COMPUTERALGEBRA ON A KSR1 PARALLEL COMPUTER

HEINZ KREDEL
RECHENZENTRUM UNIVERSITÄT MANNHEIM*

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 machine runs OSF1 Unix and has a virtual shared memory with 64 bit integer architecture. The port of Maple was unsuccessful until now, since the system strongly relies on 32 bit integers and pointers. 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 custom made processor with a 64 bit integer and 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 mechanisms 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.

The KSR1 machine runs under OSF1 Unix, provides most GNU utilities, X-windows and a C, C++ and a FORTRAN 77 compiler with KAP (semi-automatic parallelization preprocessor). Low-level concurrent programming is possible with the POSIX threads library based on the

* E-MAIL: KREDEL@RZ.UNI-MANNHEIM.DE

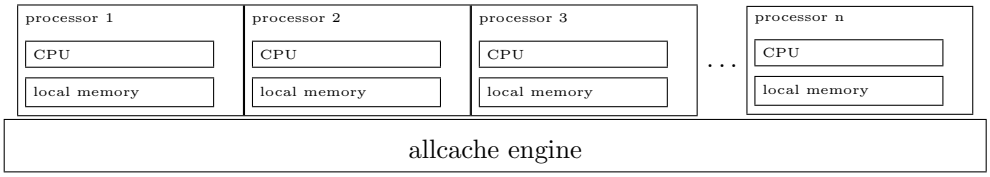


FIG. 1. *Hardware Architecture of KSR1*

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 next section we will

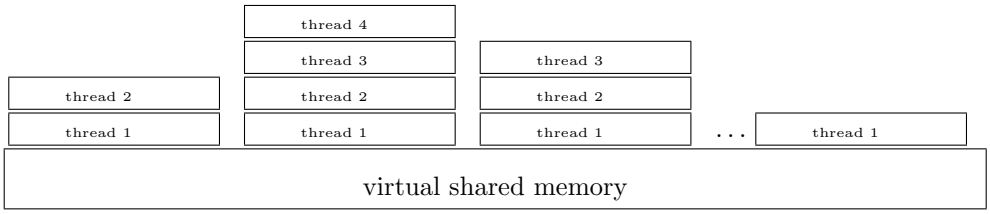


FIG. 2. *Software Model of KSR1*

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). During the mark step references into the cell space of other processors are ignored. 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

FIG. 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.

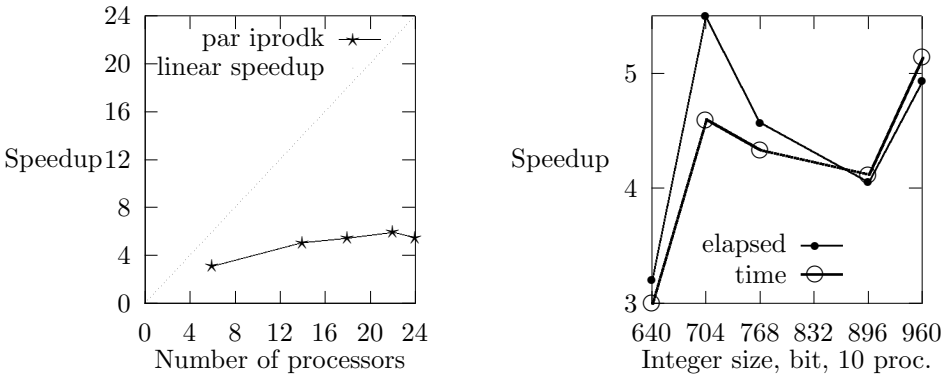


FIG. 4. *Integer Product*

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 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 11 and 12. 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.

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

```
PROCEDURE NF(P, S);
(*1*) (*initialization*) R:=0;
      m:= channel to master;
      i:= channel to next;
      create( pt, NFpipe, (m, i, HT(P)) );
      send( chan[i], -1, pt, P);
(*2*) (*send monomials to next *)
      FORALL monomials M in S DO
             send( chan[i], M ) END;
(*3*) (*receive irreducible monomials *)
      LOOP
           receive( chan[m], M ); R:=R+M;
           UNTIL finished;
(*4*) (*finish*)
      receive( ch[m], pts ); join(pts);
      RETURN( R );
```

