Parallel Buchberger Algorithms on Virtual Shared Memory KSR1

Heinz Kredel
Rechenzentrum Universität Mannheim
D-68131 Mannheim, Germany
kredel@rz.uni-mannheim.de

June 1994

Abstract

We develop parallel versions of Buchbergers Gröbner Basis algorithm for a virtual shared memory KSR1 computer. A coarse grain version does S-polynomial reduction concurrently and respects the same critical pair selection strategy as the sequential algorithm. A fine grain version parallelizes polynomial reduction in a pipeline and can be combined with the parallel S-polynomial reduction. The algorithms are designed for a virtual shared memory architecture and a dynamic memory management with concurrent garbage collection implemented in the MAS computer algebra system. We discuss the achieved speedup figures for up to 24 processors on some standard examples.

Keywords: Parallel Buchberger algorithm, Gröbner Bases, parallel S-polynomial reduction, pipelined polynomial reduction, virtual shared memory, POSIX threads, parallel garbage collection.

1 Introduction

This article discusses parallel versions of Buchberger's algorithm for the construction of Gröbner bases [4]. We focus on coarse and medium grain sizes for parallelization and equal critical pair selection strategies as existing in sequential Buchberger algorithms. We present implementations of the algorithms on a virtual shared memory KSR1 computer with a matching dynamic memory management system. We achieve good scaling of our parallel dynamic memory management and obtain reasonable speedups for parallel Gröbner base computation.

Virtual shared memory computers are as easy to program as shared memory computers, but there is a delay by a factor of about 8 to 10 in data transport from processor to processor compared to a shared memory computer. This requires that the dynamic memory management and thread system must be designed to match this architecture. We work with the MAS computer algebra system [23], which incorporates lots of sequential algorithms from the ALDES/SAC-2 system [7] plus sequential algorithms for the computation of Gröbner bases, commutative ideal theory, non-commutative 'solvable' polynomial rings and Gröbner bases [21] and comprehensive Gröbner bases [51, 24]. By its modular design it was only required to replace the storage module by a parallel version to make all the existing sequential algorithms usable. By further

adding thread functions (one to one based on the POSIX threads of KSR1) which are aware of the dynamic memory management all prerequisites for parallel programming where provided. Our system is designed to allow parallel garbage collection without interrupting the work on other processors together with low thread creation overhead.

Several authors have pointed out, that the underlying mathematics of Buchberger's algorithm allows parallelism on different levels. Namely the concurrent reduction of the S-polynomials, a pipelined reduction and parallel monomial arithmetic. These levels correspond to coarse, medium and fine parallel grain sizes and we obtained best results with the combination of the coarse and medium grain size versions on KSR1. For the parallel S-polynomial algorithm we spent much effort in assuring that a predictable selection strategy for critical pairs is followed. Thus enabling the use of well known, well studied selection strategies. Non-deterministic selection strategies as proposed by several authors *can* lead to better computing times than our algorithms, but also the contrary may happen. Especially on time-sharing systems a different workload on different processors may make the computations irreproducible.

We significantly contribute to the study of suitable grain sizes for an important algorithm in computer algebra (we obtain speedups of 10 on 22 processors, or more than 4 on 6 processors). We give a novel design of a parallel dynamic memory management with parallel garbage collection on a virtual shared memory computer (we obtain speedups of 12 on 16 processors for applications with low communication requirements, or nearly 4 on 4 processors). The memory system is based on the portable POSIX thread system and will be of use also on shared memory computers.

The plan of the article is as follows. After giving some motivation and a discussion of related work we present some background on virtual shared memory, the KSR1 architecture and a few facts about the MAS system in section 2. In section 3 we present our thread and multiprocessor memory management. After giving some background on Gröbner bases we discuss the parallel Buchberger algorithms in section 4. In section 5 we present the speedups for our parallel algorithms measured on a KSR1. Finally we draw some conclusions.

1.1 Motivation

Buchberger's algorithm for the computation of Gröbner bases solves an important problem. Gröbner bases for polynomial ideals 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 [4, 1]. Unfortunately the computation of such polynomial bases is notoriously hard, both with sequential and parallel algorithms. So 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 problems with parallel versions are as follows.

First, there is a theoretical limitation. The problem of computing Gröbner bases is exponential-space hard and also NP hard [1]. Since by the *parallel computation thesis* [17]: "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.

Second, computer algebra algorithms and in particular Buchberger's algorithm are *highly data dependent*. The reason for that is that computations in computer algebra are done without roundoff errors, so 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). For Buchberger's algorithm it is even not possible to predict the amount of work required from the input data. This has lead to dynamic memory management systems for the sequential algorithms and it is clear that some kind of dynamic processor (or thread) management is required for parallel computer algebra.

