A Case Study of Multi-Threaded Gröbner Basis Completion

Beatrice Amrhein Oliver Gloor Wolfgang Küchlin

Wilhelm Schickard Institute for Computer Science
University of Tübingen, Germany

{amrhein,gloor,kuechlin}@informatik.uni-tuebingen.de

Abstract

We investigate sources of parallelism in the Gröbner Basis algorithm for their practical use on the desk-top. Our execution environment is a standard multi-processor workstation, and our parallel programming environment is PARSAC-2 on top of a multi-threaded operating system. We investigate the performance of two main variants of our master parallel algorithm on a standard set of examples. The first version exploits only work parallelism in a strategy compliant way. The second version investigates search parallelism in addition, where large super-linear speedups can be obtained. These speedups are due to improved S-polynomial selection behavior and therefore carry over to single processor machines. Since we obtain our parallel variants by a controlled variation of only a few parameters in the master algorithm, we obtain new insights into the way in which different sources of parallelism interact in Gröbner Basis completion.

1 Introduction

1.1 Overall Goals

The purpose of our work is to systematically study sources of parallelism in the Gröbner Basis computation which can be profitably exploited in practice on desk-top parallel machines. We try hard to produce parallel software that outperforms established sequential software and that researchers in the field could and would run on their workstations on a daily basis. While we have not yet fully succeeded in this respect, we have found a form for the Gröbner Basis algorithm which is largely compatible with traditional implementations, including modern selection heuristics and deletion criteria, yet is parallelizable and produces significant speedups on many examples, even when run on a traditional uniprocessor.

This work is part of our overall effort to create PARSAC-2 [17], a parallel Computer Algebra library targeted for execution on networks of multi-processor workstations. A comprehensive introduction and motivation for this approach has been given in [19].

A major point of our work, which sets it apart from most previous efforts, is our reliance on standards in both hard- and software. We start from the Gröbner Basis implementation GRÖBNER [24], which uses the SACLIB [4] library written in C. GRÖBNER is then parallelized using the multi-threading support of PARSAC-2. This, in turn, relies on standard POSIX threads, in our case provided by the Solaris 2.x operating system executing on single and multi-processor desk-top machines.

1.2 The Parallel Completion Challenge

The critical pair / completion procedure [5] exists in two related forms, Knuth-Bendix (KB) term completion and Gröbner Basis (GB) polynomial completion. The completion procedure can be formulated as a set of logically independent inference rules. *In theory*, these can be applied concurrently, but it is well known that such an installation would be horrendously inefficient due to excess work. *In practice*, it is notoriously hard to obtain significant speedups, because a good sequential implementation including modern selection heuristics and deletion criteria can avoid much of the work on which a parallel implementation thrives.

The efficiency benefits of performing only the *best* next computation (as selected by a completion strategy) must be balanced against the speedup benefits of performing multiple such computations in parallel. Selection heuristics and deletion criteria favor the sequential algorithm, while the parallel algorithm achieves speedups if many S-polynomials must be simplified (work parallelism), or if it can improve the selection strategy (search parallelism).

In the framework of a desk-top environment, the parallel algorithm must perform on a par with a high-quality sequential algorithm when executed on a uniprocessor workstation. In addition, it must be able to utilize the power of multiple processors if they are available in the same or in other workstations.

1.3 Overview of Our Approach

In the Knuth-Bendix case, where selection heuristics and deletion criteria are comparatively weak, good speedups can be achieved by the equivalent of converting *all* pairs to S-polynomials and simplifying all of them in parallel [9, 10]. With a perfect (omniscient) selection strategy, it might be best to go to the other extreme and select only a single pair to simplify and to orient into a basis polynomial. In this work, we experiment with the middle ground, allocating a buffer of some width w of S-polynomials which we simplify in parallel and from which we finally select the new basis element.

For parallelizing the Gröbner Basis algorithm we started from the sequential system GRÖBNER [24]. Then we rearranged the completion loop of GRÖBNER (cf. Section 4) in order to obtain a sequential master implementation suitable for practical parallelization.

