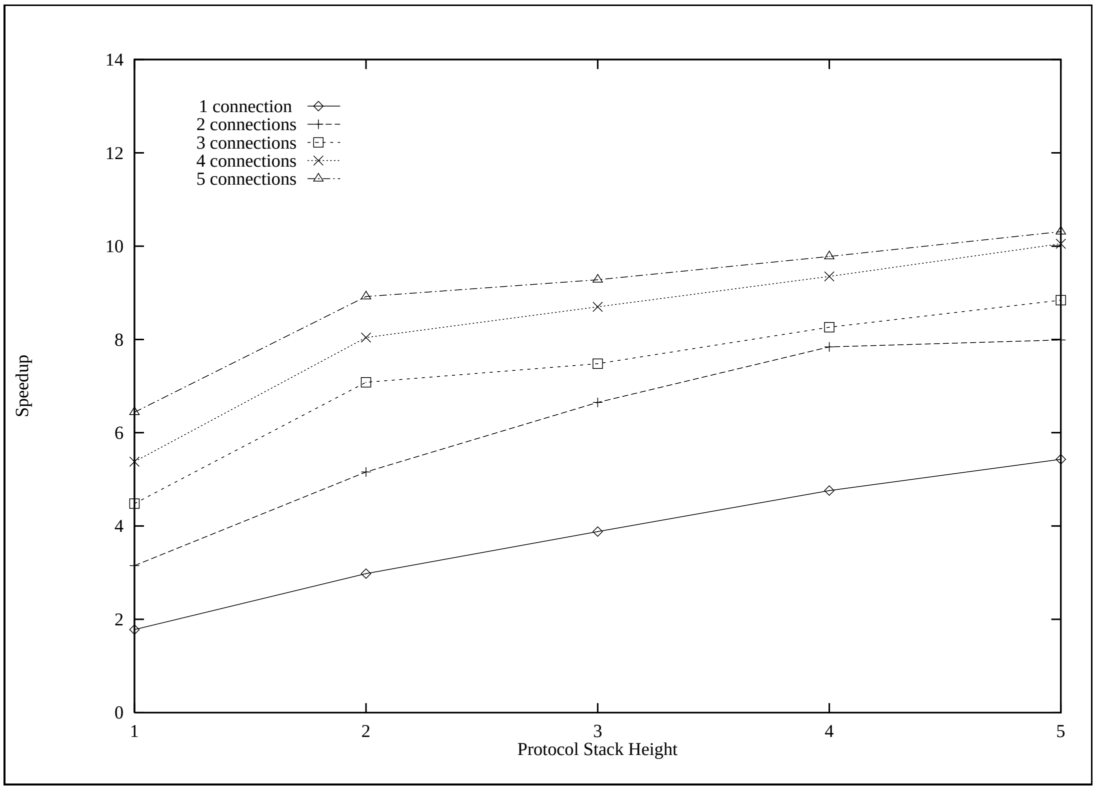


Figure 1: Performance Results for Multiple Connections

4 Future work on the KSR1

In the project described above we are currently working on an improved version of the Estelle compiler. In another project, we have ported an MPEG encoder to the KSR. MPEG is a standardized compression format for digital movie streams. We are investigating optimal parallelization units for the encoding of MPEG streams.

References

[1] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM '90 Symposium Communication Architectures & Protocols*, pages 200–208, Philadelphia, September 1990.

[2] D. Comer. *Internetworking with TCP/IP*. Prentice–Hall, Englewood Cliffs, 1988.

[3] S. Fischer. Generierung paralleler Systeme aus Estelle–Spezifikationen. Master's thesis, Lehrstuhl für Praktische Informatik IV, Universität Mannheim, 1992.

[4] Stefan Fischer and Bernd Hofmann. An Estelle Compiler for Multiprocessor Platforms. In Richard L. Tenney, Paul D. Amer, and Ümit Uyar, editors, *Formal Description Techniques*

VI, Boston, USA. Elsevier Science Publishers B.V. (North-Holland), October 1993. to appear.

[5] S. Frank, H. Burkhard III, and J. Rothnie. The KSR1: High performance and ease of programming, no longer an oxymoron. In H.-W. Meuer, editor, *Supercomputer '93: Anwendungen, Architekturen, Trends*, Informatik aktuell, pages 53–70. Springer Verlag, Heidelberg, 1993.

[6] B. Hofmann, W. Effelsberg, T. Held, and H. König. On the Parallel Implementation of OSI Protocols. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Tucson, Arizona, February 1992.

[7] Information processing systems — Open Systems Interconnection — Basic Reference Model. International Standard ISO 7498, 1984.

[8] Information processing systems — Open Systems Interconnection — Estelle: A formal description technique based on an extended state transition model. International Standard ISO 9074, 1989.

[9] Rachid Sijelmassi and Brett Strausser. The PET and DINGO tools for deriving distributed implementations from Estelle. *Computer Networks and ISDN Systems*, 25(7):841–851, 1993.

[10] L. Svobodova. Measured performance of transport service in LANs. *Computer Networks and ISDN Systems*, 18(1):31–45, 1989.

Optimization of large-scale order problems by the Evolution Strategy*

Hans–Georg Beyer
University of Dortmund
Department of Computer Science, Chair of Systems Analysis
D-44221 Dortmund, Germany
beyer@ls11.informatik.uni-dortmund.de

Abstract

An implementation of the Evolution Strategy (ES) on the KSR1 for solving large-scale order problems is presented. The ES exhibits a (theoretical) linear speed up if parallelized. Results are presented for the optimal LINAC design (accelerator physics) and for the well known traveling salesman problem (TSP).

1 The Evolution Strategy

Order problems are known to belonging to the class of *NP-complete* problems. A powerful and very general method approximating optimal solutions to these problems is given by DARWIN's paradigm of evolution '*the survival of the fittest*'. The Evolution Strategy (ES), originally designed for parameter optimization and tuning [1], is a straightforward implementation of this idea. Two problems have been solved on the KSR1 by ES:

- The optimal arrangement problem for linear accelerators (LINAC), i. e. finding an arrangement of N different (and numbered) accelerator structures that minimizes the Multibunch-Beam-Breakup BBU [2].
- The traveling salesman problem (TSP) of finding the minimal tour length F of N cities.

Assuming that the cities and structures, respectively, are numbered the arrangement can be expressed by a state vector $X := (x_1, x_2, \dots x_i, \dots x_N)$ containing the order of (e. g.) the cities. Thus, the combinatorial optimization task is to find the optimum state vector $\hat{X}$ that minimizes a certain objective, often called *fitness*, $F := F(X)$ (BBU-value, tour length, etc.) by permutation of the initial state(s) $X^{(0)}$ (e. g. $X^{(0)} := (1, 2, 3, \dots, N)$).
The basic algorithm of the so-called $(\mu,^+ \lambda)$-ES (without recombination) works as follows [3]:

1. `Initialization`
   Generate randomly μ parental states X_P^m, $(m = 1 \dots \mu)$. Determine the fitness $F_P^m := F(X_P^m)$.
2. `Reproduction`
   Produce λ $(\lambda > \mu)$ offspring X_O^l. An offspring X_O is produced by taking one of the parental states X_P at random (*asexual reproduction*).

*This work was done during a sabbatical at the computing center of the University Mannheim.

3. `Mutation`
   Perform small (and feasible) permutations on the offspring:
   $X^l_O := MUTATION(X^l_O)$, i. e.
   TSP: perform *Lin-2-opt* step(s): choose at random two cities (i. e. positions) of the tour and reverse the order in which the cities in between this pair are visited.
   LINAC: choose at random two positions i, k in X and exchange their values $x_i \leftrightarrow x_k$.

4. `Selection`

   (+)-selection
   Determine the offspring's fitness $F^l_O := F(X^l_O)$ $(l = 1 \ldots \lambda)$. Establish a ranking taking into account both the parental F^m_P and the offspring's fitness F^l_O. Choose the best μ individuals to be parents of the next generation.

   (,)-selection
   Perform selection by taking the offspring's fitness F^l_O into account only, i. e. the parents become extinct *per definitionem*. Choose the best μ individuals.

5. `IF` stop criterion `NOT` true `GO TO` 2.

2 Implementation on the KSR1

Due to the basic principles explained above there is a 'top level' parallelism inherent in the ES which is superior for MIMD machines. Reproduction, mutation, and selection (except for the sorting of the fitness values) can be done in parallel. The *master-worker-paradigm* has been applied.
The master is responsible for the set up of the workers, the user interface, fitness sorting (and selection), and bookkeeping operations. If there are π processors available, the master reserves one for itself and generates $(\pi - 1)$ *pthreads* running the worker code on the $(\pi - 1)$ remaining processors. Each worker performs after the start-up (setup of the parallel random number generator[1]) the selection, mutation (and perhaps recombination), and the fitness evaluation. Since in $(\mu,^+ \lambda)$ strategies λ offspring are to be produced the master-worker-approach exhibits a (theoretical) linear speed up as long as $\pi < \lambda$, or to be more precise, if $k(\pi - 1) = \lambda$, $k \in \mathcal{N}$ holds. This theoretical linear speed up, however, will be diminished if communication comes into consideration. It is therefore of vital interest to keep (almost) all data local to the processor. There are only four globally shared arrays:

1. the parental and the offspring's state vectors X,
2. the parental and the offspring's strategy parameters (needed for the self-adaptation of the ES),
3. a vector containing the fitness values, and
4. a status vector indicating the master/worker that worker/master is ready for a new task.

Write operations on these globally shared arrays have been minimized, i. e. only viable offspring are allowed to write.

[1] In order to speed up the mutations a fully parallelized random number generator has been developed which is much faster than the KSR `random()` and even the `prng()` generator.

The program is written in FORTRAN. The only non-standard routines used are: `pthread_procsetinfo`, `pthread_self`, `pthread_create`, and `pthread_move`. There is no need for high level parallel constructs (parallel sections, parallel regions), mutexes, or barriers to implement the master-worker-concept since the Search Engine of the KSR ALLCACHE architecture ensures globally sequential data consistency.

3 Results and Outlook

3.1 LINAC optimization

Originally the BBU minimization was performed on a SUN-workstation cluster comprising 4 SPARC-2-stations with communication via NFS [4]. Since the fitness evaluation (i. e. computation of the BBU-value by a so-called tracking code) is very time consuming there is no communication bottleneck. The same holds for a second version implemented on a 32-T800 Parsytec system using the message passing paradigm [5] exhibiting a linear speed up. Thus, it is no surprise that the linear speed up is observed for the KSR too.
BBU optimization is not a real challenge for the KSR1, however, it can be used for floating point benchmarking. Fitness calculation on SPARC and on KSR takes almost the same time: 43 seconds (using `-O2` optimization). This is by a factor of six faster than the implementation on the T800 transputer. The average floating point performance on the KSR1 is estimated to $2MFLOPS$. This is a rather disappointing result compared to the peak performance of $40MFLOPS$ [7]. It seems very difficult to reach the peak performance by programs/algorithms which do not exhibit the ordered structure of a matrix-matrix multiplication.

3.2 TSP optimization

As pointed out the real challenge for the KSR1 is given by a master-worker-application with a high degree of communication compared to the fitness calculation. This is the case for the TSP where the fitness is simply the tour length F.
The tests were performed for a $N = 1024$ cities TSP with random city positions. The ES used was a $(30 + 300)$-ES with self-adaptation of strategy parameters and a city-by-city exchange mutation operator (not Lin-2-opt). Benchmarkings were done on a dedicated KSR1-32 having exclusive access to the processors. With the `allocate_cells` command the number of processors 'seen' by the ES program was preselected to 6, 11, 16, 21, 26, and 31, respectively. This corresponds to a number of 60, 30, 20, 15, 12, and 10 offspring per worker and generation (for one generation there are $\lambda = 300$ offspring to be generated). The fitness calculation per offspring takes $t_F = 6.2\,msec$, reproduction and mutation $t_m = 2.6\,msec$. The total time t_t used for one offspring is displayed in Table 1.

$p = \pi - 1$	3	5	10	15	20	25	30
t_t $[msec]$	14.0	14.3	15.3	16.4	17.7	19.2	21.0
t_o $[msec]$	5.2	5.5	6.5	7.6	8.9	10.4	12.2
t_c $[msec]$	0.4	0.7	1.7	2.8	4.1	5.6	7.4

Table 1: The total time t_t used for one offspring depends on the number of workers p. Raising the number of processors π results in an increase in overhead time t_o caused by increasing communication time t_c.

The overhead time t_o includes the constant ranking t_r, selection t_s, and bookkeeping time t_b needed by the master. The sum of t_r, t_s, t_b is estimated to $t_r + t_s + t_b \approx 4.8\,msec$. Thus, the

time t_c used for communication can be estimated: $t_c = t_o - (t_r + t_s + t_b)$.
It is interesting to see the cumulative effect over a constant running time, e. g. $t = 200\,sec$. Fig. 1 depicts the number of generations (each generation contains $\lambda = 300$ offspring) produced by the ES program versus the number of workers p (processors $\pi - 1$).

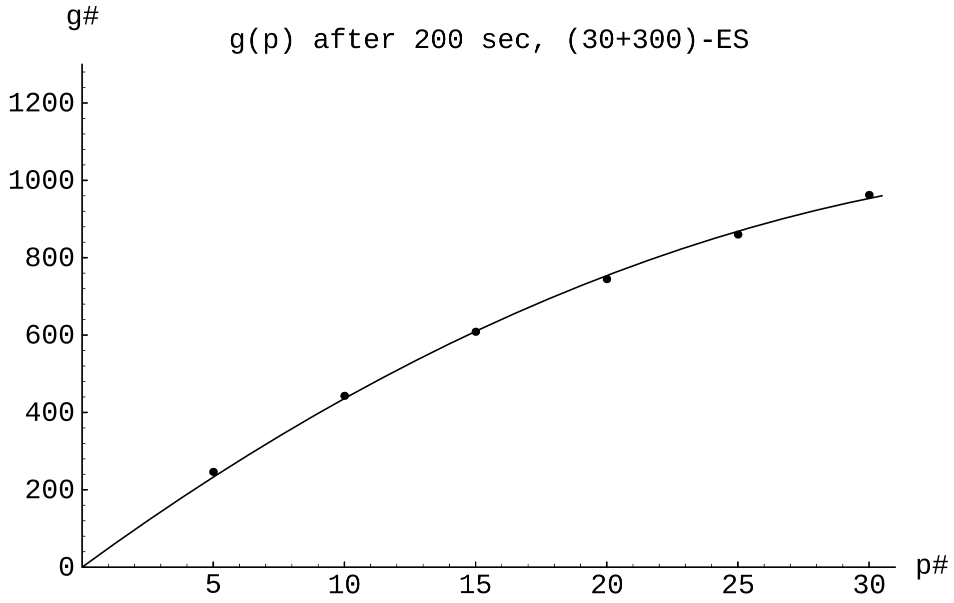


Figure 1: Number of generations g produced by the ES depending on the number of processors $(p + 1)$ used within a time intervall of 200 seconds.

One observes a sublinear speed up which can be approximated by a quadratic function: $g(p) \approx 49.5\,p - 0.59\,p^2$. The extrapolated maximum is at $\hat{p} = 42$, $\hat{g}(\hat{p}) = 1038$, i. e. the maximal number of processors useful for this application ($N = 1024$, $(30 + 300)$-ES) should be less than 43. It would be interesting to verify this prediction on a larger, say KSR1-128, system.
Fig. 2 depicts the optimization progress for different numbers of processors. The logarithm of the (best) tour length is a sublinear function of the number of generations produced ($\lg F \approx a - b \lg g$). This is a general property of algorithms solving for NP-complete problems (conjectured by the author [3])
The influence of the number of processors on the time needed to reach a certain optimization level can easily be seen. The advantage of the parallel ES approach is obviously demonstrated.

3.3 Outlook

The results obtained are promising and it is planed to extend the investigations to more powerful mutation operators for the TSP. The communication is the bottleneck of the TSP-ES. Therefore, it is a logical consequence to put more 'intelligence' in the mutation and fitness parts of the worker raising the ratio of computation/communication.
Generally it can be stated that combinatorial optimization problems having more complex fitness functions than the TSP should perform much better on the KSR1 (NB: one extreme is

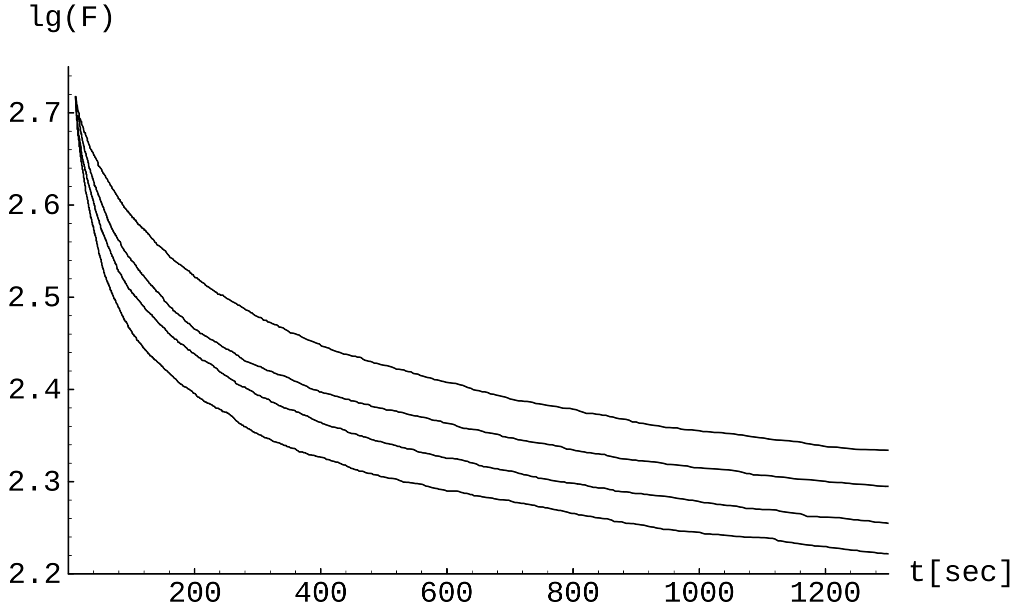


Figure 2: The evolution progress versus the total running time of the ES displayed for $p = 5, 10, 15$, and 30 workers realizing a $(30 + 300)$-ES. (From top to bottom: $p = 5, 10, 15, 30$.)

the LINAC BBU minimization). E. g., it would be interesting to investigate an ES for solving quadratic assignment problems (QAP).