Third, as with the parallel Gauß algorithm in linear algebra, there are some inherent sequential parts which cannot be performed independently in parallel. E.g. for the Gauß algorithm the pivot element has to be known and for the Buchberger algorithm the head terms of the involved polynomials have to be known before work can proceed in some pipelined fashion. For the Buchberger algorithm moreover the adjunction of a new polynomial to the basis is a sequential process. Only after a new polynomial has been added a new parallel test round is meaningful.

Finally, the selection strategy for critical pairs is already in the sequential Buchberger algorithm of great importance and there is a whole industry working on improvements [1]. For the parallel Buchberger algorithm some authors proposed to employ the non-determinism provided by parallel task scheduling [43, 19, 46]. This can lead to very good speedups if by chance a fortunate selection strategy results. However also the contrary can happen. We and also others [5] focus on predictable strategies, which can employ the experience of well studied sequential selection strategies. One would idealy hope that a provable and hence a predictable, parallel selection strategy can be found. Note, that in general it is *not* possible to construct a parallel reduction algorithm, which produces exactly the same sequence of polynomials as the sequential algorithm since the low order terms may be reduced in different ways.

As long as there is no 'universal' parallel computation hardware and software concept, every attempt to implement computer algebra algorithms must somehow reflect the underlying machine architecture. Since the suitable grain size depends also on the architecture it is interesting to work on several levels (memory management, scheduling, algorithm parallelization) to explore the possible speedups. To obtain an overall speedup it is also important to study the mapping of a varying number of subproblems to a fixed number of processors. Our parallel memory management and thread management for virtual shared memory machines provides a good basis for the investigation of all such problems. On the coarse grain level our efforts in deriving a predictable selection strategy are also significant.

A possible parallelization method which has not been studied up to now is on a higher level. It is known that the computation of a Gröbner base highly depends on the chosen term ordering. So a possible algorithm could start the computation with respect to several (which ?) term orderings and use 'good' intermediate results from each computation. In this way also the construction of universal Gröbner bases could be speedup [1]. The computation of comprehensive Gröbner bases (some kind of tree of Gröbner bases over polynomial rings with parameters) could be parallelized by computing the subtrees on different threads [51, 24, 1].

1.2 Related Work

Theoretical studies on parallel computer algebra focus on parallel factoring and problems which can exploit parallel linear algebra. See e.g. the articles by Lenstra and von zur Gathen [16, 32].

Most reports on *experiences* and results of parallel computer algebra are from systems written from scratch or where the system source code was available. From the commercial systems some reports are about Maple [50, 6, 47] (workstation clusters

and transputers), and Reduce [13, 14, 33] (automatic compilation, vector processors). Mathematica has an external interface called 'MathLink', which could be useful on workstation clusters, but no results are reported up to now. A port of Axiom to workstation clusters (IBM SP1) using the PVM message passing library is in work.

Systems written mostly from scratch are MuPAD (Multi Processing Algebra Datatool) [15] for shared memory multiprocessors, PAC (Parallel Algebraic Computing) [38, 39] for hypercubes and transputers DSC (Distributed Symbolic Computation) [11, 12] for heterogeneous workstation clusters and MAO!! (Mechanized Algebraic Operations !!) [10] for massively parallel computers (CM-2 with *LISP). Software reusage and adaptation is reported from ALDES/SAC-2 based systems PARSAC-2 (PARallel SAC-2) [25, 26, 31], PACLIB (Parallel SACLIB) [19, 43] for shared memory multiprocessors and by Seitz for workstation clusters [45]. For further overviews see the workshop reports [9, 52], the report [49] and the further articles by Ponder [36, 37].

The main applications and *algorithms* chosen for parallelization were e.g. integer arithmetic [33, 30], real root isolation [8], solution of large sparse linear systems [12], Buchberger's algorithm [50, 34, 46, 18, 44, 47, 48], the related Knuth-Bendix completion procedure [5], power series multiplication and expansion [10], polynomial greatest common divisors and resultants [27, 28, 45], integer and polynomial factorization [11] and cylindrical algebraic decomposition [42, 20].

This studies showed that shared memory multiprocessor machines are well suited, and for problems with low communication requirements, workstation clusters are to a certain extent also suited for the implementation of computer algebra software.

For the parallel Buchberger algorithm the idea of parallel reduction of S-polynomials seems to be originated by Buchberger and was in the folklore for a while. First implementations have been reported by Vidal, Hawley and Schwab [48, 18, 44]. Pipelined reduction was proposed by Melenk and Neun and was also implemented by Schwab [34, 44].

In a different pipelined approach by Siegl each basis polynomial is dedicated to a different task and performs a complete reduction with respect to this polynomial. Although this algorithm performs only a partial reduction with respect to all basis polynomials (not even a top-reduction) it behaves well when combined with backward interreduction and S-polynomial generation [46].