FIG. 5. *Pipelined Normalform: NF*

```
PROCEDURE NFpipe(m,i,PH);
(*1*) (*initialization*) receive( chan[i], pt1, pt2, P );
(*2*) (*search reducible monomial *)
      LOOP receive( chan[i], M );
           IF M is irred wrt. PH
              THEN send( chan[m], M );
              ELSE exit END;
           END;
(*3*) (*prepare for reduction pipe *)
      Q:= reduction poly in P;   u:=M/HM(Q);
      IF required THEN ip:= channel to next;
                       create( pt, NFpipe, (m, ip, PH) );
                       send( chan[i], pt2, pt, P); END;
(*4*) (*reduction pipe *)
      receive( chan[i], M' ); w:= next monomial in Q;
      LOOP IF finished THEN exit END;
           CASE T(M') ? T(u w) OF
                      >        DO send( chan[ip], M' );
                                  receive( chan[i], M' );
                      <        DO send( chan[ip], - u w );
                                  w:= next monomial in Q;
                      =        DO v:= M' - u w;
                                  IF v <> 0 THEN
                                     send( chan[ip], v ); END;
                                  receive( chan[i], M' );
                                  w:= next monomial in Q;
                END (*case*);
           END;
(*5*) (*finish*)
      IF required THEN send( chan[ip], fin );
                  ELSE send( chan[m], pt2 ); END;
      IF pt2 > 0 THEN join(pt2) END;
      dispose channel i;
```

FIG. 6. *Pipelined Normalform: NFpipe*

```
PROCEDURE GB(F);
(*1*) (*initialization*) G:=F;
      B:={ (f,g) | f, g in G };
(*2*) (*send and receive reduction polynomials *)
      LOOP
(*2.1*)    (*receive work*)
           WHILE required DO i:= work channel;
                 receive( chan[i], H );
                 IF H <> 0 THEN
                    IF red possible THEN send( chan[i], newPoly );
                                    ELSE update G and B END;
                    END;
(*2.2*)    (*next pair*)
           REPEAT (f,g):=next pair of B; S:=Spol(f,g);
                  UNTIL S <> 0 AND criterions say so;
(*2.3*)    (*send work*)
           IF required THEN i:= next channel;
                           create( pt, NFx, (i) );
                           send( chan[i], P, S );
                      ELSE i:= old channel;
                           send( chan[i], newPoly, S ); END;
           END (*loop*)
(*3*) (*stop working threads *)
      RETURN( G );
```

FIG. 7. *Parallel GB: main*

```
PROCEDURE NFx(i);
(*1*) (*initialization*) receive( chan[i], P, S );
      R:=NF(P,S);  (* or sequential NF *)
(*2*) (*request loop *)
      LOOP receive( chan[i], action );
           CASE action    OF
                finish    DO exit;
                send      DO send( chan[i], R );
                upd Poly  DO receive( chan[i], newPoly );
                             R:=NF( P+newPoly, R);
                upd + red DO receive( chan[i], newPoly, S );
                          R:=NF( P+newPoly, S);
                END CASE;
           END;
(*3*) (*finish*)
```