Acknowledgements

The author gratefully acknowledges the support given by H.-W. Meuer and R. Schumacher, Computing Center University of Mannheim.

References

[1] Rechenberg, I.: *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog, Stuttgart, 1973.

[2] Beyer, H.-G. et al.: *Minimization of Multibunch-BBU in a LINAC by Evolutionary Strategies.* In Rossbach, J. (ed.): *15th International Conference on High Energy Accelerators HEACC'92.* Int. J. Mod. Phys. A (Proc. Suppl.) **2B**(1993), pp. 848–850.

[3] Beyer, H.-G.: *Some Aspects of the 'Evolution Strategy' for Solving TSP-Like Optimization Problems.* In Männer, R.; Manderick, B. (eds.): *Parallel Problem Solving from Nature, 2.* Elsevier, Amsterdam, 1992, pp. 361–370.

[4] Beyer, H.-G.: Technical Report *Benutzeranleitung für das EVO-LAN-Programm zur Minimierung der BBU durch evolutionsstrategische Optimierung der Anordnungsreihenfolge der Beschleunigungsstrukturen.* Technische Hochschule Darmstadt, FB 18, FG TEMF, Schloßgartenstr. 8, Darmstadt, 1992.

[5] Beyer, H.-G.: Technical Report *Anwendung des PARSYTEC-Transputersystems zur Designoptimierung bei einem 0.5TeV-Linear Collider: Evolutionsstrategie zur Lösung eines TSP-ähnlichen Reihenfolgeproblems.* Technische Hochschule Darmstadt, FB 18, FG TEMF, Schloßgartenstr. 8, Darmstadt, 1992.

[6] Meuer, H.-W.: *Supercomputer '93, Anwendungen, Architekturen, Trends.* Springer-Verlag Berlin, Heidelberg, New York, 1993.

[7] Frank, S.: The KSR1: High Performance and Ease of Programming, No Longer an Oxymoron. in [6], pp. 53–70.

Analysis of a Two Dimensional Dynamic System on the KSR1
An Economic Application

Volker Böhm and Markus Lohmann
Lehrstuhl für VWL, Wirtschaftstheorie
Universität Mannheim, D–68131 Mannheim, Germany
vboehm@econ.uni-mannheim.de
malo@pool.uni-mannheim.de

The analysis of complex nonlinear dynamic systems imposes high performance requirements on computing devices which are used to simulate the associated models. To obtain specific results it is often necessary to perform large numbers or sequences of iterations which, depending on the complexity of the system, may involve 10^6 or more calculations for each individual numerical experiment.

The research project **'Dynamic Macroeconomics: Business Cycle Theory and Experimental Economics'** studies as one of its principal systems a two dimensonal nonlinear macroeconomic model with an overlapping generations structure.

Some of the central problems to be investigated using numerical techniques originate from the theory of bifurcations. Others involve the calculation of high order fixed points and their basins of attraction. In order to generate numerical results, the model was programmed and set in C and initial calculations were carried out on a PC 486DX50 EISA. Because of the specific complexity of the model (– each iteration/period requires multiple steps of calculations rather than the evaluation of a simple dynamic equation, as in most dynamic systems in physics –), numerical procedures were typically more time consuming and required involved numerical techniques.

The simulations carried out on the PC typically showed a slow input output response, so that alternative procedures had to be investigated. Since many questions dealt primarily with tests over different parameter values which could be answered by independent loops, it was natural to use parallel computing techniques and import the computations to the KSR1–32.

For the computation of the bifurcation diagrams the numerical interval under consideration is divided into 20 subintervals. For each of these a separate independent thread is generated which evaluates its bifurcation values independently from all others. The results are filed in a shared data matrix which produces a print file after completion of the simulation in all threads. A screen output is generated during the simulation in order to control the output. Fig. 1 shows a srceen dump while producing a bifurcation diagram. The 20 individual elements grow together into a single graph. The number of iterations for these on the KSR1–32 is about 650,000 compared to about only 100,000 iterations for the equivalent diagram requiring the same amount of time on the PC.

The performance increase is even more pronounced for the calculation of basins of attraction. The procedure used varies two parameters simultaneously which increase the number of iterations considerably. Numbers up to 40,000,000 are no exception. Fig. 2 shows a specific basin as it develops.

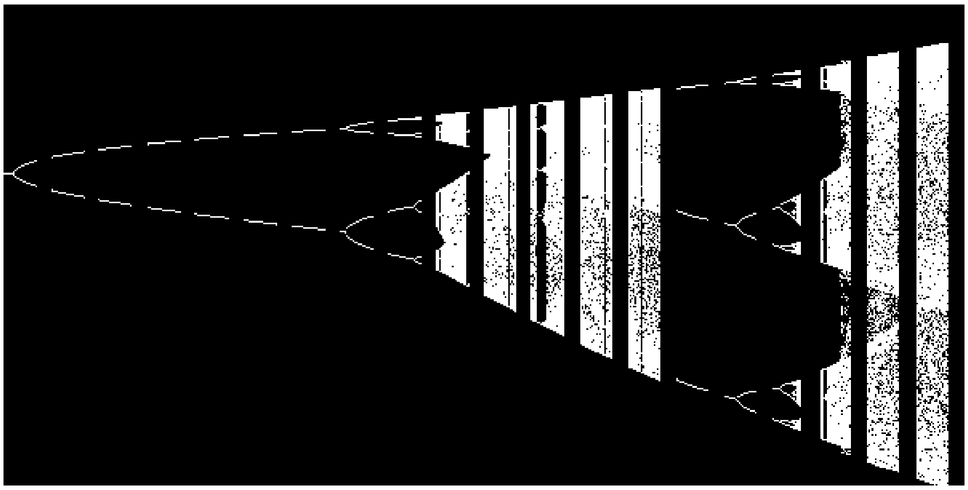


Figure 1: Screen dump while producing a bifurcation diagram

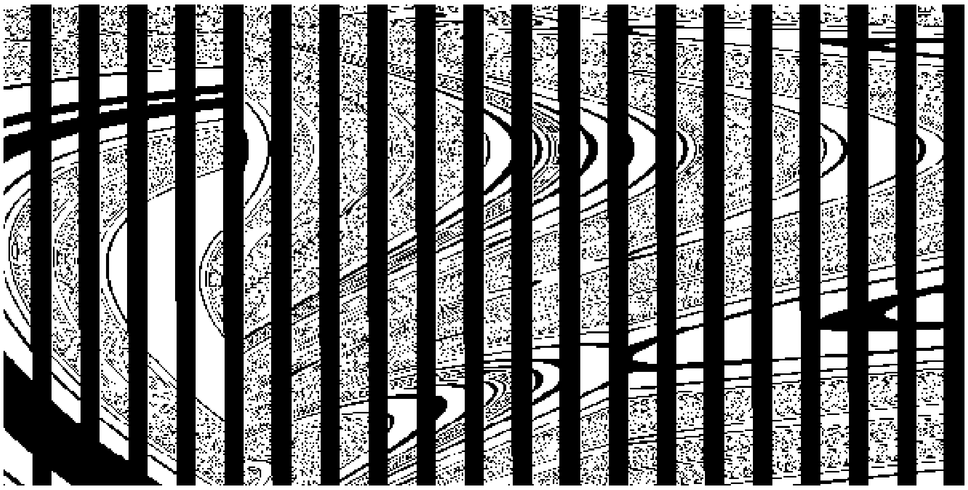


Figure 2: Screen dump while producing a basin of attraction

As before the range of one parameter is divided into 20 subintervals for which the calculations are carried out independently. For the second parameter a parallel procedure cannot be used. The output procedure here is analogous to the one chosen for the bifurcation diagrams. The basin of attraction in fig. 2 required 1,669,500,000 iterations (2625 values for the X–coordinate, 2100 for the Y–coordinate and 300 iterations for each pair). The KSR1–32 needed 4 hours and 37 minutes on a set of 20 CPU's which was available exclusively. Other means of calculations most likely would have taken days.
As a summary, it is obvious that importing the simulation model onto the KSR1–32 greatly improves the performance and makes applications possible which are out of reasonable reach for PC's and workstations. Further applications are intended in the future.

Programdevelopment on the KSR1

Erich Strohmaier and Richard Weiland
Computing Center University of Mannheim
Postfach 10 34 62, D–68131 Mannheim, Germany

Abstract

In this article we present our experiences with the development of a Fortran program on the KSR1. We focus on certain basic aspects like timing of programs and efficiencies of the compiler and of different methods of parallelisation. As a working–example we have chosen the matrix–multiplication.

1 Timing on the KSR

Besides the wellknown standard–timers of the UNIX–operating system the KSR1 has two hardware–based timers per cell. These timers are 64bit counters and reside in registers. They are incremented every 8 clock ticks and have therefore an accuracy of 400ns due to the clockrate of 20MHz.
The **XIU_register%x_user_timer** is a thread–oriented timer. Each thread has an own copy of the counter and this counter will be incremented only when the thread is active. If a thread is moved from a cell C1 to a cell C2, its countervalue is also moved.
The **XIU_register%xall_timer** is a cell–oriented timer which is initialised on all cells at boottime and thereafter incremented steadily. However this counters are not exactly synchronized and no sychronisation or exchange takes place, when threads are moving between cells.
In Fortran or C–programs one can use these hardware timers by calling the PMON–library. A call to these library routines gives back not only the user_seconds (based on XIU_register%x_user_timer) and the all_seconds (based on XIU_register%xall_timer) but much more cell and thread oriented informations like e.g. the number of subcachemisses or pagemisses. For linking to this library one has to use the –lpmon option by calling the loader. An example for its usage in a program is given below:

```
INCLUDE '/usr/include/pmon.fh'

INTEGER ibuff(ipmsize),istatus

CALL pmon_delta ( ibuff, ipmsize, istatus )

... computations ...

CALL pmon_delta ( ibuff, ipmsize, istatus )
```

After the second call the function pmon_delta gives back the differences of the internal countervalues with respect to the first call. These values are stored in the integer–array ibuff. A short list of these counters is given below. For a detailed description see the manual pages for pmon.

1	user_clock	number of user clock cycles
2	wall_clock	number of elapsed clock cycles
3	instruction_count	estimated instruction count
4	ceu_stalls_time	total time of CEU stall
5	xiu_ins_inst_time	total time of inserted instructions initiated by the XIU
6	cache_ins_inst_time	total time of inserted instructions initiated byn CCU
7	pg_hit	number of cache page hits
8	sp_hit	number of cache subpage hits
9	pg_miss	number of cache page misses
10	sp_miss	number of cache subpage misses
11	sp_miss_time	total time of subpage misses
12	dsc_miss	number of data subcache misses
13	ae1	number of packets that traveled through allcache engine:1
14	prefetches	total number of prefetches
15	prefetch_miss	prefetches of data which where not in cache
16	prefetch_prt_hit	prefetches which hit a full prt
17	thread_migration	number of times the thread migrated between cells
18	page_faults	number of os page faults

For timing of serial programs the above mentioned two hardware timers are very usable. For the timing of parallel programs however, they are of limited value. If you use the thread–oriented timer user_second for timing of your master–thread, the time of the child–threads is not taken into account. So if the master–thread is moved out of the machine during the execution of a child–thread, his counter is not incremented. As an effect of this you don't get back the time of the longest running thread but in the worst case a time, which has nothing to do with the execution time of your program. The cell–oriented timer wall_clock can only be trusted if you have a dedicated machine (no other programs running) and if your master–thread is bound to a certain cell.
During developing or porting programs to the KSR1 you can use these timers mostly for serial runs. The best method for timing parallel programs seems to be calling the Unix–timer time. The function time, which can be called from Fortran or C, gives back an integervalue which is the systemtime in seconds based on a fixed date. This wallclock time has a much smaller accuracy then the hardware timers. In the case of micromeasurements in the range of a few seconds or below, one has to take care about all the possible effects on timing, we have discussed.

2 Serial programs on the KSR1

The KSR1 in Mannheim has 32 Processors (cells), which are connected by the Allcache engine in a ringlike architekture. Each cell has a local memory called 'local cache' of 32MB and a cache called 'local subcache' of 0.5MB, half of it (256KB) can be used for data. The exchange of data between cells is organized in subpages of 128B (16 words). The accesstime to a word in the subcache is 2 cycles, to a word in the local cache is 20 cycles and to a word in a remote cache is about 140 cycles. Due to this complex memory hierachy and the two orders of magnitude difference in the access times, the data access pattern have in general big influence on the performance.
The performance results for the simplest ways of matrix–multiplication for different loop–ordering are shown in figure 1. One can clearly see the big impact of loop–ordering on the

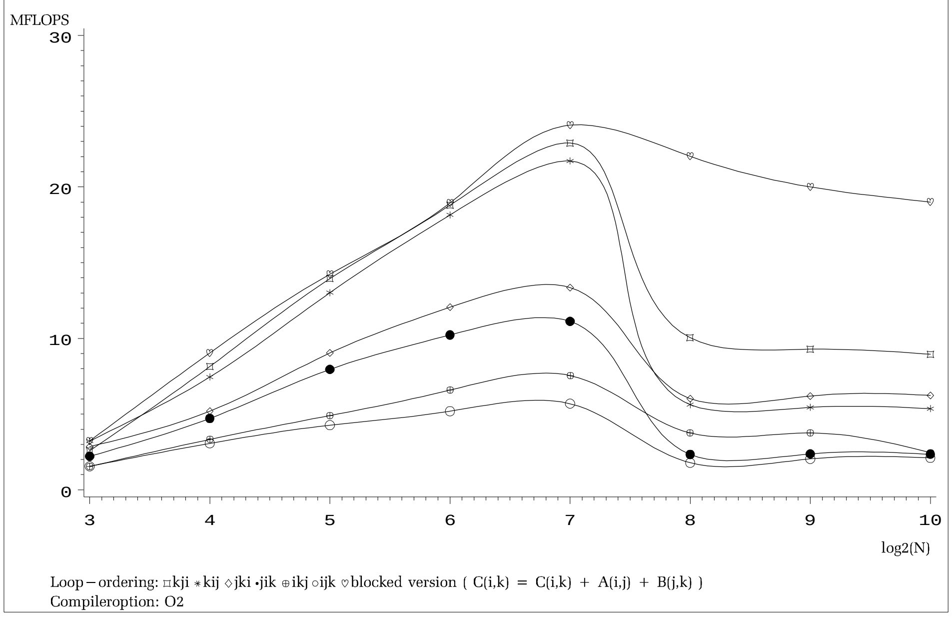


Figure 1: Influence of loop−ordering
NxN−matrix−multiplication (serial case)

performance. This is an effect of the very different memory–access–patterns of the different orderings and shows that the compiler is not yet able to substitute this simple example of a standard operation by library calls or specialized code–fragments. The maximal performance is reached for matrix–sizes of about 100x100. In this case the 240KB data for the three matrices still fits completely in the subcache. For bigger matrices the performance decreases due to subcachemisses. Above matrix–sizes of 1000x1000 (24MB data) pagefaults starts to happen, as the local cache can no longer hold all the data.
For such bigger matrices the performance is dominated by the amount of memory traffic, rather than the number of floating point operations involved. One way of achieving a better efficiency is to build blocks of your data. The idea of 'blocking' is to divide a matrix in several submatrices, where each of them fits completely in the cache. Then one can load a submatrix in the cache and perform all the computations for this submatrix. By doing this serially for all submatrices one gets a better performance, due to a smaller amount of cache–faults. An example for such a blocked version of the matrix–multiplication is given below:

```
C   NB = NUMBER OF BLOCKS

      DO 100 JJ = 1, NB
        J1 = 1 + (JJ-1) * N/NB
        J2 = JJ * N/NB
        DO 100 K = 1, N
```

```
      DO 100 J = J1, J2
        DO 100 I = 1, N
          C(I,K) = C(I,K) + A(I,J) + B(J,K)
100 CONTINUE
```

The performance gain of this simple modification is shown in figure 1.
For optimization of Fortran– and C–programs by the compiler one can use the options –O and –O2. In former releases of the fortran–compiler however, due to compilerbugs it could happen for complex codes, that –O2–compiled programs run incorrectly. For the current release F77 1.0 all of our codes are running correct. In the simple case of matrix–multiplication one gets by using –O2 up to a factor of 5 increase in performance, when the subcache is used efficiently. However for more complex codes and data access (e.g. FFT, sparse matrices,..) in many cases the performance decreases by using –O2, that's why one still has to take care about using it.

3 Different methods for parallelisation on the KSR1

For parallelisation of Fortran programs exists three quite different methods on the KSR1:

1. inserting compilerdirectives for parallel execution
2. coding direct call to the pthread–library
3. coding with calls to a message–passing–interface (tcgmsg, pvm)

We have only considered the first two of them, as they are more natural on a shared memory machine.
In the first case one can use three different parallel constructs:

- **Tile:** It divides the iterationspace of one or nested do–loops for parallel execution. Several strategies for the distribution can be applied.
- **Parallel Section:** Totaly different sections of the codes can be executed in parallel.
- **Parallel Region:** The same section of the code run on different cells, working on different parts of the problem, based on the thread–id–number.

Parallel section and parallel region directives must be inserted by the programmer. Tile directives can also be inserted by the source–to–source–preprocessor KAP, as KAP can check wether do–loops are parallelisable or not. For inserting tile–directives one has three different possibilities:

1. **automatic tiling:** Parallelisation is completely left to KAP. The problem by doing so is, that KAP often inserts not the optimal directives, especially for short do–loops or nested do–loops KAP often chooses the wrong way.
2. **semiautomatic tiling:** The progammer inserts ptile–directives (please tile) for KAP to tell the preprocessor, which do–loops should be parallelised. The details like declaring private variables are left to KAP.
3. **manual tiling:** KAP is not used, all directives are inserted by hand.