2 Virtual Shared Memory and Threads

Multiprocessor computers with main memory shared among all processors suffer from a performance degradation for high numbers (typically $\geq 8, 16$) of processors due to contentions on the shared memory interconnection. On the other hand, multiprocessor computers with distributed memory are more difficult to program, since the data transport between the processors must be explicitly designed and programmed. Virtual shared memory (VSM) is a concept which is implemented on distributed memory multiprocessor computers, but hides the data transportation layer from the programmer, i.e. the programmer sees the computer as a shared memory multiprocessor machine. Since the data transportation is not for free, the challenge in programming virtual shared memory machines is to improve data locality, i.e. to avoid too much data movement.

Programming on multiprocessor computers is done using the threads model of computation. In the threads model 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. The threads model of computation is implemented e.g. by the POSIX threads library based on the MACH operating system kernel. A schematic overview of the software model gives figure 1.

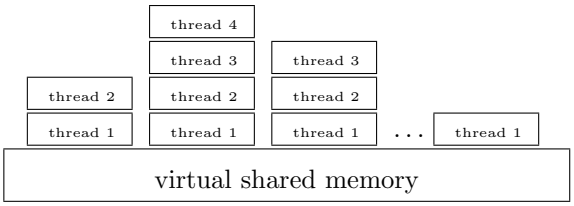


Figure 1: VSM and Threads

The thread functions for parallel computation are `create` and `join`. `create(p,f,x)` takes as arguments a thread identificator `p`, a function name `f` and a single argument for this function `x`. The meaning is that the function `f(x)` is executed in a separate thread (e.g. on a separate processor). `join(p,y)` takes as arguments a thread identificator `p`, and a place where the function result `y` is stored. `join` waits until the respective thread has terminated and has delivered its result. Thus

```
create(p,f,x); ...; join(p,y)
```

is equivalent to `y:=f(x)`.

Especially the KSR1 computer is a virtual shared memory 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. The latency to access data in the memory of another processor is typically a factor of 8 to 10 higher than the latency to access data in the processors local memory. A schematic overview of the hardware design gives figure 2. The KSR1 machine runs under OSF1 Unix. For further

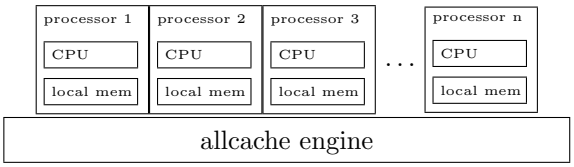


Figure 2: Architecture of KSR1

details on the architecture of the computer see e.g. [22, 40]. The new KSR2 has the same architecture except the CPU's are twice as fast.

3 Multiprocessor Memory Management

As pointed out already, computer algebra software needs some way of dynamic memory management with garbage collection (GC) since the amount of data and work generated by a run of an algorithm may vary dynamically. For an overview on state of the art dynamic memory management see e.g. the proceedings [2].

In parallel computer algebra there are two major attempts to implement dynamic memory management on shared memory multiprocessors (besides ignoring the problem). The *first* one in the PARSAC-2 system combines the generation and destruction of dynamic data with the creation and destruction of threads [25, 26, 29, 31]. This approach seems to scale very well to higher number of processors but there is a significant overhead for memory initialization during thread creation. However the employed user level C-thread package has higher initialization overhead than POSIX threads anyway. The *second* one in the PACLIB system uses a global shared dynamic memory [19, 43]. To reclaim unused data all threads are blocked until fresh memory becomes available. This method works well for small number of processors, but leads to idle processors during garbage collection.

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. In our first version we just reused the global shared dynamic memory but used lock/unlock to serialize the creation of new objects between threads and only one thread performed GC (therefore locking all activities on other processors). This version immediately worked, but was rather slow and lead to unbalanced processor usage. So it became apparent, that we must design a distributed memory management with low overhead introduced for thread creation.

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 about 8-10 times 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 on one processor is done by the *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. Moreover the cell space access functions (`FIRST` and `RED`) have been modified to ignore GC mark bits during access. The global variables are handled by one dedicated processor.

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 principle the thread handling is now similar to the virtual threads package of [31]. 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.

An example with suitable grain size and consequently good speedup figures comes from number theory [41]. The problem is to compute and count many characteristic polynomials. After the distribution of a few global constant matrices, the computation of the polynomials as determinants of locally constructed matrices, can proceed on each processor. At the end the number of different polynomials is computed by a master thread. In this example many garbage collections are performed on the used processors. The speedup figures for a batch of subproblems are shown in figure 4. The timings by the `time` function include the garbage collection times and the printing times (549 seconds on 1 processor). The dotted line shows the total time including the startup and initialization of the whole system (697 seconds on 1 processor).