FIG. 8. *Parallel GB: NFx*

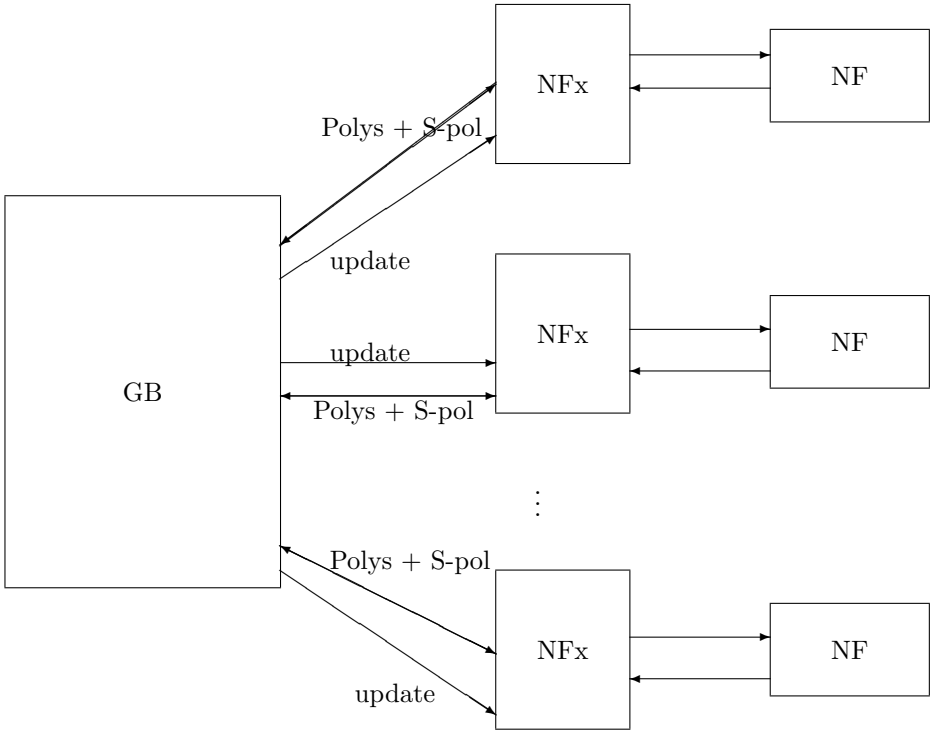


FIG. 9. *Parallel Gröbner Basis*

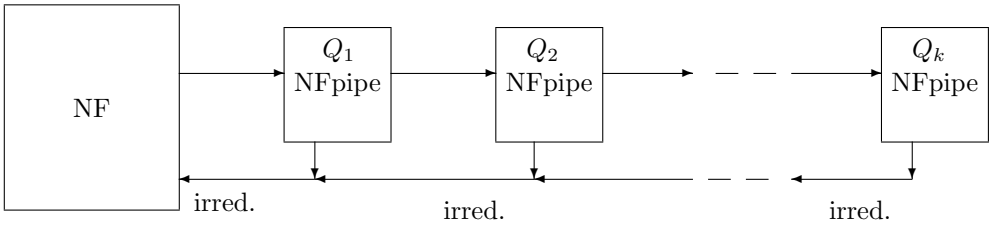
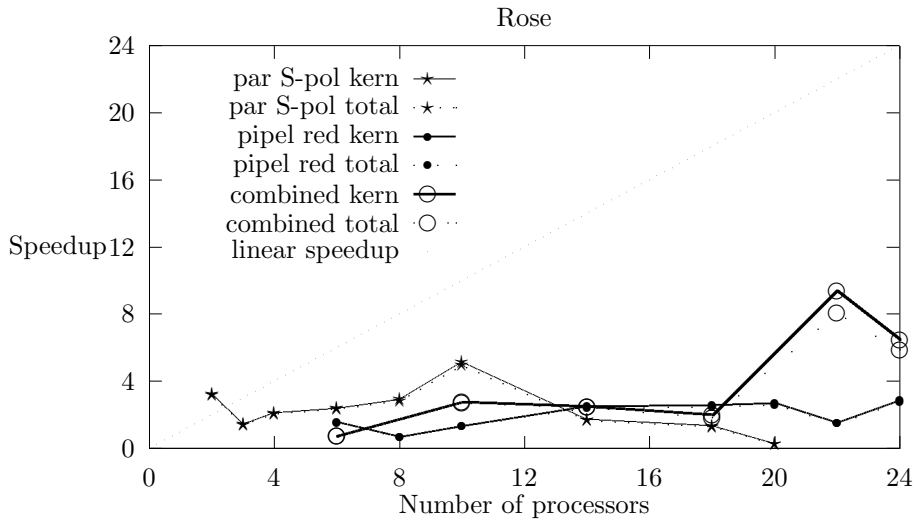
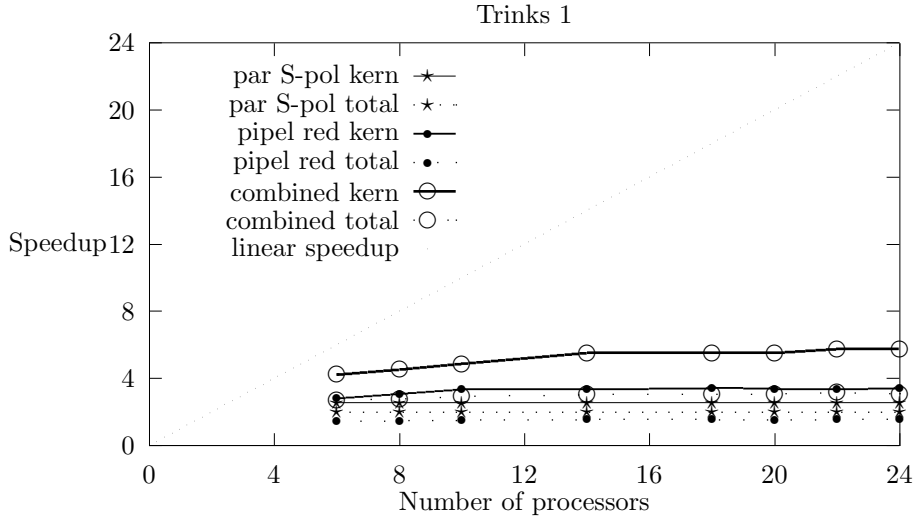


FIG. 10. *Pipelined Reduction*



kern = parallel part, total = parallel and sequential part

FIG. 11. *Example Rose*



kern = parallel part, total = parallel and sequential part

FIG. 12. *Example Trinks 1*

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. at GC time 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. However the KSR1 operating system should be more stable to run the examples.

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.