We show these three methods on the following piece of code:

```
      DO 100 K = 1, N
        DO 100 J = 1, N
          DO 100 I = 1, N
            C(I,K) = C(I,K) + A(I,J) * B(J,K)
100   CONTINUE
```

1. KAP inserts the following directive:

   ```
   C*KSR* TILE (K,J,I, ORDER=(J))
   ```

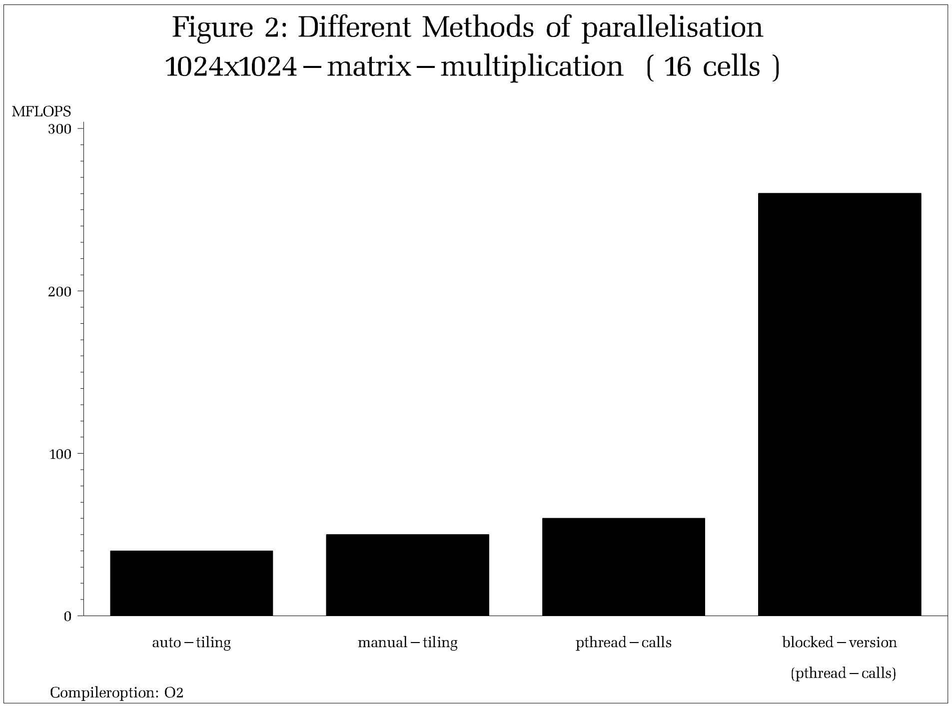
2. The programmer inserts C*KSR* PTILE (K) and KAP inserts the same directive as in 3.

3. Because of this data access one can try to parallelize the K–loop only. This is achieved by

   ```
   C*KSR* TILE (K, private = (I,J)
   ```

Inserting directives is a comfortable way of parallelisation. To parallelise by calling the pthread–library–functions one has to spend more work, but in many cases one gets better performance as can be seen in our simple example of matrix–multiplication.

```
      INTEGER NUM_THREAD, IPT(ANZ_THREADS), BOUNDS(2)

      COMMON /pthread_defaults/
     &   ipthread_attr_default, ipthread_mutexattr_default,
     &   ipthread_condattr_default, ipthread_barrierattr_default

      DIFF = N / ANZ_THREADS

C built the threads
      DO 100 NUM_THREAD = 0, ANZ_THREADS-1
        BOUNDS(1) = 1 + NUM_THREAD * DIFF
        BOUNDS(2) = (NUM_THREAD+1) * DIFF
        CALL pthread_create ( IPT (NUM_THREAD+1), ipthread_attr_default,
     &                        IWORK, BOUNDS, istat )
  100 CONTINUE

C synchronise the threads
      DO 200 NUM_THREAD = 0, ANZ_THREADS-1
        CALL pthread_join ( IPT (NUM_THREAD+1), ireturn, istat )
  200 CONTINUE
      ...

      FUNCTION IWORK ( BOUNDS )

      ... declare's ...

      DO 100 K = BOUNDS(1), BOUNDS(2)
        DO 100 J = 1, N
          DO 100 I = 1, N
```




```
         C(I,K) = C(I,K) + A(I,J) * B(J,K)
  100 CONTINUE
```

The performances of these three methods for unblocked data access are compared in figure 2 with the result of the pthread–call based parallelisation of the blocked version. As the effort for the parallelisation by using tile–directives is lower, than by calling the pthread–library and one also still has portable programs which are running unchanged on serial machines, there is certainly a trade–off between pure performance and time–to–solution and portability. Due to the shared memory of the KSR1 there exists a big range of possible ways to write parallel programs and each user can choose the one, which fits his needs best.

Computeralgebra on a KSR1 Parallel Computer

Heinz Kredel
Rechenzentrum Universität Mannheim
Postfach 10 34 62, D–68131 Mannheim, Germany
kredel@rz.uni-mannheim.de

Abstract

We give a preliminary report on the implementation of the MAS computer algebra system on a KSR1 virtual shared memory parallel computer with 32 processors. The first topics discussed are: dynamic memory management with garbage collection, a parallel integer product, and a parallel version of Buchbergers Gröbner Basis algorithm.

1 Computer Algebra

Computer algebra software is concerned with exact and symbolic computation. E.g. with the computation of the following expressions. The computation of large numbers (e.g. `1000!`), the expansion of polynomial expressions (e.g. `(x+y)^20`), the symbolic integration of functions (e.g. `int(sin(x),x)`) or the determination of all solutions to systems of algebraic equations (e.g. `solve({x+2*y = 2, x^2-3*y = 10},{x,y})`). Prominent products in this class of software are *Maple*, *Mathematica*, *Reduce* and *Derive*.

With the availability of parallel computing hardware several attempts have been made to port computer algebra software to this machines. For an overview see the conference proceedings [3, 12] and the report [11]. It turned out that shared memory multiprocessor machines [7, 6] and also workstation clusters [10] are well suited for the implementation of computer algebra software.

In our installation at Mannheim we started porting several systems (Maple, Reduce, PARI and MAS) to the KSR1 computer with 32 processors. The port of Maple was unsuccessful until now, since Maple strongly relies on 32 bit integers and pointers, and the KSR1 is a 64bit integer architecture. The port of Reduce is making slow progress also due to difficulties with 64 bit integers and pointers. For PARI the DEC Alpha version was installable. All these ports first focused on the single processor version of the programs. The next step would be the exploitation of the parallel processors and the virtual shared memory. Only for the MAS system the single processor version was relatively easy to port (as is also reported from other ALDES/SAC-2 originated systems [7, 6, 10]). So after a few weeks the development of the multiprocessor version could be started. We concentrated the porting effort to the *kernel*, the *integer product*, the construction of *Gröbner bases* and introduced some *parallel language constructs* for the interaction language (not discussed here).

The plan of the article is as follows. First we give a short introduction into the KSR1 architecture, then we present a few facts about the MAS system and start the discussion of the multiprocessor memory management. Then we discuss the development of some applications such as arbitrary precision integer product and Gröbner bases. Parallel language constructs for the interaction language are not discussed here. Finally we draw some conclusions on the suitability of the KSR1 architecture for the implementation of computer algebra software. The references given at the end are only a short selection of the actual literature on the topic.

2 KSR1 Virtual Shared Memory

The KSR1 computer is a multiprocessor computer with up to 1088 combined processor and memory boards. The CPU is a proprietary processor with a 64 bit data and a 40 bit address architecture especially designed for the use in multiprocessor machines. All CPUs have a subcache of 512 KB and are connected to a local main memory of 32 MB. So a machine with 32 processors has a total of 1 GB main memory. The distinguishing feature of the KSR1 is its hardware connection of all local main memories, the so called *ALLCACHE ENGINE*. This connection has a bandwidth of 1 GB/sec and provides the memory coherence mechanism to make the local memories look as a single globally shared memory to the software. A schematic overview of the hardware design gives figure 1.

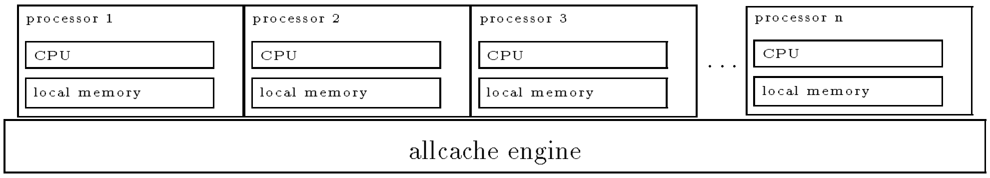


Figure 1: Hardware Architecture of KSR1

The KSR1 machine runs under OSF1 Unix, provides most GNU utilities, X–Windows. Compilers are available for C, C++ and FORTRAN 77 with KAP (semi-automatic parallelization preprocessor). Low-level concurrent programming is possible with the POSIX threads library based on the MACH kernel. In the threads model of computation an application creates several tasks (called threads) which are scheduled by the operating system to available processors (or time-sliced on processors) and which communicate among each other via the globally shared memory. Task synchronization and event signaling is provided by mutual exclusion primitives and condition variables. A schematic overview of the software model gives figure 2. In the

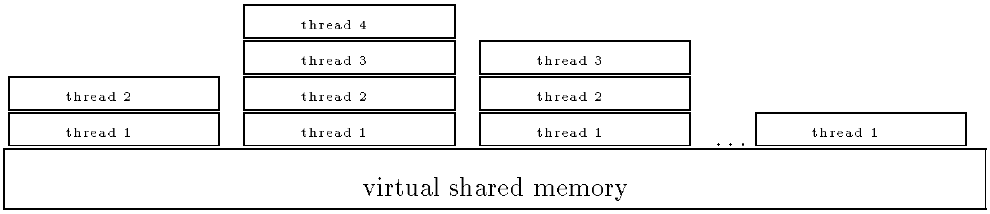


Figure 2: Software Model of KSR1

next section we will discuss how the parallel kernel of the MAS computer algebra system is implemented with the POSIX threads and virtual shared memory.

3 Dynamic Memory Management and Pthreads

In this section we give some more information on the MAS system and discuss the implementation of the MAS kernel. The MAS *Modula-2 Algebra System* was developed by the computer algebra group at the University of Passau, Germany, its current version is 0.7 as of April 1993. The system abstract says that it is *an experimental computer algebra system, combining imperative programming facilities with algebraic specification capabilities for design and study of algebraic algorithms.* The source code of the system is approximately 70 000 lines of Modula-2 code. There are about 1650 library functions and the code originated from Unix workstations, PCs and Atari STs.

The first step towards a new parallel kernel is the implementation of a parallel dynamic memory management and parallel work scheduling based on the operating system primitives. The importance of this topics stems from the fact that the computations in computer algebra are done without roundoff errors, the computer algebra software faces the problem of so called *intermediate expression swell.* Even the computation of a small expression like `(x^n-1)/(x-1)` leads to the huge expression `x^(n-1)+...+x+1` when `n` is big (e.g. 1000). So nearly all computer algebra software has some form of dynamic memory management to cope with arbitrary sized expressions and the collection of any 'garbage' expressions left over during the computation. This consideration also shows that the amount of work generated by a run of an algorithm may vary dynamically.

For dynamic memory management the MAS system uses list processing. The list processing code is contained in one Modula-2 module. The list processing memory is allocated and initialized as one large memory space (called cell space) at program start time.

A first consideration shows that the tasks (threads) generated by an application algorithm should *not be moved* between processors after they have been started. This is meaningful because a thread could reference a large portion of global cell space and if this thread would be migrated from one processor to another this data would have to be transferred to. And although the allcache engine does this transfer automatically and fast it needs more time than the usage of the local memory. So our memory model consists of a distributed list cell space on each processor and multiple threads per processor which are bound to the processor they started on. To maintain this distributed cell space, the input and output parameters for newly created tasks are copied if the subthreads start on different processors.

Garbage collection is done by the well known *mark and sweep* method, were in a first step all cells which are possibly in use are marked and then in a second step all unmarked cells are swept to a free cell list. Since the cell space is distributed the garbage collection can be performed on each processor which runs out of list memory independently of the threads on other processors. Only the threads executing on the same processor (as the garbage collecting thread) stop creating new list cells (read operations in the cell space are not interrupted). References into the cell space of other processors are ignored during the mark step. To ensure that this local mark and sweep is correct, we allow only global read access for copying data to the local cell space and then do only local update and modification of list elements. The global variables are handled by one dedicated processor. In summary the garbage collection is performed in the following steps.

1. A mark of global variables if appropriate.
2. A local mark of all stacks on all (mach-) threads on this processor and a local mark of the stack of the current thread.
3. A local sweep.

In this model the scheduling of threads is left to the operating system. However after some time we experienced that this scheduling was not efficient under our model. So we had to introduce a global task queue from which started threads take some new work when they have finished their last assignment. The disadvantage is now the queue bottleneck but the load balance for our application programs improved considerably. In figure 3 we have summarized the overhead which is introduced by thread creation in our first and second scheduling model together with the overhead which is introduced by the POSIX thread mechanism. The POSIX threads are named 'pthread' and the new MAS thread layer is named 'mthread' ('1' for the first and '2' for the second scheduling method).

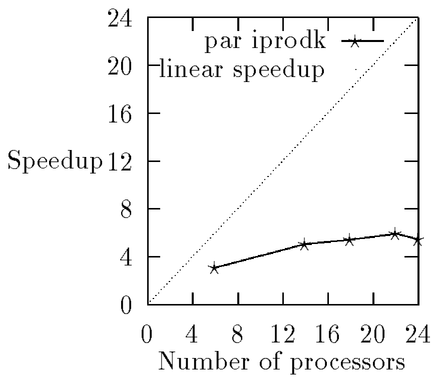
		quotient
function call	/ assignment	10.4
pthread create	/ function call	40.2
pthread join	/ function call	7.8
pthread	/ function call	48.1
mthread 1 create	/ pthread create	3.7
mthread 1 join	/ pthread join	4.2
mthread 1	/ pthread	3.8
mthread 2	/ pthread	0.6 - 1.1

Figure 3: Overhead of thread creation and scheduling

This figures also raise the question of algorithm grain size, i.e. the number of basic computation steps performed within a parallel task. And as the figures indicate we should have at least 50 to 100 function calls within a pthread to equal the time lost during the creation and destruction of a pthread. In the second mthread model almost no new overhead is introduced.
Having designed and implemented a list processing kernel our next task is the development of algorithms with suitable grain size to make efficient use of all processors during the computation.

4 Integer Product

The first application program which uses the new parallel kernel is the arbitrary precision integer multiplication. The method is due to Karatsuba and uses the identity $a * b = (a_1\beta + a_0) * (b_1\beta + b_0) = a_1b_1\beta^2 + (a_1b_1 - (a_1 - a_0)(b_1 - b_0) + a_0b_0)\beta + a_0b_0$ to recursively compute the product $a * b$ with 3 multiplications, 4 additions and some 'shifting'. In the sequential version it is known that Karatsubas method is superior to the ordinary multiplication if the sizes of the integers exceed 16 machine words. The parallel version starts a new thread for the computation of 2 of this subproducts if the size of the integers is greater than 64 words. If the size of the integers becomes smaller during recursion, first the sequential Karatsuba multiplication and then the ordinary multiplication is used. Also for very large integers, if 'much' more threads have been created than processors are available, the sequential versions are used. The preferred parallel/sequential scheduling method can be determined using a function exported from the MAS kernel.
For the first timings see figure 4. The timings are measured in seconds by the '`time`' function of the standard C library. An alternative are the functions '`user_timer`' and '`all_timer`' from the KSR1 timer functions which measure the user time and *elapsed* time spend in a specific thread. Although `time` includes all system overhead it is preferable over the others



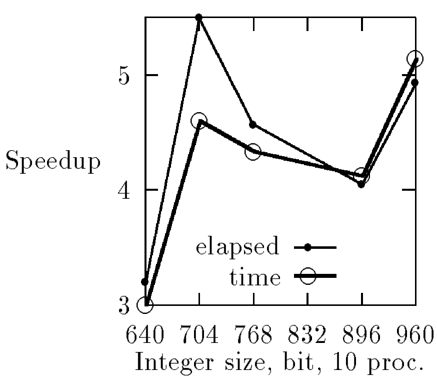


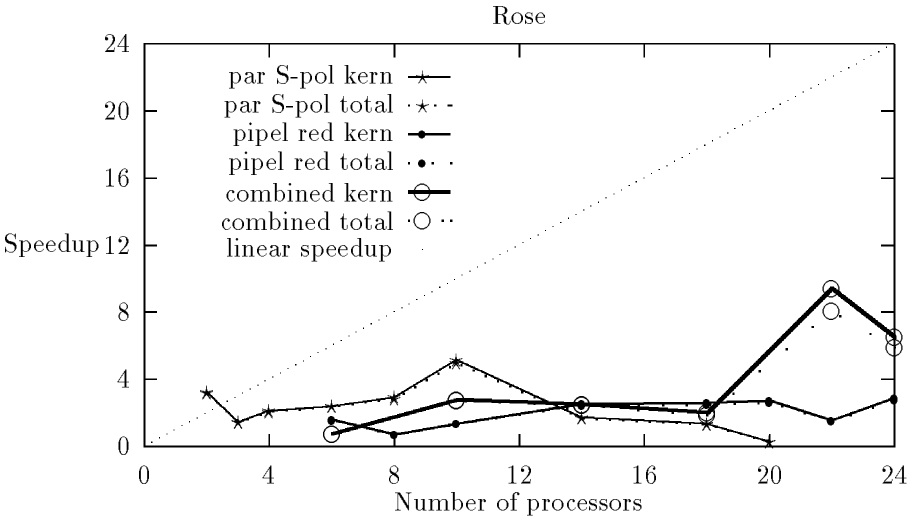
Figure 4: Integer Product

since it measures the maximal time over all threads and this is the time one experiences in an application. The speedups are comparable to the values reported by [7] for 12 processors.