4 Gröbner Bases

We asume, that the reader is familiar with the sequential version of Buchberger's algorithm [4, 1]. The algorithm takes a set of (multivariate) polynomials over a field as input and produces a new set of polynomials which generates the same polynomial ideal but additionally the reduction relation with respect to the new set of polynomials has unique normal forms. In the algorithm, first a set of critical pairs is generated, then the S-polynomial of each critical pair is checked if it can be reduced to zero. If not, then the resulting rest is added to the set of polynomials and new critical pairs are generated. The algorithm terminates if all S-polynomials of critical pairs reduce to zero

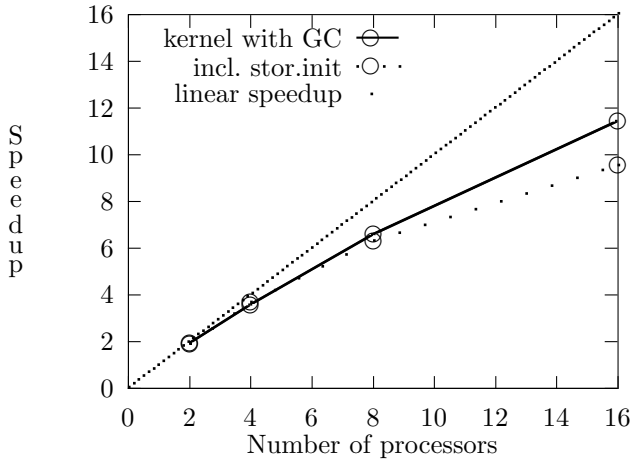


Figure 4: Characteristic Polynomials

(which is guarantied to happen by Dickson's lemma). For convenience the sequential Buchberger algorithm is shown in figure 5.

```
PROCEDURE seqGB(F);
(*1*) (*initialization*) G:=F;
      B:={ (f,g) | f, g in G };
(*2*) (*next pair*)
      WHILE B <> empty DO
            (f,g):=next pair of B wrt. strategy;
            remove (f,g) from B;
            IF criterions say so THEN
               s:=Spol(f,g); h:=NF(G,s);
               IF h <> 0 THEN G:= G + {h}
                  update B END;
               END;
            END;
(*3*) (*finish *) G:=autoreduce( G );
      RETURN( G );
```

Figure 5: Sequential GB

The implementation of the parallel version is based on the sequential algorithm as implemented in MAS [3]. The following parallel versions of the algorithm are denoted in a Modula-2 like language. `create` and `join` denote the MAS thread functions which are implemented using the POSIX thread primitives `create` and `join` and some scheduling code, as already discussed. Asynchronous message passing primitives `send` and blocking `receive` have obvious meanings and are implemented using POSIX thread primitives `lock`, `unlock`, `wait` and `signal` for synchronization and a (virtual) shared memory buffer for the actual data transfer. These primitives are only used to transfer pointers (list references) between threads in a synchronized way. The actual data is transferred by reading shared memory.

4.1 Parallel S-Polynomial Reduction

In a first version of the algorithm we created a new thread for each required reduction of a S-polynomial. But by this design the entire list of polynomials had to be copied multiple times between the threads (if on different processors). To minimize the copying we introduced reduction engines `NFx`, running in different threads, which copy data only for update purposes and new arguments.

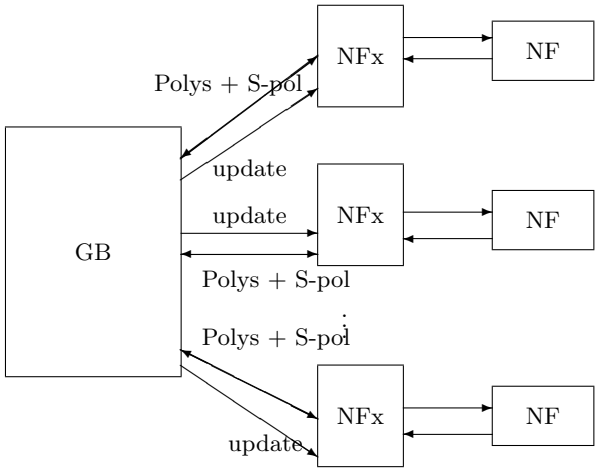


Figure 6: Parallel Gröbner Basis

The principal scheme of the Gröbner base algorithm with parallel S-polynomial reduction is shown in figure 6. It is intented that the subalgorithms `NFx` are placed on different processors and communicate with the main algorithm `GB`. The construction of the S-polynomials and the application of the criterions to avoid unnecessary reductions is performed in the `GB` algorithm. The basis polynomials (`Polys`) together with the S-polynomials (`S-pol`) are send to the `NFx` algorithm for processing by the actual reduction algorithm `NF`. The sending is done via a channel connecting both tasks, which is itself implemented by access to virtual shared memory (probably on a physically different processor). After the reduction has terminated on a particular thread the `NFx` waits for further instructions from `GB`: send the resulting polynomial, receive some new basis polynomials (`update`) and perform a further reduction, receive some new basis polynomials and a new S-polynomial and perform a reduction, or terminate. For more details see the algorithm in figure 7.