In the main conceptual part of our work, described in Section 4, we then parallelized the master implementation by forking several procedure calls (mostly for the normalization of S-polynomials) as separate concurrent threads of control. By setting a few switches in the parallel master implementation, we obtained two major variants of a parallel GB algorithm.

The first one uses only *work parallelism* and is thus *strategy compliant*. It carries out an amount of work in each completion cycle which is exactly the same for the sequential and the parallel form and leads to exactly the same completion sequence (polynomials introduced into the basis). This form exhibits stable and predictable behavior, but shows only moderate parallel speedups. The second one utilizes *search parallelism* in addition, so that the time in which a candidate polynomial is normalized affects the strategy decision of which polynomial to select as the next base element. This form exhibits less stable, if not chaotic, behavior, but frequently leads to super-linear speedups, that is, the parallel code performs a superior selection strategy and runs faster than the sequential reference even if executed on a single processor.

In the experimental part of our work, we compared the performance impact of various settings of our switches in the parallel master implementation, and thus arrived at the exact settings for our two main parallel forms of the GB algorithm. We document the performance of these two variants on a large number of standard examples in Section 5. In particular, our data shed new light on the relative merits of work parallelism vs. strategy parallelism, which have been intermingled in most previous work. It is one of our main results that strategy parallelism can be exploited with relative ease by a parallel implementation even on uniprocessor systems, and that it may achieve dramatic speedups (results vs. no results in useful time) even over a high quality sequential implementation including modern pair selection heuristics [12, 15] and deletion criteria [14].

2 Related Work

Suggestions for the parallelization of GB completion[1] go back to the mid-80's. Buchberger [7] proposed to parallelize the simplification of S-polynomials as well as their orientation into basis polynomials. He suggested the use of special parallel hardware [6]. He immediately pointed out the difficulties with synchronization, because simplification depends on the basis, which may be asynchronously changed by orientation, and because the application of *deletion criteria (confluence criteria)* [14] may asynchronously change the set of S-polynomials while it is simplified.

Deletion criteria also tend to negate the benefits of eagerly converting pairs into S-polynomials and simplifying them. In addition, the *Sugar* strategy [15] and similar heuristics [12] subsequently improved the selection of the *best* critical pair candidate for inclusion into the basis. Thus, they tend to negate the benefit of a better completion strategy that results if candidates can be selected from a large pool of simplified S-polynomials instead of from critical pairs.

[1]We now assume familiarity with the GB algorithm; cf. Section 4.

Much of the previous work on parallelizing the GB computation used non-standard methods, such as experimental programming languages, ad hoc implementations without Sugar or criteria, or special parallel hardware with unknown performance characteristics. It is therefore often difficult to judge the practical significance of reported speedups.

Vidal [22] realized Buchberger's design of parallel simplification and orientation on a 16 processor shared memory Encore. His new implementation was in C using Mach's C Threads.

Chakrabarti and Yelick [11] carried Vidal's work over to a 128-processor CM-5 with distributed memory. Each processor ran a copy of the completion loop on a private segment of the critical pair store. Thus S-polynomials are oriented into basis polynomials in parallel, and new basis elements are communicated asynchronously to all other processors.

Sawada, Terasaki, and Aiba [21] distribute the set of S-polynomials between different reducer workers. Their system, written in the language KL-1, runs on a parallel inference machine (PIM) with 256 processors.

Attardi and Traverso [1] proposed a strategy compliant distributed memory parallel Gröbner Basis algorithm, but reported limited success in practice.

Faugère [13] presents a novel hybrid method to compute Gröbner Bases over $\mathbb{Q}[x_1, \ldots, x_n]$. He concurrently computes an image basis over a finite field, which is used to predict zero reductions of S-polynomials in the main computation. Its parallel version, derived from the state-of-the-art sequential GB system, is based on a static client-server distribution onto the available processors which does not scale with problem size. Faugère also analyzed the dependencies in the deduction of S-polynomials for some examples. His results indicate that Gröbner Basis completion is inherently sequential, with a parallel content of only a few threads of control.