5 Gröbner Bases

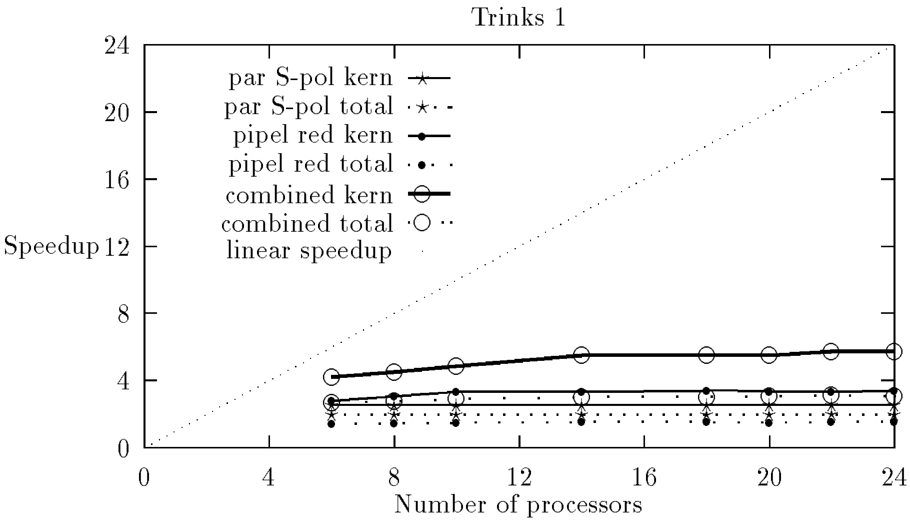
The second algorithm chosen for parallelization is Buchberger's algorithm for the computation of Gröbner bases. Roughly speaking Gröbner bases play the same role for the solution of systems of algebraic equations as the diagonal matrices, obtained by Gaussian elimination, for systems of linear equations (see e.g. [1]). It is known that the problem of computing Gröbner bases is exponential-space hard and also NP hard [1]. Since by the *parallel computation thesis* [4]: "Time-bounded parallel (Turing) machines are polynomially equivalent to space-bounded sequential (Turing) machines", one should not expect a parallel polynomial time solution for the computation of Gröbner bases. Nevertheless any improvement of this algorithm is of great importance and one would idealy like to obtain a solution in $\frac{1}{p}$-th of the time if p processors are utilized.

The implementation of the parallel version is based on the sequential Buchberger algorithm as implemented in MAS. For the parallelization there is one 'natural' choice, namely the reduction (a kind of polynomial division with respect to several divisors) of of S-polynomials (critical pairs) in concurrent steps (see e.g. [5, 9]). However it turned out that this way of parallelization is *to coarse* to make efficient use of all processors during the computation (see the timings given in the figures). To find a finer grain size it was proposed to perform a kind of pipelined reduction [9]. In this proposal each division step in the reduction is performed by a new thread. Even finer grain sizes on monomial arithmetic level did not improve the performance in the tested examples. At this time the combination of the parallel reduction of S-polynomials with the pipelined reduction of them showed the best speedup figures. For the timings for some standard test examples of [2] see figures 5 and 6. Since the figures show a problem dependent maximal parallelization degree, it seems that the grain size is still to coarse. The speedups are comparable to the values reported by [5] for 16 processors and to the values reported by [9] for 25 processors in the Trinks 1 example.



kern = parallel part, total = parallel and sequential part

Figure 5: Example Rose



kern = parallel part, total = parallel and sequential part

Figure 6: Example Trinks 1

Although the algorithms will be discussed in detail elsewhere some remarks are in order. The original Buchberger algorithm is not very complicated (opposed to its correctness proof), the parallel S-polynomial and the pipelined reduction algorithms are quite complicated. The new algorithms require all sorts of communication patterns (from shared variables to message passing with dynamic channel assignment) synchronization efforts and flow of control optimizations. But a satisfactory solution still suffers from a poor processor utilization of about $40 - 60\%$ in the tested examples. The deficiencies could come from design decisions in the polynomial representation (which has been optimized for sequential machines), from the design of the algorithm itself, from the specific example or from a insufficient understanding of the machine architecture and scheduling mechanisms. So, much research is still needed.

6 Conclusion

The efforts made in porting the approximately 70 000 lines Modula-2 code are as follows. The number of lines changed were 50 in the Modula-2 to C translator, 200 + 500 new in the MAS kernel and starting from 100 in application programs. The porting effort was approximately 1 person 2 weeks for the 1 processor version and 1 person 2-3 month research for the n processor version. As we have seen the porting of the dynamic memory management needs insight into *architecture* of the machine and into *scheduling* strategy of the operating system. E.g. for the garbage collection the register contents of all threads on a specific processor must be examined. The KSR1 shared memory concept makes it easy to program this and for the *OSF1 operating system* there was enough documentation, to extract the required information on the software architecture.

The challenge in parallel computer algebra is to design a parallel list processing with *garbage collection* and to develop algorithms with suitable (adaptable) *grain size* to make efficient use of all processors during the computation. For specific examples it is possible to obtain the expected figures on the KSR1. However it is difficult to obtain *sustained speedup* across the different subproblems which are dynamically generated. For this ongoing research the KSR1 machines provide a well suited architecture to study a wide variety of algorithms.

References

[1] Th. Becker, V. Weispfenning, with H. Kredel, *Gröbner Bases.* Springer, GTM 141, 1993.

[2] W. Böge, R. Gebauer, H. Kredel, *Some Examples for Solving Systems of Algebraic Equations by Calculating Gröbner Bases.* J. Symb. Comp., No. 1, pp 83-98, 1986.

[3] J. Della Dora, J. Fitch (eds.), *Computer Algebra and Parallelism.* Academic Press, London, 1989.

[4] Leslie M. Goldschlager, *A Universal Interconnection Pattern for Parallel Computers.* J. ACM, Vol. 29, No. 3, July 1982, pp 1073-1086.

[5] David J. Hawley, *A Buchberger Algorithm for Distributed Memory Multi-Processors.* Springer LNCS 591, pp 385-390, 1992.

[6] H. Hong, A. Neubacher, W. Schreiner, *The Design of the SACLIB/PACLIB Kernels.* Proc. DISCO '93, Springer LNCS 722, pp 288-302, 1993.

[7] W. W. Küchlin, *PARSAC-2: A parallel SAC-2 based on threads.* Proc. AAECC-8, Springer LNCS 508, pp 341-353, 1990.

[8] Computer Algebra Group Passau, *Modula-2 Algebra System, Version 0.7.* See eg. [11], pp 222-228.

[9] Stephen A. Schwab, *Extended Parallelism in the Gröbner Basis Algorithm.* Int. J. of Parallel Programming, Vol. 21, No. 1. 1992, pp 39-66.

[10] Steffen Seitz, *Algebraic Computing on a Local Net.* In [12], pp 19-31.

[11] V. Weispfenning, J. Grabmeier (eds.), *Computeralgebra in Deutschland.* Fachgruppe Computeralgebra der GI, DMV, GAMM, 1993. Erhältlich bei GI, Godesberger Allee 99, Bonn.

[12] R.E. Zippel (ed.), *Computer Algebra and Parallelism.* Springer LNCS 584, 1990.

Parallel Random Number Generation on KSR1*

Michael Hennecke
Rechenzentrum, Universität Karlsruhe, Postfach 6980, D–76128 Karlsruhe
email: hennecke@rz.uni-karlsruhe.de

Abstract

In this note, parallel random number generation on the KSR1 parallel computer is discussed. The random number generators available for the KSR1 are sketched, and performance measurements done at the KSR1-32 system in Mannheim are presented, focusing on the KSR1 implementation of a parallel random number generation method proposed recently[7]. The key feature of this generator is that the stream of random numbers generated is exactly the same than for sequential execution, which is also advantageous with respect to the quality of the resulting random number stream. Its performance is about $2.3 \cdot 10^6$ integer numbers per second and processor for data residing in the local caches, with accelerations if the data fits completely into the subcaches.

1 Introduction and survey

Random number generation is repeatedly discussed in the literature, the standard reference still being Knuth[12] supplemented by some recent surveys like [9, 13, 6, 2]. Obviously, parallel computing poses new problems of qualitative and quantitative nature, some of which are addressed in this article. In this section, the current situation for the KSR1 is summarized and the suitability of the existing generators for parallel processing is sketched. In section 2, some principles and performance measurements of the parallelization method of [7] are presented.

For the time being, the following generators are available with the KSR1 system software:

- KSR C[10] supplies three different random number generators (see man-pages also): the standard `rand` generator which is a 32-bit multiplicative linear congruential (MLC) generator supplying 16-bit numbers (32-bit in BSD-mode), the `drand48` package which is a reconfigurable 48-bit MLC generator supplying 32-bit numbers, and a KSR-specific generator `random` called "nonlinear additive feedback RNG" with user-selectable size of the state array. No further details on its algorithm are available.

- KSR Fortran[11] contains two different random number generators: the UNIX-external generator `rand` (`drand`, `irand`), and the KSR-external generator `random` (`drandm`, `irandm`). No information on the algorithms used is given. Their speed is very low: more than 20 μs per integer number and about 30 μs per floating point number.

Only one random number per call can be produced by the above generators. Although there are reentrant versions of these functions, they are all *scalar* generators. They cannot be used

*Supported by ODIN (Optimale Datenmodelle und Algorithmen für Ingenieur- und Naturwissenschaften auf Höchstleistungsrechnern), a joint project of the University of Karlsruhe and Siemens Nixdorf Informationssysteme AG.

safely in a parallel application because of the possibility of overlapping sequences, which only differ in their initial states.[1] There is another generator for scalar use, which has a large modulus implemented portably for 32-bit machines also, and is likely to be much better tested than the ones given above:

- The NAG Fortran Library (Mark 15) is available for the KSR1. Its random number routines are based on the generator[14]

$$x_i = (13^{13} \cdot x_{i-1}) \; mod \; 2^{59} \tag{1}$$

  The generator has a period of 2^{57}, the timings for floating point numbers are about 3.5 μs per number when generating one number by `G05CAE`, and from 1.7 to 2.0 μs when generating a vector of random numbers by `G05FAE`, depending on the vector length.

For parallel applications, it should be guaranteed that the sequences generated in different tasks are not only different (forced by different initial states), but also *disjoint*. There are two established methods to achieve this: either using different generators for each task, or using disjoint substreams of one generator. The latter requires some knowledge in order to be able to skip a large distance of the generator's sequence.[2] The following generators represent these two different approaches:

- Based on theoretical work of Percus and Kalos[15], a parallel random number generator called **prng** has been developed at the Cornell Theory Center[3]. This is a set of multiplicative linear congruential generators of the form

$$x_{i,p} = (11^{13} \cdot x_{i-1,p} + c_p) \; mod \; 2^{64} \tag{2}$$

  where different threads p are using different additive terms c_p. The current (November 1993) implementation supports 1022 different values of c_p. The generator supplies only one floating point number per call, which takes about 1.9 μs.

- KSR also proposes[16] to use a multiplicative linear congruential generator of the form

$$x_i = (a \cdot x_{i-1} + 1) \; mod \; 2^{64} \tag{3}$$

  with $a = 6\,364\,136\,223\,846\,793\,005$. Different tasks in a parallel application shall use disjoint sections of the random number stream, initiated by starting seeds that are $k = 6\,369\,051\,672\,525\,833$ iterations apart. There can be $2^{64}/k > 1088$ disjoint sections. No timings are available since this generator was proposed for inline use.

Although these solutions are more reliable than simply using the scalar generators with different seeds, there are always two problems when using those parallel generators: it is not obvious how to judge the quality of the combined random number streams,[3] and the fact that a fixed, task-based parallel generation method is used has the consequence that the resulting random numbers depend on the actual number of tasks used to solve the problem at hand. These two problems are taken into account by a new method of parallel random number generation proposed recently by the author[7]. It is described below.

[1] It should be noted that *every* generator can be used in parallel with this strategy if the danger of overlapping sequences (which decreases with increasing period) is accepted.

[2] This knowledge *is* available for MLC generators[2, 17] and for lagged Fibonacci generators using XOR[1].

[3] Bad experiences can be found in de Matteis et.al.[5, 4] for different substreams of one MLC generator.

2 The proposed parallel generator

The static, task-based parallelization schemes existing so far have the disadvantage that changing the number of tasks (threads, processors, ...) also changes the random number sequence that is produced. Although simulation results are only statistical averages anyway, it is desirable to be able to reproduce simulation runs with exactly the same results, using different processor configurations.
Additionally, in programming models like High Performance Fortran[8] and shared memory architectures like the KSR1 which do not necessarily break up the work into rigidly defined tasks, is is desirable that a parallel generator is available that is not controlled by a task index like `ipr_mid()`, but by the more general concept of linking the random number vector to the application data using it. The user then "only" has to specify the parallelism and data distribution of the application, and the generator filling the (similarly distributed) random number vector has to adhere to this information.
In ref. [7], these concepts are discussed in detail. The algorithm used is the multiplicative linear congruential generator, with the constants used in the the NAG generator (1). The MLC method is both well understood theoretically and suitable for this kind of parallelization: The key observation is that applying the MLC algorithm n times and collecting expressions containing the additive term c leads to

$$x_{i+n} = (a^n x_i + c_n) \; mod \; m \qquad with \qquad c_n = c \cdot \left(\sum_{k=0}^{n-1} a^k \right) \; mod \; m \; , \tag{4}$$

for arbitrary $i \geq 0$ and $n \geq 1$. Computing a^n is $\mathcal{O}(\log n)$, whereas c_n is $\mathcal{O}(n \cdot \log n)$ if $c \neq 0$. In order to allow the user to control the distribution of data onto processors (pthreads), the generator is supplemented by a routine to explicitly specify the distribution in a way similar to High Performance Fortran syntax. After having choosen a particular distribution of the random number vector, the generator can be called like a sequential generator, it uses equation (4) internally to generate the distributed vector from an initial seed x_0 in parallel. Apart from the precise form of the subroutine parameters, it is fully equivalent to the sequential NAG generator. The performance of the generator has been investigated in two stages. As a first step, the generator has been executed sequentially on one KSR1 processor in order to inspect the single-node performance. Afterwards, the generator has been executed in parallel on up to 16 processors.

Theoretically, the following can be expected from equation (1) or its parallel equivalent: The Integer Processing Unit (IPU) needs two integer operations to *compute* x_{i+1} from x_i (multiplication by a and modulo-m calculation by bit manipulation), the cycle time of 50 ns thus implies that the generator cannot produce more than 10 million random numbers per second. But the KSR1 functional units are pipelined: the integer multiplication is done in a 4-step pipeline, whereas clearing the most significant bits is done in a one-cycle operation. So the recurrence inherent in the generator algorithm would reduce the speed to 4 million numbers per second, and even when manually "unrolling" the recurrence it is difficult (but possible) to interlace the pipelined multiplications with the bit manipulations. By non-optimal loop unrolling and register usage, the compiler actually degrades the achievable speed by 30%.
The results have to be *stored* by the Cell Execution Unit (CEU). For vectors completely fitting into the subcache, the store operation does not delay execution since the subcache has a latency of 2 cycles, which is exactly the time required for the computations. For larger vectors a performance loss is to be expected, caused by the higher latency (20–24 cycles) of the local cache. An even larger drop appears if the vectors are so large that they cannot be held in the processor's local cache.

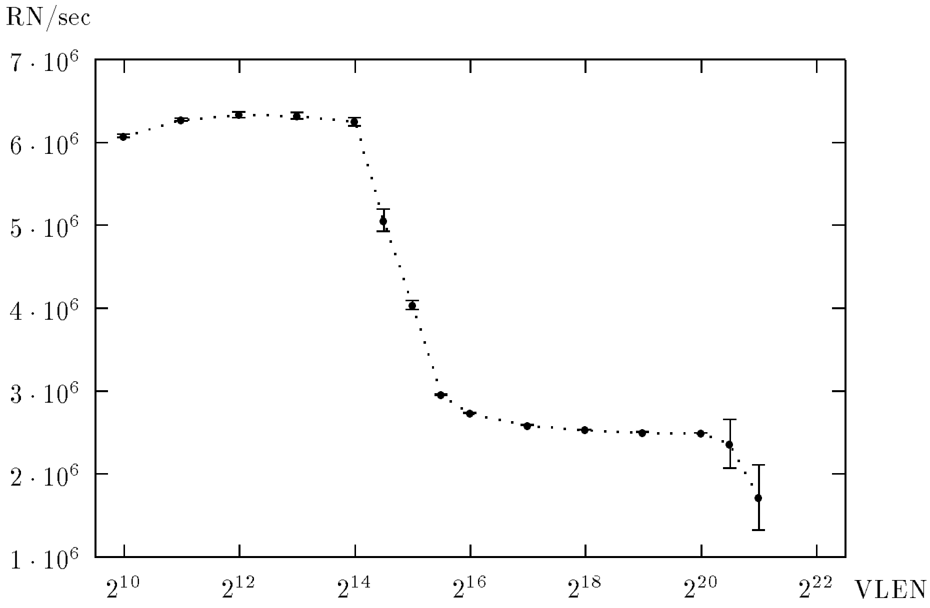


Figure 1: Performance of the parallel random number generator on KSR1 for BLOCK distribution, executing sequentially on one processor. Shown is the random number generation speed for integer random numbers versus vector length. All data are averages over 5 measurements.

The experimental results are shown in Figure 1, where the speed of scalar random number generation is shown as a function of the length of the vector that is filled. They agree well with the theoretical expectations: In subcache, a performance of about 6.3 million random numbers per second is achieved, and the drop to about 2.5 million random numbers around the vector length 2^{15} is due to the increasing number of subcache misses.[4] The results for parallel execution of the generator are shown in Figure 2. For those vectors residing in the processors local caches, the performance per processor remains almost constant when increasing the number of processors. This is expected theoretically, since the algorithm does not need any communication. However, the performance gain for small vectors fitting into the subcaches is not very good. The speed expected from the scalar execution results is never reached, and the slight benefits of subcache usage are rapidly compensated by the overhead resulting from the thread management, which dominates for small vector lengths.

To conclude this section, some details not visible in the data shown above should be mentioned to give an impression of those details of the KSR1 that should be taken into account for time-critical applications:

- All measurements presented here have been done on an exclusively allocated processor set: The KSR1 is capable of multiprocessing on each processor, but running two numerically intensive processes in parallel on one node normally results in performance losses of more than one order of magnitude. For an effective usage of the KSR1 system, it is therefore necessary to execute those programs one at a time.

[4] For very small vectors, the overhead due to the subroutine calls and loop control becomes visible. Results for very large vectors show that the processor's local cache starts swapping.