The `GB` algorithm performs all book-keeping to maintain and keep the subtasks busy. Step `2.2` is the traditional main loop of Buchberger's algorithm. In step `2.3` the subthreads `NFx` are started or reused as appropriate. The first step in the loop `2.1` implements the managing algorithm. The boolean expression `required` (with its companion in step `2.3`) together with the `work_channel` and the `send-receive` pair implements the chosen reduction strategy. `required` (in `2.1`) checks if there is some work under way or `B` is empty and we must wait anyway for delivery of a reduced polynomial. In step `2.3` the predicate `required` decides if an existing `NFx` should be used or a new one should be created, thereby trying to keep all processors busy while not overloading them. Besides the standard selection strategy of S-polynomials imple-

```
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;
```

Figure 7: Parallel GB: NFx

mented by `next pair` we have (in `2.1`) a further choice of which reduced polynomial shall be accepted for inclusion into the basis. We experienced best results when trying to accept the polynomials in 'mostly' the same sequence as the sequential algorithm would do so. A polynomial is possibly further reducible if meanwhile a new polynomial has been added to the basis `G`, but this new polynomial has not jet been transmitted to the reduction engine. For not accepted and not sufficiently reduced polynomials the `NFx` task is asked to perform further reductions. For more details see the algorithm in figure 8.

The correctness of the `GB` algorithm (i.e. the output produced is a Göbner base for the input) can be concluded as in the sequential case by showing that all required S-polynomials reduce to zero (this is the termination condition of the loop). The termination follows from Dickson's lemma since (all) polynomials added to the basis have irreducible head-terms with respect to the previous basis. The algorithms are interference free, since only local variables are used and data is explicitly transferred using send and receive. The algorithms are also deadlock free, since the sub-tasks are independent of each other and the master only executes receive when work is under way, in which case the sub-task receives the send request and then sends the polynomial.

4.2 Pipelined Reduction

In this sub-section we discuss the second parallelization point in the Gröbner bases algorithm, the pipelined reduction algorithm. In the Gauss algorithm with pivoting for the solution of systems of linear equations, a complete parallelization is not possible due to the fact, that the next pivot element becomes known too late. Also for the pipelined reduction only limited parallelization is possible since the next term to be reduced must first be computed and is not known in advance.

The principal scheme of the algorithm is shown in figure 9. It is intented that the subalgorithms `NFpi` are placed on different processors and communicate with the main algorithm `NF` and with the immediate successor and predecessor.

The main algorithm `NF` creates the first sub-worker and tells it the communication channels, the head terms of the basis polynomials and the (location of the) basis polynomials. In step `2` the monomials of the S-polynomial are send to the sub-worker and in step `3` the processed monomials are received. Although not shown explicitly the whole algorithm is designed such that the received monomials are already in the

```
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;
            send( chan[i], send); receive( chan[i], H);
            IF H <> 0 THEN
               IF reduction_possible
                  THEN send( chan[i], upd Poly );
                       send( chan[i], newPoly );
                  ELSE update G and B END; END;
      IF work_done THEN EXIT 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], upd + red );
                       send( chan[i], P, S );
                  ELSE i:= old channel;
                       send( chan[i], upd + red );
                       send( chan[i], newPoly, S) END;
      END (*loop*)
(*3*) (*stop working threads *) G:=autoreduce( G );
      RETURN( G );
```

Figure 8: Parallel GB: main

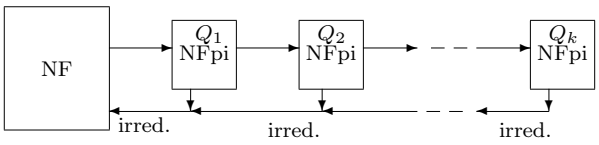


Figure 9: Pipelined Reduction

correct sorted termorder sequence. By this no further sorting in the `R:=R+M` statement is required (only list construction). For more details see the algorithm in figure 10.