This analysis seems to be confirmed in most of the experiments reported in the literature. Significant speedups seem possible if super-linear effects can be brought to bear, but with a single exception not much gain has been reported beyond one or two dozen processors. In addition, few parallel systems have demonstrated that they are really faster than a high quality sequential system run on a standard workstation.

3 Parallel System Support

3.1 Parallel Hardware

The target hardware for PARSAC-2 is a shared memory multi-processor workstation running a multi-threaded operating system. For the present work we used a 4 processor SPARCstation 10 with 64MB main memory and four 66MHz Ross Hypersparc processors, running Solaris 2.4.

3.2 PARSAC-2

Our parallel Computer Algebra system is PARSAC-2 [17, 19], which contains the *S-threads* [18] parallel symbolic programming environment and libraries of sequential and multi-threaded parallel code.

S-threads is *middle-ware* between the hardware and operating system (OS) below, and the application software above. Most importantly, an S-thread extends an OS thread by a local heap segment, so that it can allocate list-cells concurrently. Practically all sequential procedures of the SAC-2/SACLIB [4] library can now be executed concurrently, by forking them on a separate S-thread.

S-threads is also designed to support *virtual parallel programming:* an algorithm is to be parallelized according to its inherent logical parallel structure, and the resulting logical threads of control are mapped at run-time, by the S-threads system, to real concurrency provided by the hardware and the operating system. Thus the same executable can run on 1, 2, or more processors, and S-threads can be ported to different operating systems, preserving application code.

For the current work, S-threads was ported to Solaris 2, and extended in functionality to support the concept of *thread groups* (cf. Section 3.3.3 below). The memory management subsystem of S-threads is currently in an interim state (awaiting improved functionality in Solaris 2.5). Therefore, all timings have been stripped of their garbage collection content.

The fork/join concept of shared memory parallel programming can be carried over to the distributed memory network, provided all global access to data-structures can be confined to parameters that can be packed with fork/join. We have extended S-threads to a system called DTS which transports fork/join calls across the net, and we have experimented with the distributed computation of multivariate polynomial resultants [3]. Thus, while all experiments of the present work used shared memory, future extension to a network of several multi-processors is feasible within the same system environment.

3.3 Higher Parallelization Concepts in PARSAC-2

3.3.1 Work Parallelism

Pure Work Parallelism ($\forall$-Parallelism) is the concept supported most directly by the fork/join paradigm of multithreading. A given amount of work is divided into multiple threads of control and forked onto a number of threads. Each thread is finally collected (joined) again by the parent, and its result is retrieved.

3.3.2 Search Parallelism

Pure Search Parallelism ($\exists$-Parallelism) reflects a parallel search for some item. A number of search threads are forked, and the first child to find the item synchronizes with the parent to deliver the result. The parent subsequently terminates the other children, not being interested in their results.

This kind of parallelism arises frequently in automated theorem proving. In the Gröbner Basis algorithm, we use a form of it to search for the first of a number of S-polynomials which is fully reduced by the current basis.

The synchronization needed by search parallelism is now supported in S-threads by the concept of thread groups.

3.3.3 Thread Groups

A *thread group* is a collection of threads (usually working towards a common objective), similar to a process group in the traditional UNIX world. PARSAC-2 thread groups are designed to support both work and search parallelism in a uniform way.

After a thread group has been opened (`thr_group g;`), threads can be forked (`thr_group_fork(g, func, arg)`) into the group at any time. The parent can wait for the next thread of the group (`thr_join_any(g)`), or it can send a signal to the entire group, for example to terminate or to suspend all threads.

In GB completion, we employ the thread group to support the concurrent reduction of S-polynomials. Here, the processing of the result is relatively trivial (computing a strategy value such as the total degree) and the thread group would not be essential to support this task as pure work parallelism.

On shared memory machines with fast context switching, the main use of a thread group is in supporting search parallelism. In this case, all threads of the group execute as concurrent OS threads. On a uniprocessor machine, execution of these threads will be interleaved to maintain logical concurrency.