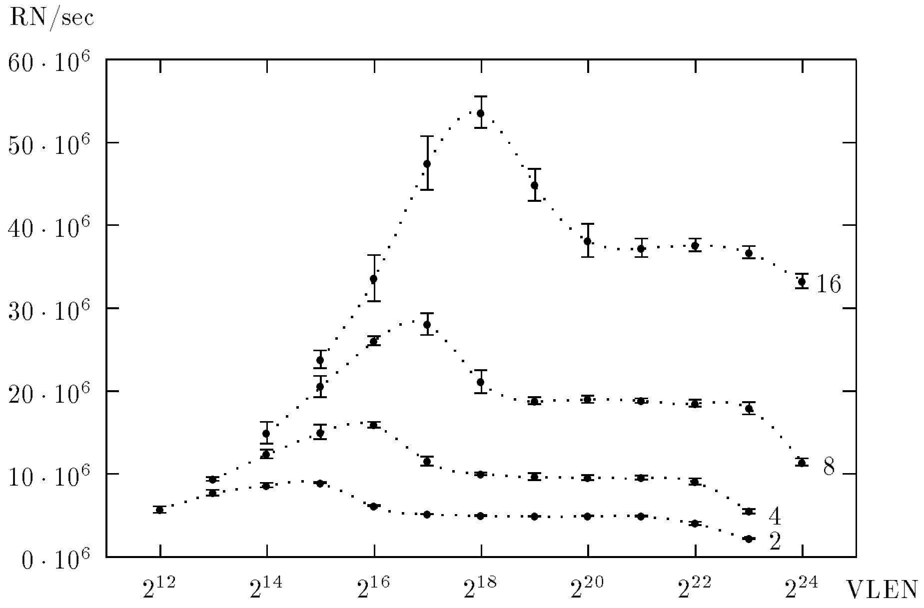


Figure 2: Performance of the parallel random number generator on KSR1 for BLOCK distribution onto 2, 4, 8 and 16 processors. Shown is the random number generation speed for integer random numbers versus vector length. All data are averages over 6 measurements.

- Caused by the KSR1 memory model, there is substantial system activity when accessing large distributed data structures the first time. For timing measurements not concerned with initial overhead, it is essential that data which is repeatedly accessed has been distributed to the KSR1 nodes before the measurements start.

- Care must be taken for data distributions where different data items are accessed in parallel which reside in the same "subpage" (128 bytes) which is the elementary memory unit for which access rights like write permission are set. See ref. [7] for consequences concerning the parallel random number generator.

Finally, it should be noted that no data except the random number vector was accessed, which does not make sense except for timing measurements. However, Figure 2 can also be used to estimate timings for applications including user data.

3 Summary

In this note, parallel random number generation on the KSR1 has been reviewed, and performance measurements of the parallel generator introduced in ref. [7] have been presented. Being equivalent to the sequential algorithm, this generator offers a higher degree of reproducibility than previous parallel generators. Comparing the timings to existing MLC generators shows that significant performance improvements have been achieved by generating random number *vectors* and taking the *pipeline* architecture of the KSR1 into account.

The author would like to thank O. Haan for many fruitful discussions on random number generation, R. Schumacher for various informations and technical support on KSR1, and H. Strauss for correspondence on KSR random number material.

References

[1] S. Aluru, G.M. Prabhu, and J. Gustafson. A random number generator for parallel computers. *Parallel Computing 18*, pages 839–847, 1992.

[2] S.L. Anderson. Random number generators on vector computers and other advanced architectures. *SIAM Review 32*, pages 221–251, 1990.

[3] D. Bergmark et.al. Parallel random number generator `prng`. Available by anonymous ftp from `info.tc.cornell.edu` in `pub/utilities/prng.*.tar`.

[4] A. de Matteis and S. Pagnutti. Parallelization of random number generators and long-range correlations. *Numerische Mathematik 53*, pages 595–608, 1988.

[5] A. de Matteis and S. Pagnutti. A class of parallel random number generators. *Parallel Computing 13*, pages 193–198, 1990.

[6] P. l'Ecuyer. Random numbers for simulation. *Communications of the ACM 33:10*, pages 85–97, 1990.

[7] M. Hennecke. Parallel random number generation and High Performance Fortran. Submitted to *Parallel Computing*.

[8] High Performance Fortran Forum. High Performance Fortran Language Specification. Appeared in *Scientific Programming*, vol. 2, no. 1, June 1993. Also available by anonymous ftp from `titan.cs.rice.edu` in `public/HPFF/draft` and `ftp.gmd.de` in `hpf-europe`.

[9] F. James. A review of pseudorandom number generators. *Computer Physics Communications 60*, pages 329–344, 1990.

[10] Kendall Square Research. *KSR C Programming*, 15 February 1992.

[11] Kendall Square Research. *KSR Fortran Programming*, 15 February 1992.

[12] D.E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison Wesley, Reading, Massachusetts, second edition, 1981.

[13] G. Marsaglia. A current review of random number generators. In L. Billard, editor, *Proc. of Computer Science and Statistics: The Interface*, Amsterdam, New York, 1985. North-Holland.

[14] Numerical Algorithm Group, Oxford. *NAG Fortran Library Manual, Mark 15*, 1st edition, June 1991. Chapter G05.

[15] O.E. Percus and M.H. Kalos. Random numbers generators for MIMD parallel processors. *Journal of Parallel and Distributed Computing 6*, pages 477–497, 1989.

[16] H. Strauss (KSR Germany). Private communication.

[17] A. van der Steen. Portable parallel generation of random numbers. *Supercomputer 35*, pages 18–20, 1989.

The efficiency of the KSR1 for numerical algorithms

Hartmut Häfner and Willi Schönauer
Rechenzentrum Universität Karlsruhe, Postfach 6980
D–76128 Karlsruhe, Germany

The KSR1 is the first MIMD distributed memory parallel computer with a virtual shared memory [3,4,5]. It works with standard UNIX on each node. Thus multitasking is possible on every single node. Exemplarily different parallelized implementations of the matrix mutiplication and of the Gauss algorithm are investigated. For these algoritms the gap in efficiency between the peak performance of the KSR-1 and the real performance of an application program is pointed out. To explain the different sources for the loss of efficiency a 'global performance formula' - developed by W. Schoenauer [1,2]- is used. This formula factorizes all types of losses and is explained here briefly:

$$\overline{r}_{real} = \underbrace{(1000/\tau[nsec]) * 2 * P}_{\text{theoretical peak}} \underbrace{\begin{array}{l} * f_{cache\_bottleneck} * f_{memory\_bottleneck} * f_{compiler} \\ * f_{parallel\_part} * f_{communication} * f_{load\_balance} \\ * f_{utilization}[MFLOPS] \end{array}}_{\text{reduction factors}}$$

$\overline{r}_{real}$ is the real performance for a certain class of applications; it is the theoretical peak performance reduced by the now explained reduction factors. P is the number of arithmetic groups (∗,+).

$$f_{single\_processor} = f_{cache\_bottleneck} * f_{memory\_bottleneck} * f_{compiler}$$

$f_{single\_processor}$ reduces the peak performance for a single processor of the KSR-1, which is 40 MFLOPS, to the real performance. To fix $f_{single\_processor}$ for a certain class of applications one only has to know the ratio of the different basic operation s for which then performance measurements must be available. Three parameters determine $f_{single\_processor}$:

- $f_{cache\_bottleneck}$ reduces the performance of a single processor in proportion to [needed words per cycle to/from cache]:[transferable words per cycle to/from cache];
- $f_{memory\_bottleneck}$ is due to the idling processor if cache misses occur and therefore cache lines must be transferred between cache and memory;
- $f_{compiler}$ is due to the compiler, which mostly does not produce optimal code.

  To interpret the performance of a parallel application three more reduction parameters concerning the parallelization must be considered:

- $f_{parallel\_part}$ reduces the performance if the parallelizable part (fine grain parallelism) of an application is less than one (Amdahl's law). This factor can be determined theoretically and depends on the the number of processors and the parallelizable part.

- $f_{communication}$ is the most complex reduction factor. For parallel computers without asynchronous communication it only depends on the number and the length of the messages and on the communication-bandwidth. On the KSR-1 the communication-bandwidth additionally depends on the amount of communication of the other jobs. For parallel computers with asynchronous communication it also depends on the communication-to-computation-overlapping ratio. This ratio again depends on the volume-to-surface ratio of an algorithm and is in the responsibility of the user.

- $f_{load\_balance}$ reduces the peak performance by bad load balancing (coarse grain parallelism). This parameter can be determined theoretically for a single application.

All the above mentioned parameters reduce the peak performance of the user applications. $f_{utilization}$ strongly depends on the process/thread scheduling. A good scheduling could enhance the performance with respect to the master process user time because of a possible lower impact of $f_{load\_balance}$. On the contrary an unfavourable scheduling could decrease the performance strongly. Thus one can take futilization as the factor which shows the efficiency of the process/thread scheduling in a multiprogramming environment of a parallel computer. $f_{utilization}$ is the only factor, which theoretically can be greater than one, but the product $f_{load\_balance} * f_{utilization}$ is always less than one.

1 THE PERFORMANCE ON A SINGLE PROCESSOR

We have measured some basic operations which are relevant to numerical algorithms. In the following table you can see the measurements and the reduction factors – belonging to each basic operation – for a single processor.

vector	measurement with vector length								
operation	$n = 1$	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$	$n = 100000$	$f_{cache\_b.}$	$f_{memory\_b.}$	$f_{compiler}$
multiplication	1.0	4.5	5.2	5.7	4.2	1.5	0.333	$0.23 - 1$	0.86
linked triad	1.6	6.7	10.2	11.1	8.5	2.9	0.333	$0.23 - 1$	0.83
scalar product	1.7	7.7	13.5	15.6	13.5	4.9	0.5	$0.25 - 1$	0.78
vector triad	1.5	6.0	7.6	6.9	3.5	2.1	0.25	$0.21 - 1$	0.76

Table 1: The performance of basic operations in MFLOPS and the corresponding reduction factors.

$f_{cache\_bottleneck}$ is determined theoretically. If all data of the vector operations can be stored in the cache, $f_{memory\_bottleneck} = 1$ and the indicated values (< 1) for $f_{compiler}$ hold. The more cache misses occur, the smaller becomes $f_{memory\_bottleneck}$ and the greater becomes $f_{compiler}$. If all data must be loaded from the memory, the indicated values (< 1) for $f_{memory\_bottleneck}$ hold and $f_{compiler} = 1$ is valid.
Additionally we have measured four different implementations of the matrix multiplication listed in table 2.

matrix	measurement with matrix length							
multiplication	50×50	100×100	350×350	500×500	1000×1000	$f_{cache\_b.}$	$f_{memory\_b.}$	$f_{compiler}$
a) scalar product	20.0 (5.4)	19.6 (6.8)	3.3	3.4	1.5	0.5	$0.17 - 1$	1
b) columnwise	21.9 (16.1)	22.7 (20.8)	8.8	8.8	5.8	1	$0.39 - 1$	0.57
c) b)+cache opt.	20.8 (14.9)	20.0 (17.9)	18.4	17.1	8.5	1	$0.82 - 1$	0.52
d) c)+unroll. on j	25.0 (17.9)	24.4 (21.3)	21.7	19.3	10.0	1	$0.77 - 1$	0.63

Table 2: The performance of different implementations of the matrix multiplication in MFLOPS and the corresponding reduction factors. The values in the parentheses are valid, if the matrices are dimensioned as 500×500-matrices.

The basic vector operation of the columnwise matrix multiplication is the contracting linked triad. As this operation requires only one word per cycle to load, $f_{cache\_bottleneck} = 1$ holds. For the matrix multiplication with cache optimization $f_{memory\_bottleneck}$ tends to one. For large matrices (e.g. 1000×1000-matrices) the performance drops because of page faults in the local memory. Then the needed data have to be loaded either from a remote memory or from the disk.

2 THE PARALLEL MATRIX MULTIPLICATION

For the parallelized columnwise matrix multiplication $f_{parallel\_part} = 1$ and $f_{load\_balancing} = 1$ holds and the volume-to-surface ratio is proportional to n. Thus the communication can be hidden behind the computation for large matrices.

2.1 The parallel matrix multiplication by tiling

We have measured the above mentioned implementations of the columnwise matrix multiplication on a processor set with 16 processors.

matrix mult. by	measurement with matrix size		speedup	
semi-auto. tiling	500×500	1000×1000	500×500	1000×1000
a) columnwise	123.8	74.8	14.1	12.9
b) a)+cache opt.	283.7	277.8	16.6	32.6
c) b)+unroll. on j	312.5	300.3	16.2	30.0

Table 3: The performance of different impementations of the parallel matrix multiplication by semi-automatic tiling in MFLOPS and the corresponding speedup factors.

The semi-automatic tiling (compiler directives inserted by the user) has been performed on the index j. For large matrices the real speedup can not be determined because of page faults for the non parallelized program. The parallelized program runs with less page faults because the program needs less memory in the local memory. Thus the performance rates of a single processor are higher for the parallelized program than for the non parallelized program.

matrix mult. by	measurement with matrix size		speedup	
automatic tiling	500×500	1000×1000	500×500	1000×1000
a) columnwise	26.2	30.6	3.0	5.3
b) a)+cache opt.	130.2	63.0	7.6	7.4
c) b)+unroll. on j	195.3	95.1	10.1	9.5

Table 4: The performance of different impementations of the parallel matrix multiplication by automatic tiling in MFLOPS and the corresponding speedup factors.

The automatic tiling (no compiler directives are inserted by the user) has been performed on the indices i and k. To achieve a good performance it is evident that the user must think about the parallelization of his program and must insert the tiling directives into the code, whereas it is to say that many of the needed parameters are automatically inserted by the compiling system. Remark: KSR achieves 380 MFLOPS for the matrix multiplication on 16 processors. The library routine has about 4000 (!) lines of code.

2.2 The parallel matrix multiplication by the message passing library TCGMSG

We have measured the parallel matrix multiplication with the alternating buffer technique on 16 processors.

matrix mult. by	measurement with matrix size		speedup	
TCGMSG ($P = 16$)	500×500	1000×1000	500×500	1000×1000
a) columnwise	105	99	11.9	17.1
b) a)+cache opt.	105	133	6.1	15.6
c) b)+unroll. on j	113	136	5.8	13.6

Table 5: The performance of different implementations of the parallel matrix multiplication by TCGMSG in MFLOPS and the corresponding speedup factors.

If the message passing library supports asynchronous communication, it can be completely hidden behind the computation with the alternating buffer technique for large matrices. Thus we expected nearly the same MFLOPS-rates as for semi-automatic tiling. But TCGMSG does not support an asynchronous receive of messages and therefore $f_{communication} \ll 0.5$ holds. Again the real speedup, which in almost every case is much lower than the indicated values, can not be determined because of the memory hierarchy of the KSR-1 (local cache – local memory – remote memory – disk).

3 THE PARALLEL GAUSS ALGORITHM

For the "wrap around by columns"-algorithm $f_{parallel\_part}$ is nearly one and $f_{load\_balancing} = 1$ is valid and the volume-to-surface ratio is proportional to n for the forward elimination. On 1 processor we have measured the following MFLOPS-rates for the forward elimination/backward substitution/whole algorithm: $n = 1000$: 3.6/6.2/3.6, $n = 2000$: 2.2/2.8/2.2 . There is no cache reuse for the whole algorithm. Thus the low performance rates are explainable (see table 1 in row "linked triad"). On 16 processors we have measured the following MFLOPS-rates for the forward elimination/backward substitution/whole algorithm:

	$n = 1000$	$n = 2000$
Tiling	46.3/2.8/45.2	50.8/2.6/50.2
TCGMSG	57.1/0.3/45.7	77.8/0.3/66.3

Then the speedup factors are:

Tiling	12.9/0.45/12.6	makes no sense!
TCGMSG	15.9/0.05/12.7	makes no sense!

It is nearly impossible to get exact speedup factors, because the user never knows how much of the local memory is available to him. Thus $f_{communication}$ can not be determined.
To get an acceptable performance for the parallel gauss algorithm above all one has to perform blocking to get cache reuse.

4 TIMESHARING ON PROCESSOR SETS AND CONCLUDING REMARKS

We have run the parallel matrix multiplication and the Gauss algorithm (both parallelized by tiling) concurrently on a processor set with 16 processors. We have chosen the sizes of the matrices so that all needed parts of the matrices fit into the corresponding local memories. First one can say that $f_{utilization}$ is always less than one, because the KSR-1 has a synchronous memory concept, i.e. threads waiting for subpages from a remote memory or from the disk are idling. If we only run two jobs concurrently, the performance rates of the two programs drop by factors up to 40. If we run three or four jobs concurrently, the factors increase further.

This means that the more programs run concurrently, the more decreases $f_{utilization}$. Thus $f_{utilization} << 1$ holds, if the processor sets operate in timesharing modus. Obviously KSR has not yet solved the problem of multiprogramming on a processor set with the present release OS_Release_R1.1.3 of the operating system. What are the conclusions of the measurements?

- To get acceptable performance rates the algorithms have to be redesigned with regard to cache reuse. This is valid for all parallel computers with superscalar processors and caches.
- It is very difficult to compute exact speedup factors on a parallel computer with a virtual (shared) memory like the KSR-1 because the performance depends heavily on the environment.
- Only space sharing should be allowed on processor sets that are used as production pools.
- The Allcache concept needs detailed compiler directives to achieve a high performance.
- TCGMSG: asynchronous communication should be available (better would be a migrated version of PVM3.x).

REFERENCES

1. W. Schönauer, H. Häfner, Performance estimates for supercomputers: The responsibilities of the manufacturer and of the user, Parallel Computing 17 (1991), pp. 1131–1149.
2. W. Schönauer, H. Häfner, Explaining the gap between theoretical peak performance and real performance for supercomputer architectures, in H. Küsters, E. Stein, W. Werner (Eds.), Proc. of the Joint international Conference on Mathematical Methods and Supercomputing in Nuclear Applications, M & C + SNA '93, Kernforschungszentrum Karlsruhe, 1993, Vol. 2, pp. 75–88.
3. W. Schönauer, H. Häfner, Supercomputer Architectures and their Bottlenecks, Proc. of ParCo 93, Grenoble, in press.
4. H. Weberpals, Struktur und Leistung paralleler Algorithmen für den Parallelrechner KSR-1, PARS Nr. 11, Juni '93, pp. 88–93.
5. H. Weberpals, Analysis of Parallel Algoritms for a Shared Virtual Memory Computer, Proc. of ParCo 93, Grenoble, in press.