The most complicated algorithm is the sub-worker algorithm `NFpi`. It combines a division step with a subtraction step on monomials piped through the algorithm. The algorithm first receives the (location of the) basis polynomials and the required communication channels. In step `2` monomials are received from the immediate predecessor until a reducible (divisible) monomial is found. Irreducible monomials are send to the master `NF`. In step `3` the reduction polynomial is transferred to the thread, the division is performed and if required the next sub-worker algorithm `NFpi` is started. Step `4` is the subtraction pipe loop. Based on the ordering of the terms under consideration (`T(M') ? T(u w)`) the next actions are determined. If the incoming monomial `M'` is *greater than* the actual monomial to be subtracted, `M'` is passed to the next sub-worker. If the incoming monomial `M'` is *less than* the actual monomial to be subtracted, the negative of the monomial (`-uv`) is passed to the next sub-worker. In case both terms are *equal* then the subtraction is performed (`v:= M' - u w;`) and the result is passed

```
PROCEDURE NF(P, S);
(*1*) (*initialization*) R:=0;
      m:= channel to master;
      i:= channel to next;
      create( pt, NFpi, (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 );
```

Figure 10: Pipelined Normalform: NF

to the next sub-worker. For more details see the algorithm in figure 11, however still not all details which make the algorithm work are shown. To implement top-reduction the loop in step 2 must be limited to one iteration, i.e. it must be replaced by a single if-statement.

Termination of the algorithms follows since the ordering between the terms is preserved in the pipelined algorithm and the one step reduction is noetherian. Correctness can be concluded from the correctness of the subtraction loop and the one step reduction. The algorithms are interference free, since only local variables are used and data is explicitly transferred using send and receive. The algorithms are also deadlock free, since the sub-tasks are independent of each other and the master sends a termination request at the end of step 2, which is then passed around to each subsequent task. In statements not shown, each receive in a sub-task is accompanied by termination detection code.

5 Results and Experiences

The timings are obtained by the UNIX '`time`' function on the KSR1, so they present the result which is delivered to the user. We experienced that the combination of the parallel reduction of S-polynomials (using the sequential selection strategy) with the pipelined reduction of them showed the best speedup figures. For the timings for some standard test examples [3] see figures 12 and 13. 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 Hawley for 16 processors and to the values reported by Schwab for 25 processors in the Trinks 1 example [18, 44]. The time for the sequential algorithm in the example Rose is 253 seconds for the kernel and 258 seconds total. For the example Trinks 1 it is 127 seconds for the kernel and 155 seconds total.

We used some event logging and visualization tools available on the KSR1 computer (`gist` display program for event logs) to get a better understanding of the activities going on during computation. A sample from a run of the Trinks 1 example using the sequential Gröbner base algorithm with the pipelined reduction is shown in figure 14. In the figure each line shows the activity on a particular thread. The numbers on the left side give processor number and the thread number by the scheme 'proc no' $\times$

```
PROCEDURE NFpi(m,i,PH);
(*1*) (*init*) 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 non_trivial_case THEN ip:= channel to next;
         create( pt, NFpi, (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*) IF non_trivial_case THEN send( chan[ip], fin );
         ELSE send( chan[m], pt2 ); END;
      IF pt2 > 0 THEN join(pt2) END;
      dispose channel i;
```

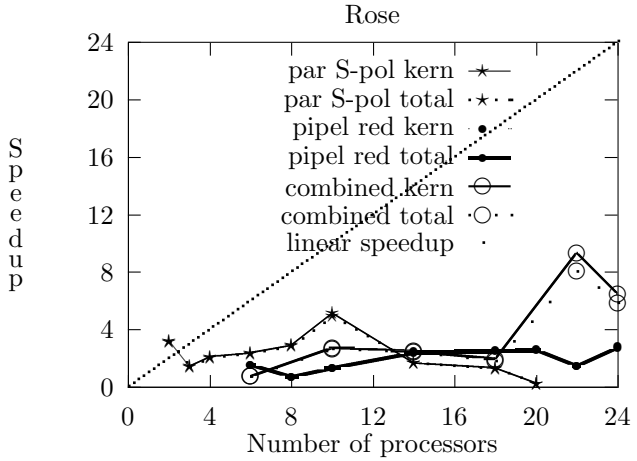
Figure 11: Pipelined Normalform: NFpi

$1000 +$ 'thread no' (e.g. number 7002 indicates the 2-nd thread on the 7-th processor). The line with 0 shows the master thread during the reduction process. Empty space shows no activity in the reduction process. In this case there is activity in S-polynomial construction, application of the different criteria and idle time waiting on the reduction results. We see that at the beginning of the run the grain size is to fine (only short peeks of activity on all threads). In the middle, when the size of polynomial coefficients increases the grain size seems appropriate (activities lasting for several seconds). This suggests that the algorithm should modify the grain size depending on the actual work required.

A overall 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 by lists, 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.