In our case, we wish to find an irreducible S-polynomial as quickly as possible. The parent forks a number of reduction tasks and waits for the first (few) to produce a result; the other tasks are suspended in order to be resumed later.

4 Algorithm Organization

4.1 The Standard Gröbner Basis Algorithm

Buchberger's Gröbner Basis algorithm [2] consists of a main completion loop with four steps:

❶ select (according to some heuristic) one of the remaining critical pairs and compute its S-polynomial.

❷ reduce the S-polynomial with respect to the current basis until it is irreducible; if it becomes zero, delete it and go back to step ❶.

❸ insert the (non-zero) reductum into the basis.

❹ form the new pairs (caused by the new basis element) and apply the deletion criteria on all pairs.

Our experiments have confirmed the results in [8] that the reduction step ❷ is the most time consuming part. Therefore we now rearrange the completion loop slightly to arrive at a sequential reference algorithm which performs work on several S-polynomials that can later be parallelized.

4.2 The Sequential Reference Algorithm

In this version of the algorithm, we increase the buffer for S-polynomials from width 1 to a flexible size w. In the new completion loop, illustrated in Figure 1, we now select ① several critical pairs and convert ❶ them to S-polynomials. Then, we simplify ❷ all of the polynomials in the buffer until they are irreducible with respect to the current basis. One of these is finally selected ②, according to a new selection heuristic, for insertion ❸ into the basis. Then, we form the new pairs as usual ❹ and apply deletion criteria both on the pairs and on the buffer of reduced S-polynomials.

This algorithm admits a new, two-level selection heuristic. The level one heuristic ① operates only on critical pairs, similar to the traditional algorithm. It can only be based on relatively rough estimates, such as Sugar, for the quality of the offspring that the pair will produce. The level two heuristic ②, however, can compute a quality measure based on the actual normalized S-polynomials which are stored in the buffer, and which are candidates for inclusion in the basis. Note that this part also depends on the size of the buffer.

We now obtain our parallel algorithms by specifying how exactly the operations on the S-polynomial buffer are executed.

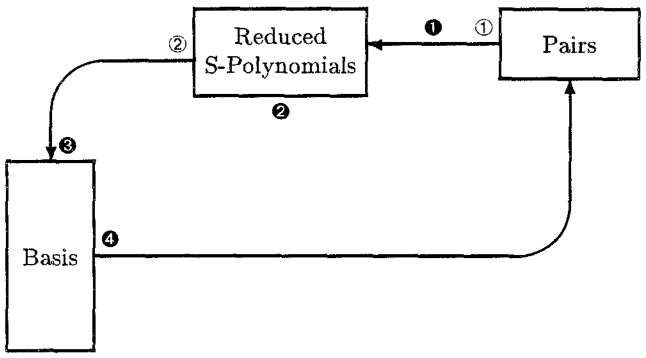


Figure 1: Sequential Reference Completion Structure

4.3 The Master Parallel Algorithm

The master parallel algorithm (cf. Table 1) uses a parameterized internal structure of the S-polynomial buffer, illustrated in Figure 2. By giving exact settings for the parameters, we shall obtain our concrete parallel incarnations.

A straight-forward *work parallelization* will perform all reductions ❷ in the buffer in parallel. A *narrow search parallelization* will wait only for the *first* result, suspend the other reductions, insert the result into the basis ❸, and finish the iteration as usual. At the beginning of the next iteration, the buffer is topped up by selecting further pairs ① and converting them to S-polynomials ❶, and the reductions are started (respectively restarted ❺).