Simulation of amorphous semiconductors on parallel computer systems

Bernd Lamberts and Michael Schreiber
Institut für Physikalische Chemie, Johannes-Gutenberg-Universität
D-55099 Mainz, F.R. Germany

1 Introduction

Our research project comprises essentially two parts. On the one hand we are interested in examining the physical properties of amorphous semiconductors with models which are as realistic as (numerically) possible. On the other hand we want to test the capabilities of parallel computers in such simulations.

2 Physical aspects

The foundations of the first, the physical aspect of this project are laid by our own experience with molecular dynamics simulations of amorphous semiconductors [1] . There we found that it is possible to describe the static properties of amorphous systems with satisfactory results using empirical many-body potentials [2]. However, that the results were so convincing may well be due to the fact that the structural properties depend essentially on geometric aspects which are well reproduced by the empirical potentials. But trying to use the same approximation for the analysis of dynamic variables did not give comparably good results. This indicates that the used empirical potentials offer an insufficient description of the dynamics on the atomic time scale. Therefore we propose a parameter-free method in order to describe such systems on an atomic time scale. Here the ab initio determination of the potentials and their derivatives is most important, because they yield the acting forces and thus the dynamics. The best way to perform such a calculation would be a complete quantum mechanical treatment. This is up to date not possible because of the incredible amount of numerical work which would be necessary to solve the dynamics of such many-particle quantum mechanical systems.

In our ansatz we try to restrict the quantum mechanical calculations to those parts of the simulation, which would yield questionable results with empirical methods. This means in particular the particle-particle interaction, as mentioned before. But the amount of computer time, which is needed to solve the quantum mechanical problem for every time step, is huge. Therefore we map the quantum mechanical interaction onto a parametrized many-body function, which is consecutively used to calculate the dynamics of our system by means of a classical molecular dynamics approach. However it is difficult to imagine that a single parametrized many-body potential is sufficient to represent the particle interaction in the entire configuration space in a satisfying way. It is more likely that the parameters are a function of the relative position of the particles, too. To obtain at each time step of the simulation a good and if possible, even improving description of the real system using the parametrized interaction function, we optimize our parameterization continuously during the simulation.

This optimization is performed after a certain number of molecular-dynamics simulation steps; the actual number depends on the simulation parameters. The refinement is done using the results of an ab-initio local-density-functional calculation. In this ab-initio calculation we take the last molecular-dynamics configuration as input data to calculate the quantum-mechanical results for the interaction energy and the one-particle energy. With the latter values the parameterization of our molecular-dynamics many-particle potential is improved and then the molecular-dynamics simulation is continued until the next refinement is due. Thus we get finally a non-empirical description of the interaction potential for the interesting part of the configuration space.
This offers us the possibility to investigate the dynamics of our system with the help of a classical molecular dynamics simulation, but without referring to empirical interaction potentials. So we can use the partial derivatives of the interaction to find the forces acting on the different particles. Thus the calculation of the forces is done in the same way as in other supposedly purely quantum mechanical methods. Here the Hellman-Feynman theorem can be used to justify this procedure.
Concerning the local-density-functional method for the calculation of the quantum mechanical results we refer to the literature [3],[4],[5] which shows that this method is the state of the art to calculate ab initio results in solid state physics, in particular in semiconductor physics, especially with respect to the calculation of the total energy.

3 Aspects related to the computer system

The aspects concerning the computational part of our project are mainly focused on the development of simulation programs for parallel computer systems. The resulting conclusions from this work are of course valid only for the presented topics, and should not be generalized to the treatment of arbitrary problems with parallel computer networks. The main numerical questions which are addressed in our project are:

- Do we get a significant performance gain of our programs?
- How big is the additional time for the parallelization of the programs?
- Is the ratio between the performance gain and the additional time for the development of the software reasonable?
- Which computer networks are most suitable for our problems?
- Is it meaningful to develop specially adapted programs for particular hardware platforms?
- Is it possible to develop transferable programs for different hardware environments?

We think that the presented project is quite suitable to investigate the above questions because it combines a number of very different numerical tasks in one program. Thus the communicational and the number-crunching demands vary in a wide range, which offers the possibility to examine the different kinds of computer networks in detail. Our experience shows that in our project there are a couple of problems well suited for parallelisation, which give good performance results on every hardware. Thus these programs yield information about the speed of the CPU only but not about the performance of the network. Other algorithms for example make special demands on the amount of transferable data per connection. Thus to judge a parallel computer environment our project with its wide scope of different algorithms can be helpful.

In the following list we compile the different numerical aspects which we cover with our programs. Afterwards we will discuss the parallelisational aspects of these points. We have to perform:

- setting up different matrices with varying expense for the matrix elements
- numerical integration in up to six dimensions
- linear and non-linear fits with up to twenty parameters
- diagonalisation of matrices with different algorithms
- orthogonalisation of matrices

The first point deals with the calculation of secular matrices of different Hamiltonians. Here we can distinguish between the local-density-functional and the molecular-dynamics programs. Concerning parallel computation both program parts are not problematical because the matrix elements are not depending on each other. Therefore we can calculate them independently, i.e. simultaneously. Thus the communicational tasks are restricted to the initialization of the network and the collection of the results. It is only necessary that the number of matrix elements is much bigger than the number of processors.

The second point refers mainly to the numerical calculation of the exchange-correlation energy in the local-density-functional program. This computation is normally uncritical, too, because the number of independent data points for the integral is usually much higher than the number of processors. Thus we can distribute their computation onto the different processors. But depending on the actual integrand and on the precision needed we can find a threshold value for the efficiency, if the network is significantly slower than the CPU. This holds true, for example, with respect to Ethernet-based PVM workstation networks.

The third point is connected with the parameterization of our many-body potential function, which we use for the molecular dynamics. Here we find a very coarse parallelisation easily possible which might be improved on a finer level. The first step means the minimization beginning from different starting points and continuously comparing the results. This is a good example for adapting an algorithm to the use of parallel computers, because only with this type of computer the presented strategy is really more powerful than others. Here it is extremely important how many operations one needs for a single minimization step, or in other words how much work has to be done for one evaluation of the cost function. In our project the cost function is expensive to calculate so that we had no difficulties with the straightforward parallelisation.

The next point is again connected with the Hamiltonian matrices of the local-density-functional and the molecular-dynamics programs. This matrix diagonalization represents a typical task of linear algebra. In a usual iterative algorithm it consists mainly of matrix vector multiplications. Here the communication speed is very important for overall performance, because the number of communication operations is of the order of the number of calculation operations. Thus this type of problems is particulary hard to solve efficiently on parallel computers.

The last point (the orthogonalisation) appears in the molecular-dynamics program. To keep the iterative diagonalisation algorithm stable we have to reorthogonalize our eigenvector matrix at each time step. The problem is comparable with the previous point, but with the additional complication that the intermediate results are not independent. So we have to broadcast them to the network during the calculation. This is why that kind of problems is very sensitive not only to the communication speed, but also to the latency time of the network. The latter point is connected with the fact that in this case very small packages have to be transferred.

4 Special experience with the KSR1-32 in Mannheim

First we describe the porting of our programs onto the KSR. The programs have been developed on personal computers, on Texas Instruments 320C40 signal-processors, and on Inmos T800 Transputer-based networks. The last two systems are message passing systems. Because of the lack of libraries at the beginning of our program development, we decided to create our own environment for the package containing communication libraries as well as numeric routines. This allowed us to design the communication routines in a rather transparent way. Restricting the hardware-dependent routines to some key functions we have thus created a highly portable package. Because of the latter point, we favoured a very simple network topology above a more complex but possibly slightly faster one. Therefore we used a simple pipeline topology.
The programs were written using the language C, as far as possible we followed the ANSI standard. Here we had to make concessions in connection with the communication routines because of their hardware dependent function calls.
Because the KSR1 is a Unix system following the OSF-1 standard, we supposed that we could port the serial version of our programs without much problems. This expectation was completely fulfilled. It was possible to do the implementation in a couple of days. This time would have been shorter without some hardware problems at the beginning. Here we have to acknowledge that we were indeed a kind of beta user, having access to the machine directly after the first installation. So this type of problems has to be viewed as typical for such complex computer systems. (The reader might just remember his last PC installation with Windows and some third-party packages.) Up to our knowledge the mentioned hardware problems have been solved in the meantime.
Our further experience shows that the implementation of the parallel versions of the programs is more difficult. As mentioned we had already some practice with portability on message-passing machines. Therefore it was a pleasant surprise for us that even this step could be done quite fast. It was possible to port the whole simulation package in approximately one week during which we replaced the message passing routines by POSIX threads and the Simple Presto library (SPC package). We can strongly recommend this procedure, because it offers more transparent programs and better performance capabilities than message-passing-based programs on the KSR1. Furthermore it allows us to use standard software interfaces like POSIX which might be important for future applications on other Unix systems. In this context we would like to mention our current efforts to replace all SPC library calls in our package in order to limit ourselves to Posix standard functions. Thus it would be possible to run our program on all machines belonging to the OSF-1 standard without changing the source code.
But as usual, the straightforward porting of complete programs gives only poor performance compared to the theoretically possible speed. As we have seen this holds true for both cases, the sequential and the parallel performance. Because of the machine architecture, which shall not be discussed here, the performance aspects of sequential programs can be handled as if one is programming a modern RISC machine with a large cache system. Unfortunately we do not have available exact numbers about this performance, but as a rough rule of thumb we got a performance gain between 50% and 100% by adapting our code to the machine. Here we have to stress that this estimate relates to the whole program and not only to one optimized routine, where one can get much bigger increases. The above statements are valid in principle for the parallel case, too. But here the difference between optimized and unoptimized programs is more drastic, yielding an additional factor between 3 and 5. This is due to the fact that the system overhead for the communication can be decreased enormously. To get some feeling for this it is helpful to check the program with the Unix timer-function call. The user time is almost stable during the optimization but one can decrease the system time drastically, because the calls to

system functions which move data from one processor to another are contributing to the latter time. Unfortunately the performance monitoring tool was not yet completely installed when we developed and tested our package, so we are not able to give exact numbers here. But when it was available we have seen that if one really wants to optimize a parallel program on the KSR1 this tool is very useful if not necessary. Here we would like to mention again that one of the most important advantages for programing on the KSR1 is the fact that one works within real Unix environment with all the benefits known from sequential Unix machines.
To give some feeling for the scaling of parallel programs on this machine we present in the following table benchmarks of the molecular-dynamics routine on different numbers of processors.

Table 1: Comparison of the run time of the MD-routines for the KSR1-32

number of processors	time [arb.units]	parallelisation [%]
1	100.0	-
2	55.5	90
4	29.4	86
6	20.3	83
8	17.3	78

Here we have to mention that these values do not relate to the whole program, they just represent the timings we measured for that routine of our package which is most critical concerning parallelism.
The second table contains some benchmarks for two sequential programs which include typical numerical tasks from our project.

Table 2: Comparison of the run time for the program MDSI on different computer systems, for different numbers of atoms in the supercell and different numbers of MD steps.

		8 atoms ; 250 steps		64 atoms ; 50 steps	
Computer	Compiler	[min:sec]	factor	[min:sec]	factor
486Dx 25MHz	Turbo-C 2.0	0:28	27.0	-	-
386Dx/87 33MHz	Turbo-C 2.0	1:42	100.0	20:35	100.0
T800 20MHz	Inmos C V2.0	0:44	43.1	8:11	39.7
C40/50	TIC (o=0)	0:03	2.9	0:36	2.9
KSR1–1 proc	cc -O2 -qdiv	0.03	2.9	0:44	4.5
HP720	cc +O2 V9	-	-	0:26.4	0.021
IBM 6000/530	cc -O	-	-	0:35.2	0.028
IBM 6000/580	cc -O	-	-	0:18.6	0.015
DEC alpha (130)	cc -O2	-	-	0:20.0	0.016
Super Sparc	gcc -O2	-	-	0:36.5	0.030

The program mdsi contains a molecular dynamics routine with three-body interaction. The applied potential is mainly expressed by exponential and sine functions multiplied with some polynomial. In the program oversij an interaction matrix with nearest-neighbor exponential potential is calculated first. Afterwards a diagonalisation of this matrix is performed using the

Table 3: Comparison of the run time for the program OVERSIJ on different computer systems, for different numbers of atoms in the supercell.

		8 atoms		64 atoms	
Computer	Compiler	[min:sec]	factor	[min:sec]	factor
486Dx/87 25MHz	Turbo-C 2.0	0:15	50.1	-	-
386Dx/87 33MHz	Turbo-C 2.0	0:30	100.0	-	-
T800 20MHz	Inmos C V2.0	0:11	36.6	-	-
C40 50MHz	TIC(o=0)	0:01.7	5.6	-	-
C40 50MHz	TIC(o=2)	0:01.0	3.4	-	-
IBM 6000/530	cc -O	0:00.42	0.028	6:12.22	36.8
IBM 6000/580	cc -O	0:00.22	0.015	3:27.31	20.5
HP720	cc +O2 V9	0:00.40	0.027	6:28.16	38.4
KSR1–1 proc.	cc -O -qdiv	0:00.9	0.06	16:50.00	100.0
DEC alpha (130)	cc -O2	-	-	4:11.00	24.8
Super Sparc	gcc -O2	0:00.60	0.04	7:26.00	44.1

Householder algorithm.
As it can be seen from the table, the performance values for a single KSR1 processor are not so good if one compares them with those of an up-to-date workstation. This is especially true, if one takes into account that the KSR1 uses a 64bit RISC CPU. This is consistent with our statement that one has to optimize the programs to get good sequential performance on this machine. But the first table demonstrates the good balance between the communication speed and the sequential CPU performance. We believe that this is the most interesting feature, and we observed that this is appropriate. To increase only one of these performances would not be very meaningful for a general purpose parallel computer like the KSR1.
To conclude, we found that working with the KSR1-32 did not solve all our computer problems automatically, indeed it offered some new problems, but it offered also speed, a good hardware platform, and a very good programming environment. This last point cannot be overrated from the programmer's point of view.

References

[1] B. Lamberts, Diplomarbeit, Universität Dortmund (1989).

[2] A. Stillinger and W. Weber, Phys. Rev. B **31**, 5262 (1985).

[3] W. Kohn and L.J. Sham, Phys. Rev. **140**, A1133 (1965).

[4] P. Hohenberg and W. Kohn, Phys. Rev. **136**, B864 (1964).

[5] M.T. Yin and M.L. Cohen, Phys. Rev. B **26**, 5668 (1982).

An Explicit CFD Code on the KSR

Michael Fey and Hans Forrer
Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule Zürich
CH–8092 Zürich
Switzerland

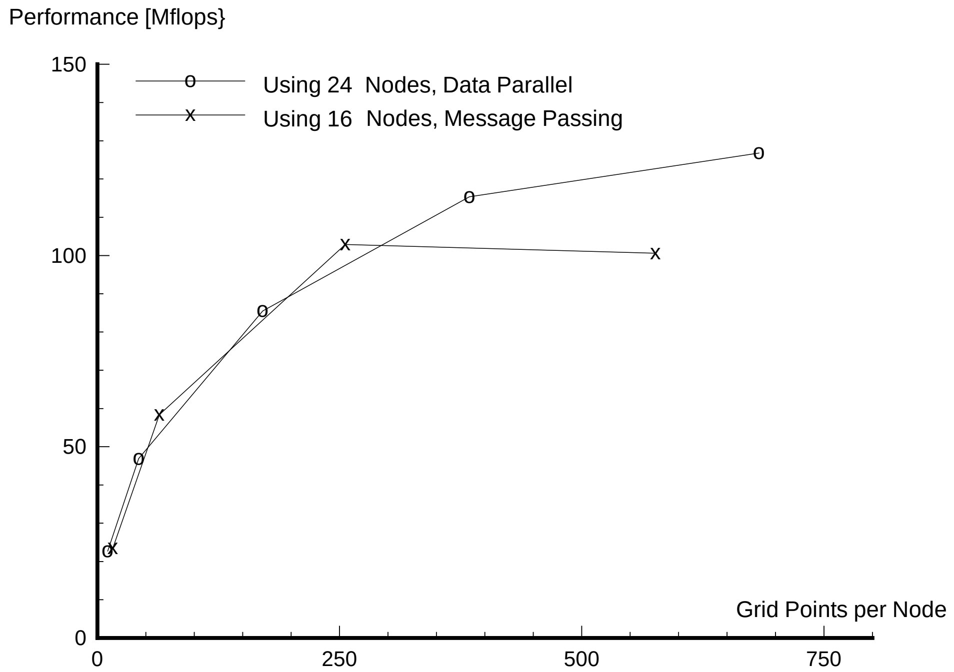
In this work we used parallel computers to integrate the Euler equations numerically by explicit schemes. We solved a 2-D Riemann problem and examined performance, speedup and ease of implementation. As each grid point needs information only from the neighboring points to update the state vector, data has good locality in explicit schemes, which makes it easy to attain load balancing.
In addition working with finer grids, the number of points per processor is proportional to $1/\Delta x^2$ (Δx is the spatial increment of the 2-D grid) but the amount of data to be exchanged is proportional to $1/\Delta x$. So for finer grids or bigger problems the ratio computation to communication gets better.
To take advantage of the parallel resources of the KSR we worked in the data parallel mode provided by the Presto - routines, as well as with the message passing TCGMSG - library.
The scheme, developed by M. Fey [1][2], is a truly multidimensional finite volume method, as the information travels along characteristic directions. So the method is nearly independent of the underlying grid and gives good results for complex shock interactions and hypersonic flows. The scheme allows infinitely many directions of propagation, which can be reduced to a finite number for reasons of efficiency and simplicity. In this implementation we used a regular Cartesian grid, so each cell uses flow values of the eight nearest neighbors.
The KSR is a machine with shared memory. Parallel support for Fortran programs is given by various constructs. We used the tile families. They allow the simultaneous execution of some iteration space. The user can specify the distribution of the iteration space over the available processors. Because the same iteration will appear several times, by this construct the performance is increased by minimizing data movement. Efficiency is further increased by coordinating the tiling decision made for groups of tile families that reference the same data, using the construct of affinity regions [3].
In order that some temporal variable has a copy on every processor it must be declared as private. However private variables must not be arrays. By this restriction we had to rename some temporal arrays to scalars. In the meantime we learned that this work could have been circumvented by using private common blocks.
By the compiler directive -kap the parallel constructs are inserted automatically into the serial Fortran code. But there is no guaranty that the choice made by the compiler is optimal. In our program the (i,j)-iteration space of a 2-D grid is tiled correctly, but also the local dimension of the state vector was split up, so it happened that the (ρ, u, v, e)-components of some (i,j)-state vector were on different processors. But one can use KAP as a start to parallelize a program.
Because on other parallel computers there are message passing routines to parallelize programs, to port these codes on the KSR we were interested in such constructs on the KSR. We used the TCGMSG library, developed by Robert J. Harrison [4], which is optimized on the KSR,

and provides all common message passing commands.
For the cited Mflops rates, we have run the program on a Y-MP to count the floating point operations. On the KSR the single node peak performance of the Custom VLSI processor is 40 Mflops at clock rate of 20 MHz.
We compared this peak performance with the single node performance we got with our Fortran code for two different grid sizes.