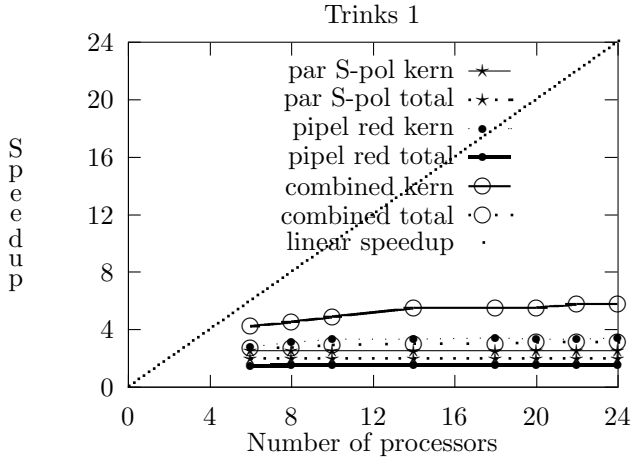
5.1 Porting efforts

In our installation at Mannheim we started also porting several systems (Maple, Reduce, PARI and MAS) to the KSR1 computer. Only for the MAS system the single processor version was relatively easy to port (as is also reported from other ALDES/SAC-2



kern = parallel part, total = parallel and sequential part

Figure 12: Example Rose



kern = parallel part, total = parallel and sequential part

Figure 13: Example Trinks 1

Figure 14: Visualization with gist

originated systems [25, 19, 45]). So after a few weeks the development of the multiprocessor version could be started. MAS was developed by the computer algebra group at the University of Passau, Germany since 1988 [23, 35]. It originates from the ALDES/SAC-2 system and its current version is 0.7 as of April 1993. 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 number of lines changed in MAS 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.

6 Conclusions

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; POSIX thread and MAS thread creation is 50 times more expensive than a function call. The KSR1 virtual shared memory concept (together with the OSF1 operating system) makes it easy to program this.

The original Buchberger algorithm is not very complicated (opposed to its correctness proof), but as we have seen the parallel S-polynomial and the pipelined reduction algorithms are quite complicated. To improve data locality for efficient usage of virtual shared memory we used 'engines' and explicit communication with send / receive instead of implicit communication via global variables. The new algorithms require several communication patterns (from shared variables to message passing with dynamic channel assignment), synchronization efforts and flow of control optimizations.

The challenge in parallel computer algebra on virtual shared memory multiprocessors 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.

References

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

[2] Y. Bekkers, J. Cohen (eds.), *Memory Management*. Proceedings, Springer LNCS 537, 1992.

[3] 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.

[4] B. Buchberger, *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*. Dissertation, University of Innsbruck, 1965.

[5] R. Bündgen, M. Göbel, W. Küchlin, *A Fine-Grained Parallel Completion Procedure*. To appear in Proc. ISSAC'94, Oxford, July 1994.

[6] B.W. Char, *Progress report on a system for general-purpose parallel symbolic algebraic computation*. Proc. ISSAC'90, ACM Press, pp 96-103, 1990.

[7] G.E. Collins, R. Loos, *ALDES and SAC-2 now available.* ACM SIGSAM Bulletin Vol. 12, No. 2, p 19, 1980.

[8] G.E. Collins, J.R. Johnson, W. Küchlin, *Parallel Real Root Isolation Using the Coefficient Sign Variation Method*. In [52], pp 71-88, 1990.

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

[10] A. Deprit, E. Deprit, *Massively Parallel Symbolic Computation*. ACM Press, 1989, pp 308-316.

[11] A. Diaz, E. Kaltofen, K. Schmitz, T. Valente, *DSC A System for Distributed Symbolic Computation*. Proc. ISSAC'91, ACM Press, pp 323-332, 1991.

[12] A. Diaz, M. Hitz, E. Kaltofen, A. Lobo, T. Valente, *Process Scheduling in DSC and the Large Sparse Linear Systems Challenge*. Proc. DISCO'93, Springer LNCS 722, pp 66-80, 1993.

[13] J. Fitch, *Can REDUCE be run in parallel* ? Proc. ISSAC'89, ACM Press, 1989, pp 155-162.

[14] J. Fitch, *Compiling For Parallelism*. In [9], pp 19-31.

[15] B. Fuchssteiner *et al.*, *MuPAD Benutzerhandbuch*. Birkhäuser Verlag, Basel, 1993. See also [49], pp 180-185.

[16] J. von zur Gathen, *Parallel Algorithms for Algebraic Problems*. SIAM J. Comput., Vol. 13, No. 4, November 1984, pp 802-824.

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

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

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

[20] H. Hong, *Parallelization of Quantifier Elimination on a Workstation Network*. Proc. AAECC-10, Springer LNCS 573, pp 170-179, 1993.

[21] A. Kandri-Rody, V. Weispfenning, *Non commutative Gröbner bases in algebras of solvable type.* J. Symb. Comp., **9**, pp 1–26, 1990.