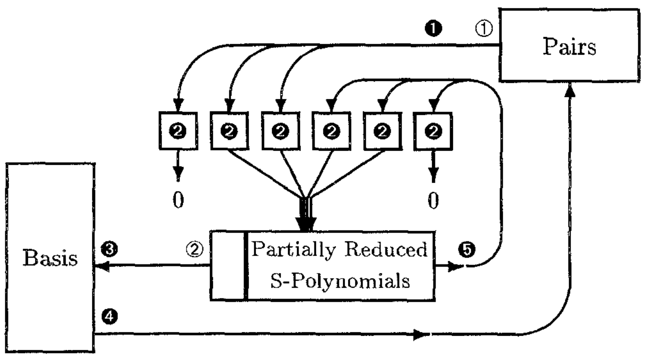


Figure 2: Parallel Gröbner Basis Completion

Our experiments show that it is even profitable to perform a *wide search parallelization*, waiting for *several* results before we suspend all other reductions. We then select ② one of the irreducibles and insert it into the basis ❸, and finish the iteration as usual. At the beginning of the next iteration, the buffer is topped up by selecting further pairs ① and converting them to S-polynomials ❶, and all reductions are resumed ❺, respectively started. (This includes re-starting reductions on previously irreducible polynomials as the basis has increased.)

For the parallel algorithm to be competitive with the sequential one, it is necessary to avoid excess work. Thus, the optimal application of the deletion criteria [14] is of prime importance. For this reason, we accept the synchronization point after every reduction cycle to apply the criteria even on partially reduced polynomials, and so to cut unnecessary reduction steps.

The following parameters specify the internal buffer structure.

```
Parallel_Groebner_Basis_Completion
Input, Output: Set of Polynomials $G$

build initial pairs $P$ from $G$;
apply criteria;

while(there are pairs in $P$
  or there are partially reduced polynomials in $PR$)
{
  restart ❺ the reduction ❷ of the partially reduced
    polynomials in $PR$;                                        } fork
  while(not MAX_CONC_RED reductions run and $P \neq \emptyset$)
    {build S-polynomials from pairs $P$ ❶
       and start their reduction ❷;}

  while(not yet NO_OF_IRRED non-zero reductions have
       terminated and there are reductions running)
    {
     /* optional: MAINTAIN_LOAD(); */
     collect the next result of the reduction processes;
     if(the result is non-zero)
        append the result to $PR$;                              } join
    }
  if($PR \neq \emptyset$)
    {select an (irreducible) element $p$ from $PR$ ②;}
  stop all reductions;
  append non-zero intermediate results to $PR$;

  if($PR \neq \emptyset$)   /* we found $p \neq 0$ */
    {
     insert $p$ into the base $G$ ❸;
     add the new pairs to $P$ ❹;
     apply the criteria on $P$ and $PR$;
    }
}
interreduce $G$ to obtain a reduced Gröbner basis;
```

Table 1: The Parallel Completion Algorithm

- MAX_CONC_RED: the (maximum) number of reductions running concurrently (= buffer width w).
- NO_OF_IRRED: the (maximum) number of reductions that we wait for and from which we select the best for the insertion.
- MIN_CONC_RED: we start additional reductions if, due to zero reductions, the number of concurrent reductions drops below MIN_CONC_RED.

If there are many zero-reductions, we would wait for NO_OF_IRRED results while there may only be a few reductions running with the system starving for parallelism. Therefore, we start new reductions as soon as the number of concurrent reductions has dropped below the low water mark MIN_CONC_RED. This is accomplished by function

```
MAINTAIN_LOAD()
if(fewer than MIN_CONC_RED reductions run)
   while(not MAX_CONC_RED reductions run and $P \neq \emptyset$)
      build S-polynomials from pairs $P$ ❶
         and start their reduction ❷;
```

With these parameters, we obtain pure work parallelism if NO_OF_IRRED = MAX_CONC_RED and MIN_CONC_RED = 0 (MAINTAIN_LOAD() is not executed). We obtain search parallelism (with width NO_OF_IRRED) if NO_OF_IRRED < MAX_CONC_RED.

Note that the *selection strategy (completion strategy)* for the selection of the new basis polynomial is parameterized by the heuristics ① and ② together with NO_OF_IRRED, which are all static properties. However the actual *selection behaviour* depends also on the exact times at which the irreducibles are produced, because these determine which ones are considered by the level two heuristic at point ②. These times are dynamic properties, depending for example on the threads scheduling of the OS which is affected by external load and asynchronous system events. Parallel selection behavior cannot be precisely emulated by time-slicing the threads, because the slices would have to be as thin as a single machine cycle. In this respect, there is no real substitute for real parallelism.