Single Node Performance (Percentage of Peak) for our Fortran Code:

Grid Size	16x16 grid	128x128 grid
Mflops	15.95 (40 %)	10.31 (26 %)

Because of the allcache memory of the KSR, for the small grid size where the whole data fits into the 0.25 MB subcache we got 40 % of the peak. With increasing problem size the performance decreases, but is still 26 % of the peak performance for the 128x128 grid.
The following figure shows the performance varying the grid size.



Performance for variable Grid Sizes

It shows that the TCGMSG message passing library on the KSR gives good results. (The performance on the KSR for the SIMD mode using the *presto* - library can be improved using prefetch and poststore.)

References

[1] Michael Fey and Rolf Jeltsch. A new multidimensional Euler scheme. In A. Donato, editor, *Proceedings of the 4th International Conference on Hyperbolic Problems, Taormina 1992*. Vieweg Verlag. to appear.

[2] Michael Fey and Rolf Jeltsch. A simple multidimensional Euler scheme. In Ch. Hirsch, editor, *Proceedings of the First European Computational Fluid Dynamics Conference, Brussels, 7-11 September 1992*. Elsevier Science Publishers, 1992.

[3] *KSR Parallel Programming*. KSR Corporation, 1991.

[4] R. J. Harrison. *TCGMSG Toolset*. via anonymous ftp: ftp.tcg.anl.gov.

Performance Studies on the KSR1

Jean–Daniel Pouget and Helmar Burkhart
Informatics Department, University of Basel, CH-4056 Basel
Switzerland

1 Prologue

Sponsored by the Priority Programme Informatics of the Swiss National Science Foundation, researchers at the Basel Parallel Processing Laboratory attack some of the major problems in the field of massively parallel processing. Because our research wants to improve the programmer's situation when using modern parallel computers, hands-on experience with such systems is mandatory. This activity report summarizes some experiments that we have made on the KSR machine of the University of Mannheim in fall 1993. We describe the experiments and present the performance figures measured. Finally, we sketch our future research activities with the KSR1.

2 First experiments

In order to get a first feeling for the performance of the KSR machine, we have done comparisons with conventional computers heavily used in our local environment: VAX 7000, VAX 6510, NeXT, and DECstation 5000. All measurements have been done in a production environment; especially all multi-user systems, such as the KSR machine, reflect the net performance a typical user gets. Three simple test programs have been used:

- Optimal, a synthetic program that aims towards high MFlop/s rates. The core of this program is an unrolled loop with loop body computations on register float variables: rb = ra * rb; rc = rc + rd. These two computations are executed several thousand times (in a for loop) with an unrollment factor of 10.
- MatMult (100) and MatMult (500) are matrix multiplication routines for square matrices of size 100 and 500, respectively.

Fig. 1 shows the performance measured. We also did experiments with higher unrollment factors. For a factor of 100, we measured an increase of ($\approx$ 10%) on KSR1, VAX 7000, and VAX 6510, while for the NeXT workstation,the performance decreased since the loop body was too large for the instruction cache.
Program performance is not only dependent on the hardware architecture, but also on the programming language and compiler, respectively. Facts well known in informatics, but from time to time you have to tell these things to application programmers !
Such effects were explored for the matrix multiplication program on the KSR1; we found performance factors of $10 - 30$, depending on which loop nesting strategy (i.e. memory access patterns), which language/compiler (C or Fortran), and which blocking mode is used. Fig. 2 shows some of the results. The sequential matrix-multiplication (matrix size 512) is programmed using the access patterns ((i,j,k), (i,k,j), and (j,k,i)). c stands for C language, f for Fortran.

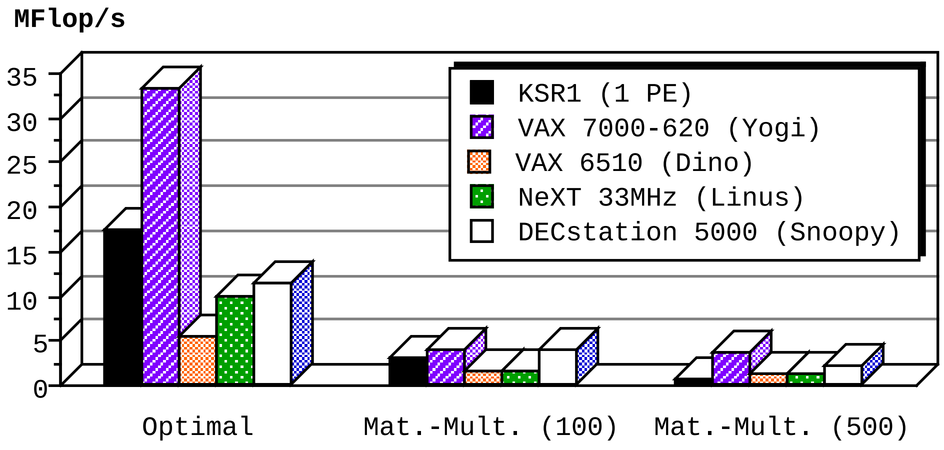


Figure 1: MFlop rates measured

Finally cb and fb stand for an additional blocking of data (a blocking factor of 16 was used because on the KSR1 with 8 Bytes per REAL a matrix row/column fits exactly into one 128 Bytes subpage). Beside Mflop rates, also data subchache misses (dsc_miss) and subpage-Misses (sp_miss) are shown.

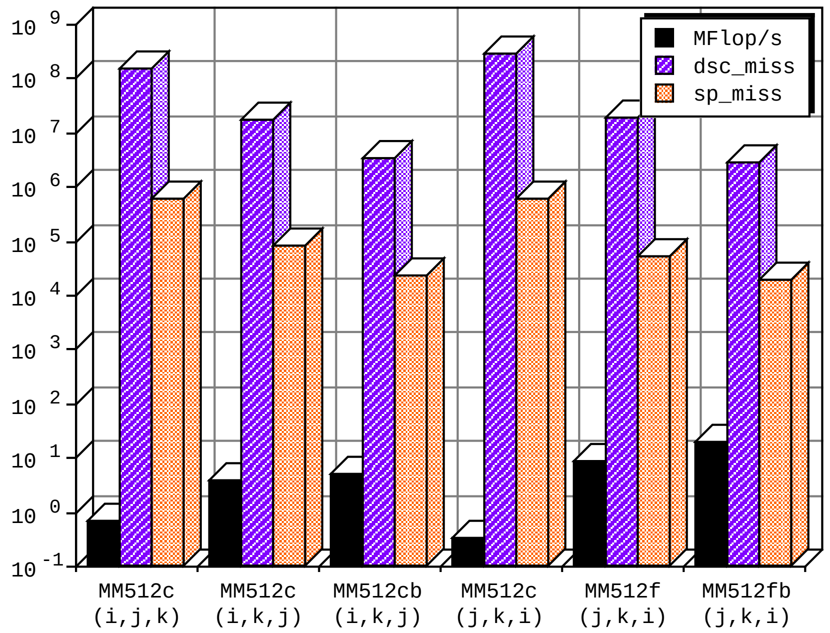


Figure 2: Different matrix-multiplication programs on the KSR

The straight-forward (i,j,k)-program has the disadvantage of using both row and column access patterns. Because C accesses by row mode, but Fortran accesses by column mode, this version (with a performance of ≈ 0.7 MFlop/s) is inefficient for both languages. For C the (i,k,j)-program is therefore much better (≈ 4 MFlop/s). With blocking of data, the performance is again better (≈ 5 MFlop/s). The decreasing cache miss figures explain all these performance gains.

For the (j,k,i)-variant, when programmed in Fortran, we expected results similar to the (i,k,j)-C-variant. However, we got a performance that was doubled (≈ 8.5 MFlop/s). Analysis of the generated assembly code revealed that the Fortran compiler used "Multiply-And-Add" instructions, while the C compiler generated two separate instructions. The best performance we got was for the blocking Fortran variant (≈ 20 MFlop/s).

We now felt ready to start with the analysis of parallelized versions. For simplicity we use the (i,j,k)-variant here. Fig. 3 and 4 show the execution times and speedup values for C. For the data distributions we used a cyclic distribution by rows (other experiments have shown that block distributions are however more efficient).

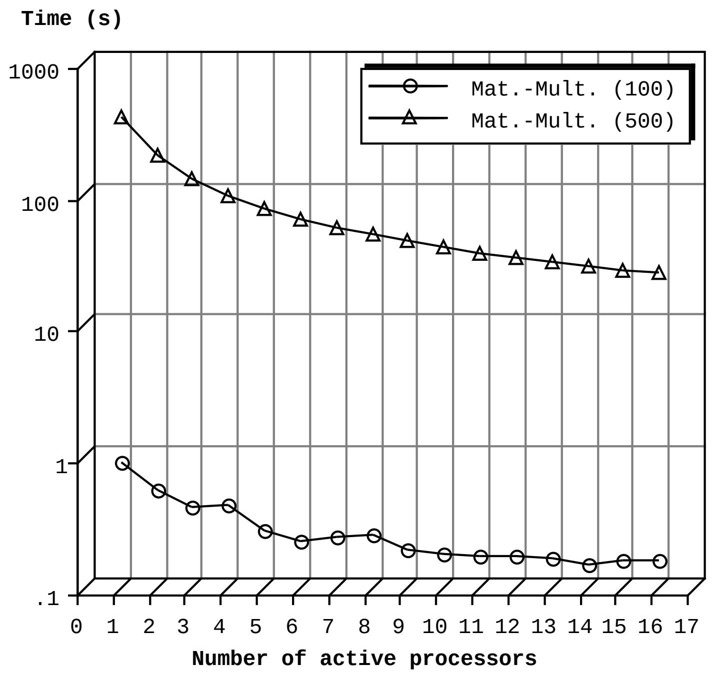


Figure 3: Execution times for the parallelized matrix multiplication

You can see a linear speedup for size 500, but a rather flat curve for size 100. If you compare the MFlop/s rates shown in fig. 5 however, you find a worse processor performance for size 500 compared to size 100. The main reason is the high number of cache misses because of improper access patterns and data distributions. Thus, the parallelized (i,k,j)-variant using blockwise data distribution results in a much better performance (55 MFlop/s on 16 processors).

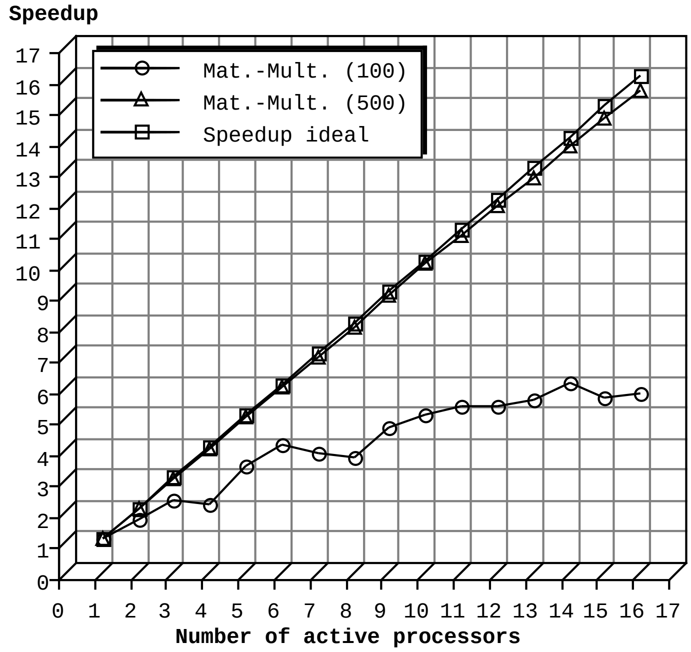


Figure 4: Speedup values for the parallelized matrix multiplication

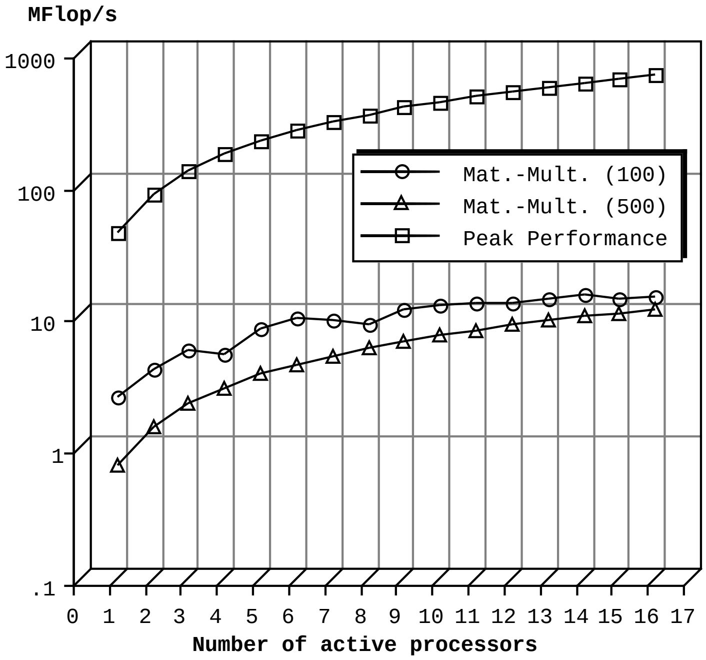


Figure 5: Comparison of MFlop/s rates for parallelized matrix–multiplication

3 How expensive are synchronization constructs ?

Process management and process synchronization are the two basic tasks an operating system has to provide operations for. We have measured the performance of synchronization operations on the KSR machine (see also [1]).

Lock/Unlock operations are necessary to build critical regions, which only one process can enter. The test program generates a number of slave-threads, which execute the lock/unlock operation pair 100000 times. The time needed for one pair is shown in fig. 6.

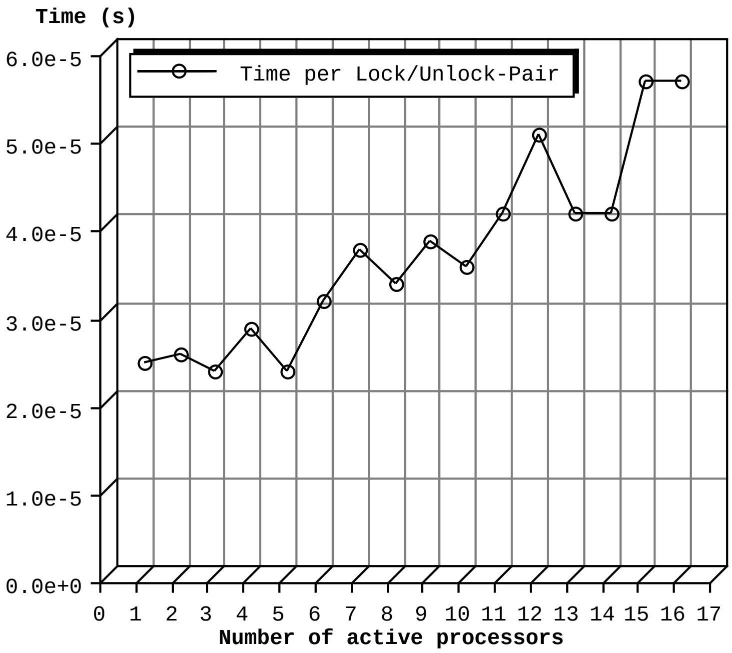


Figure 6: Execution times for lock/unlock

As the figures reveal, the execution time increases with the number of processors used. This is because the common lock variable migrates from one processor cache to another more often when processors are added. It has been interesting to see that the absolute times of 20-60 microseconds are in the same order as for an 8Mhz Motorola 68000-based system, we built in the beginning of the eighties [2] (96 microseconds using the TAS instruction). The increase of processor performance has not resulted in more efficient programming constructs. Clearly a deficiency that needs to be attacked.

Barrier is another popular synchronization construct that we have analyzed. Our test program executes the barrier-operation a thousand times. Fig. 7 shows the time measured per operation.

Again, the time needed linearly increases. The absolute times are in the range of 100 to 200 microseconds.