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

[23] H. Kredel, *The MAS Specification Component*. Proc. PLILP'91, Springer LNCS 528, pp 39-50, 1991.

[24] H. Kredel, *Solvable Polynomial Rings.* Diss. Univ. Passau, 1992, Verlag Shaker, Aachen 1993.

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

[26] W. Küchlin, *The S-Threads Environment for Parallel Symbolic Computation*. In [52], pp 1-18, 1990.

[27] W. Küchlin, *On the Multi-Threaded Computation of Modular Polynomial Greatest Common Divisors*. Proc. Parallel Computation, Springer LNCS 591, pp 369-384, 1991.

[28] W. Küchlin, *On the Multi-Threaded Computation of Integral Polynomial Greatest Common Divisors*. Proc. ISSAC'91, ACM Press, pp 333-342, 1991.

[29] W. Küchlin, N.J. Nevin, *On Multi-Threaded List-Processing and Garbage Collection*. Proc. 3-rd IEEE Symp. Parallel and Distributed Processing, IEEE Press, pp 894-897, 1991.

[30] W. Küchlin, D. Lutz, N.J. Nevin, *Integer Multiplication in PARSAC-2 on Stock Microprocessors*. Proc. AAECC-9, Springer LNCS 539, pp 206-217, 1991.

[31] W. Küchlin, J. Ward, *Experiments with Virtual C Threads*. Proc. 4th IEEE Symp. Parallel and Distributed Processing, Arlington, IEEE Press, Dec. 1992.

[32] A.K. Lenstra, *Massively parallel computing and factoring*. Springer LNCS 583, pp 344-355.

[33] W. Neun, H. Melenk, *Implementation of the LISP-Arbitrary Precision Arithmetic for a Vector Processor*. In [9], pp 75-89.

[34] H. Melenk, W. Neun, *Parallel Polynomial Operations in the large Buchberger Algorithm*. In [9], pp 143-158.

[35] Computer Algebra Group Passau, *Modula-2 Algebra System, Version 0.7*. See eg. [49], pp 222-228.

[36] C.G. Ponder, *Parallelism and Algorithms for Algebraic Manipulation: Current Work*. SIGSAM Bulletin, Vol. 22, No. 3, July 1988, pp 7-14.

[37] C.G. Ponder, *Parallel Processors and Systems for Algebraic Manipulation: Current Work*. SIGSAM Bulletin, Vol. 22, No. 3, July 1988, pp 15-21.

[38] J-L. Roch, P. Senechaud, F. Siebert-Roch, G. Villard, *Computer Algebra on MIMD Machine*. SIGSAM Bulletin, Vol. 23, No. 1, January 1989, pp 16-32.

[39] J-L. Roch, *An Environment for Parallel Algebraic Computation*. In [52] pp 33-50.

[40] J. Rothnie, *Kendall Square Research Introduction to the KSR1*. In "Supercomputer'92", Meuer (ed.), Springer, Informatik aktuell, pp 104-114, 1992.

[41] B. Runge, *On Siegel modular forms*. Journal für reine und angewandte Mathematik (Crelle), Nr. 436, pp 57-85, 1993.

[42] B.D. Saunders, H.R. Lee, S.K. Abdali, *A Parallel Implementation of the Cylindrical Algebraic Decomposition Algorithm*. Proc. ISSAC'89, ACM Press, pp 298-307, 1989.

[43] W. Schreiner, H. Hong, *PACLIB – A System for Parallel Algebraic Computation on Shared Memory Multiprocessors*. 7-th Int. Parallel Processing Symp., IPPS'93, Newport Beach, CA, IEEE Press 1993, pp 56-61.

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

[45] S. Seitz, *Algebraic Computing on a Local Net*. In [52], pp 19-31.

[46] K. Siegl, *Gröbner Bases Computation in Strand: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages*. Diplomarbeit, University of Linz, 1990.

[47] K. Siegl, || MAPLE ||: *A System for Parallel Symbolic Computation*. 7-th Int. Parallel Processing Symp., IPPS'93, Newport Beach, CA, IEEE Press 1993, pp 62-57.

[48] J.-P. Vidal, *The computation of Gröbner bases on a shared memory multiprocessor*. Proc. DISCO '90, Springer LNCS 429, pp 81-90, 1990.

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

[50] S.M. Watt, *Bounded Parallelism in Computer Algebra*. Ph.D. Thesis, Dept. Comput. Sci., Univ. Waterloo, May 1986. Also as Tech Report CS-86-12.

[51] V. Weispfenning, *Comprehensive Gröbner bases*. J. Symb. Comp., **14**/1, pp 1–29, 1992. preprint in: Technical Report University of Passau, MIP-9003, 1990.

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