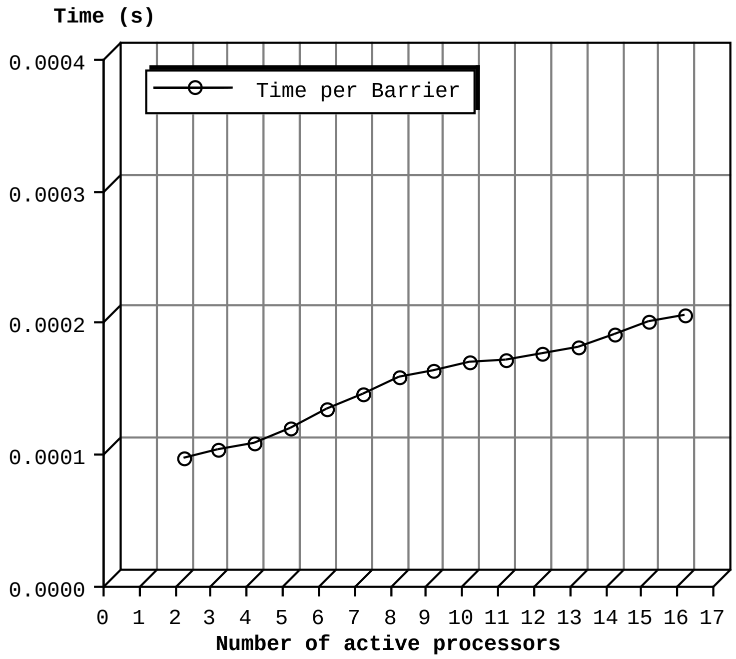


Figure 7: Execution times for the barrier-operation

4 KSR Research Projects in the Near Future

The Basel Algorithm Classification Scheme BACS defines a framework and terminology to attack the software dilemma on today's parallel systems [3]. Our driving force is program portability and algorithmic reusability. We cannot expect that complete application programs are reusable, because – even in the same problem domain – there are always slight differences, e.g. different input/output formats. We can, though, expect to get reusable components for the coordination part of a parallel program, because real-world applications are quite regular regarding the process structure. Our current research has shown that only a small number of coordination schemes are used in algorithms and applications. On massively parallel systems this trend is emphasized because hundreds or thousands of different processes cannot possibly be managed individually. The percentage of code for the coordination part is definitely much smaller compared to the computation part; but as already mentioned, it is the crucial part. It is, therefore, this portion of code that we will make reusable by providing a library of algorithmic skeletons. The approach we have in mind is called skeleton-oriented programming. Both the skeleton generator TINA [4] and the skeleton- based ParStone benchmark suite [5] will be made available for the KSR machine.

Acknowledgements

We would like to thank the University of Mannheim for providing an account on the KSR machine. R. Schumacher and H. Kredel have helped us to solve initial problems. G. Shah, Georgia Institute of Technology, gave comments regarding the synchronization costs.

References

[1] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability Study of the KSR-1. Technical Report GIT-CC 93/03, College of Computing, Georgia Institute of Technology, Atlanta (USA).

[2] Helmar Burkhart, Rudi Eigenmann, Heinz Kindlimann, Michael Moser, and Heinz Scholian. The M^3 Multiprocessor Laboratory. IEEE Trans. on Parallel and Distributed Systems. Vol. 4, No. 5, 507-519, May 1993.

[3] Helmar Burkhart, Carlos Falco Korn, Stephan Gutzwiller, Peter Ohnacker, and Stephan Waser. BACS: Basel Algorithm Classification Scheme, Technical Report 93-3, Institut für Informatik, University of Basel, Switzerland, March 1993.

[4] Stephan Gutzwiller. Methoden und Werkzeuge des skelettorientierten Programmierens, PhD thesis, University of Basel, to appear 1994 (in german).

[5] Stephan Waser. Benchmarking Parallel Computers, PhD thesis, University of Basel, July 1993.

Implementation of the PARMACS Message–Passing Library on the KSR1 system

Udo Keller, Karl Solchenbach

PALLAS GmbH, D–50321 Brühl

1 Background

The KSR1 machine is typically used with its standard shared–memory parallel programming model characterized by tiling, parallel regions, the KAP tool etc. This does, however, not exclude to execute message–passing based parallel codes on the KSR system, if a the message–passing library has been implemented. In this short paper we outline the implementation strategy for the PARMACS message–passing library and give some performance results which we obtained on the KSR1 in Mannheim.
PARMACS [2] is a widely–used message–passing library and guarantees portability of application codes (in Fortran or C) across (nearly) all parallel architectures and platforms. PARMACS are available on

- distributed memory systems: Convex Meta, Cray T3D, Fujitsu VPP500, Intel iPSC 860 and Paragon, Meiko CS–1/2, nCUBE/2, Parsytec GCel, Thinking Machines CM–5, Transtech Paramid
- shared–memory systems: Convex C2/C3, Cray Y–MP/C90, KSR1
- workstation–clusters: DEC, HP, IBM, SGI, SUN and heterogeneous combinations

PARMACS implementations are available from PALLAS (contact: info@pallas–gmbh.de) and from some hardware–vendors.

2 Implementation

On behalf of KSR PALLAS implemented the PARMACS message–passing library on the KSR1 system. Since KSR wanted to benchmark PARMACS–based application codes, the main goal of the implementation was to achieve best possible performance. In particular, the performance of PARMACS should be superior compared to other message–passing systems on the KSR1 (as PVM and TCGMSG).
PALLAS decided to use a low–level and hardware–specific implementation strategy. PARMACS processes are mapped on *pthreads* (and not on heavy–weighted UNIX processes). This strategy resulted in a good communication performance and met the ambitious performance goals. It turned out to be also very memory–effective since – in contrast to other message–passing systems – only one image of the code has to be stored in memory. Lowest–level parts

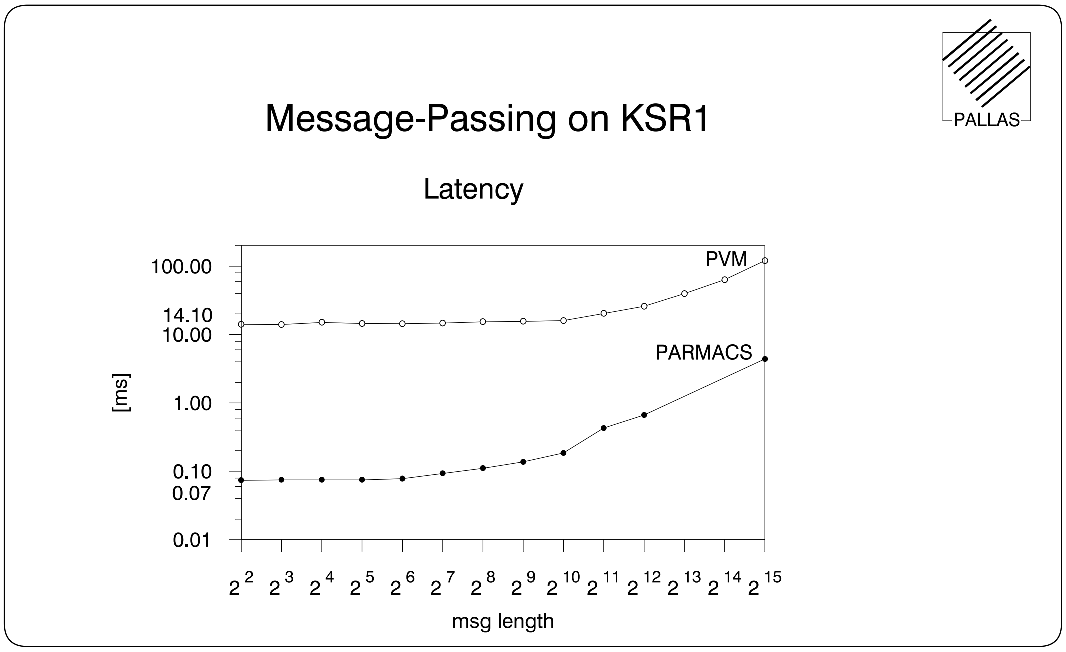


Figure 1: Ping–pong benchmark on KSR1, short messages

of the implementation have been adapted at machine–instruction level, in order to utilize processor–specific features not obtainable from existing KSR compilers.

3 Performance

The time needed to send a message of l 64–bit words from one application process to another application process is assumed to be

$$t = \alpha + \beta l.$$

α is the so–called *start–up time* or *latency*, β is the transfer time per data word. The asymptotic communication bandwidth (for infinitely long messages) is

$$b_\infty = \lim_{l \to \infty} \frac{l}{t} = \frac{1}{\beta}.$$

On many parallel systems this model describes the behaviour of a message–passing parallel computer for the *ping–pong* benchmark [1] quite exactly.
Figure 1 shows the times of the ping–pong benchmark on the KSR1 with PARMACS for short messages, Figure 2 shows the bandwidth for long messages. The dotted lines are the figures measured on the KSR1 with the public domain PVM 3.2.2 system.
In analogy to Hockney's model of the vector processor performance the message–length for which half of the asymptotic bandwidth is achieved is defined as [1]:

$$l_{1/2} = \frac{\alpha}{\beta}$$

For messages shorter than $l_{1/2}$ the latency is dominant, longer messages are bandwidth–bound.

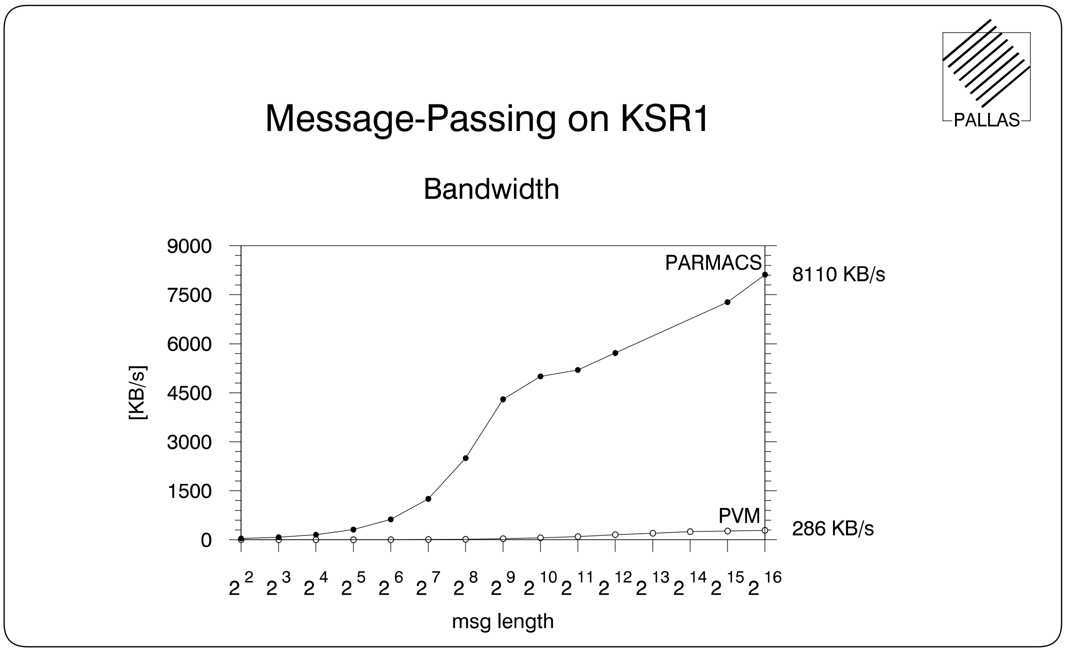


Figure 2: Ping–pong benchmark on KSR1, bandwidth for long messages

In order to quantify the balance of a parallel system, i.e. the relation between its communication performance and its single processor performance we compare the message–passing parameters α and β with the floating–point performance r of the KSR1 processor. As a realistic measure of the floating–point performance the LINPACK–100 [3] figures were used for r (instead of the unrealistic peak rate).
We define the normalized quantities

$$\overline{\alpha} = r\,\alpha, \qquad \overline{\beta} = r\,\beta, \qquad \overline{b_\infty} = \frac{b_\infty}{r}.$$

$\overline{\alpha}$ is the normalized latency time, it indicates how many floating–points operations (flop) can be executed in the same time as the start–up of one message. Similarly $\overline{\beta}$ floating–point operations can be executed during the transfer of one 64–bit data word. $\overline{b_\infty}$ is the asymptotic bandwidth measured in Mwords/s per Mflop/s.
Often the speed–up and the scalability of applications on message–passing systems is characterized by (only) these normalized parameters [4].
Table 1 shows the KSR1/PARMACS values of the communication parameters and gives comparitive numbers for other distributed–memory and shared–memory parallel computers (all with PARMACS).
Figure 3 shows the speed–up of a benchmark code PDE1 from the GENESIS benchmark suite which is part of the European software and benchmark initiative RAPS (Real Applications on Parallel Systems). PDE1 is based on PARMACS and solves a 3D Poisson equation by red–black relaxation, the classical model problem for all finite difference PDE solvers. For the measurements the grid size was chosen to be 64^3 grid–points (small problem) and 128^3 grid–points (large problem).
On a single processor 6.0 Mflop/s were achieved for the small problem and 2.9 Mflop/s for the large problem. The performance decrease is obviously due to subcache misses. The speed–up of

	raw data				normalized data			
Data	α	β	b_∞	r	$\overline{\alpha}$	$\overline{\beta}$	$\overline{b_\infty}$	$l_{1/2}$
Unit	μs	μs/W	MW/s	Mflop/s	flop	flop/W	W/flop	W
KSR1	75	1.0	1.0	15	1125	15	0.067	75
Cray Y–MP	30	0.02	50	161	4830	3.2	0.31	1500
CM–5	34	0.91	1.1	est. 10	340	9.1	0.11	37
iPSC/860	107	2.9	0.35	9.7	1038	27.7	0.036	37
nCUBE/2	220	4.7	0.21	0.78	172	3.7	0.27	47
Parsytec GCel	250	6.2	0.16	0.50	125	3.1	0.32	40

Table 1: Message–passing performance characteristics for KSR1 and other parallel computers under PARMACS. Note that data transfer is measured in 64–bit words (W)

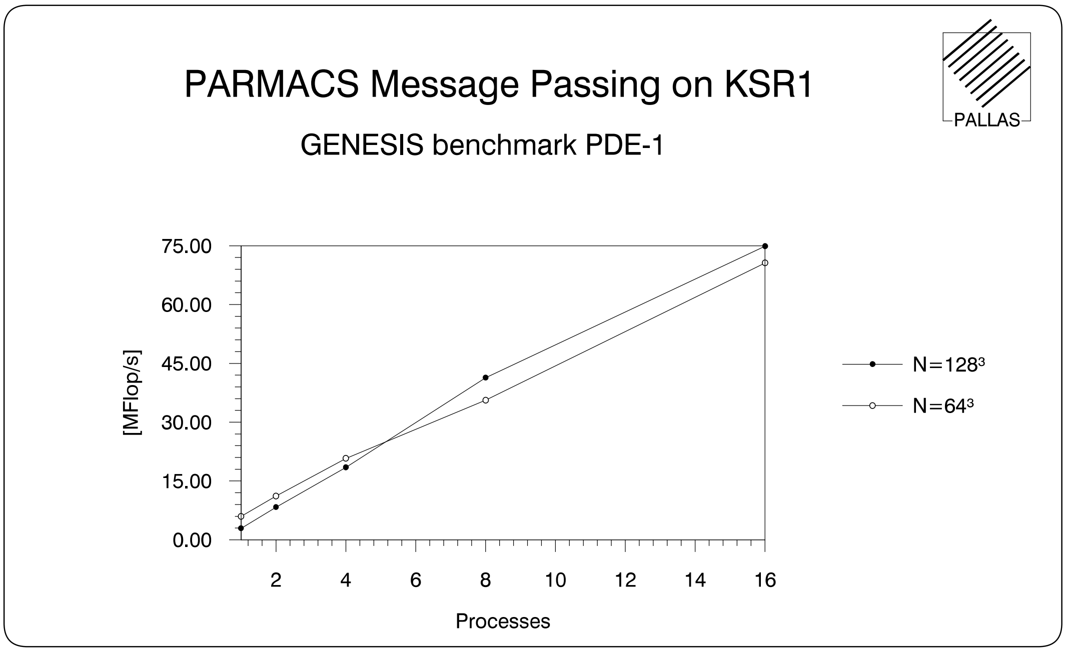


Figure 3: Speed–up of PDE1 benchmark on KSR1

11.8 on 16 nodes for the small problem is quite satisfactory. For the large problem we generate an artificial superlinear speed–up which again is due to the non–optimal sub–cache access on the single processor.

4 Conclusion

With the PARMACS library message–passing based codes can be executed on the KSR1 machine very efficiently. The communication performance compares favourably with other dedicated message–passing computers. For model benchmarks reasonable speed–ups have been measured.

All performance figures were measured on the KSR1 system in Mannheim when it was empty. We want to thank the other users for their patience during our tests. In particular, we thank Dr. Robert Schumacher for his support. We also appreciated the assistance from KSR staff in London and Munich which allowed us to identify several (very special) bugs in the KSR OS. KSR has fixed all these bugs meanwhile.

References

[1] **Addison, C., Getov, V., Hey, T., Hockney, R., Wolton, I.:** The GENESIS Distributed-Memory Benchmarks. Highly Parallel Computing Systems, 13-17 July 1992, IBM Europe Institute.

[2] **Hempel, R., Hoppe, H.-Ch., Keller, U., Krotz, W.:** PARMACS 6.0 Specification. PALLAS Report 92-7.

[3] **Dongarra, J.:** Performance of Various Computers Using Standard Linear Equations Software. CSD Univ. Tennessee, TN 37996-1301, July 22, 1993.

[4] **Solchenbach, K.:** Communication performance of message-passing systems. To be presented at RAPS Workshop, 7-8 December 1993, Southampton.

Ongoing Projects

Projects which are just started or from which we got no report so far include the following:

- Image Processing
  P. Frankenberg, M. Kappas, Geographical Faculty, University of Mannheim

- Linear Programming
  Ch. Schneeweiß, H.–J. Vaterroth, M. Hauth
  Faculty for Business Administration, University of Mannheim

- Genetic Algorithms
  R. Männer, Faculty for Computer Science, University of Mannheim

- Tax–Clientel–Effects on the German Bond Market
  S. Rasch, Center for European Economical Studies, Mannheim

- Reduce on the KSR1
  T. Bönninger, Computing Center of the University of Cologne
  H. Kredel, Computing Center of the University of Mannheim

- Simulation of Tumorgrowth
  H.–P. Altenburg, Department for Medicine, University of Heidelberg

- Implementation of PARSAC–2 on the KSR1
  W. Küchlin, Institut for Computer Science, University of Tübingen

- Simulation of the Pulsarmagnetosphere
  H. Ruder, T. Kioustelidis, Physics Department, University of Tübingen

- Optimization of Highfrequency–Pulses for NMR–Tomography
  F. Krämer, Department for Medicine, University of Freiburg

- Parallel Algorithms for Molecular Dynamics
  S. Reiling, H. Vollhardt, Chemistry Department, University of Darmstadt

- Turbomole on the KSR1
  R. Ahlrichs, Department for Theoretical Chemistry, University of Karlsruhe
  R. Völpel, GMD, Sankt Augustin

- Port of the ECMWF IFS (Integrated Forecasting System) Code to the KSR1
  U. Trottenberg, GMD, Sankt Augustin