5 Empirical Results

The problem is now how to set the parameters in the master parallel algorithm. First, we get the major options of work and search parallelism, respectively, under the conditions mentioned above. Within each option, we still have to determine actual settings under conflicting objectives.

On the one hand, it is desirable to keep the number w of S-polynomials large. We thus generate parallel work and improve the level two selection heuristic. Also, due to system reasons (mainly page faults), the processors are most efficiently utilized if we overload them two or three times.

On the other hand, it is also desirable to keep the number w of S-polynomials small. The time and space needed for the Gröbner basis computation mainly depends on the total number of reductions. If we start too many reductions, a lot of unnecessary work is done since reductions

- may later be recognized as superfluous by the deletion criteria or
- may be cut short by new elements of the base.

Furthermore, maintaining a lot of partially reduced polynomials takes a lot of space.

Thus the aim of our parameter settings is to make good use of our processors without performing too much useless work.

Whenever we obtain a spread of timings for a particular parameter setting, we display the distribution of the times according to the color coding in Figure 3. Those 20% of the values that are located around the median are colored black. The bulk of the values (60%) are displayed in dark gray, while the light gray covers the few runaways.

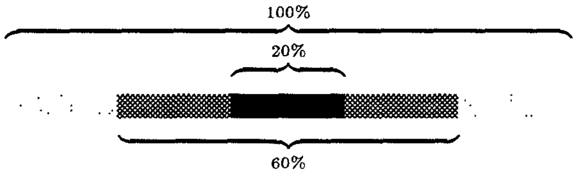


Figure 3: The Color Legend for the Figures

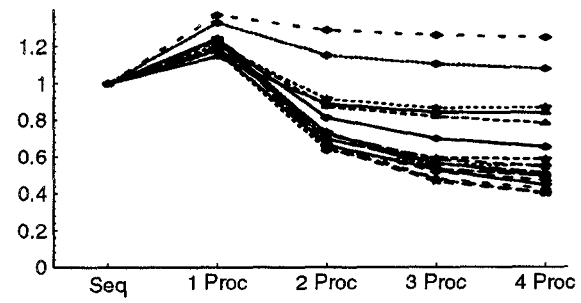
The results we present in the following are from standard examples that are publicly accessible [20].

5.1 Experiments with Work Parallelism

In a first approach, we performed a strategy compliant parallelization to compare our setup with the sequential reference algorithm (cf. Section 4.2). For that, we start up to MAX_CONC_RED = 8 reductions and wait for the termination of all of them (NO_OF_IRRED = MAX_CONC_RED). In addition, we dispense with the reloading (MAINTAIN_LOAD) of reductions (MIN_CONC_RED = 0). In this way, the course of the algorithm is independent of the number of processors and can also be performed sequentially, provided that the same number of reductions is started. In fact, with this setup, the parameters ① and ② completely determine the sequence of polynomials inserted into the basis.

The figures of this section display the times obtained with one to four processors as well as the time needed to compute the same example with the sequential reference completion algorithm ("Seq").

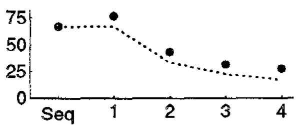
In the following figure, we show a representative selection of the examples we computed. The timings are scaled relative to the sequential times.



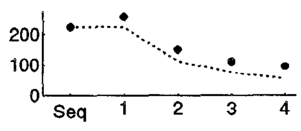
The profit of this work parallel approach depends on the distribution of the reduction times of the S-polynomials. If most of the reductions need about the same time, the processors are equally busy and speedup is almost linear. In contrast, if the reduction times vary over a large spectrum, one single lengthy reduction may cause all other processors to remain idle and there is no speedup.

Whenever we compute graded reverse lexicographic or total degree lexicographic Gröbner Bases, we set both selection orderings ① and ② according to the chosen term ordering. That is, we select the pair whose *lcm* of the leading terms is the smallest, and from the list of reduced S-polynomials the smallest with respect to the term ordering.

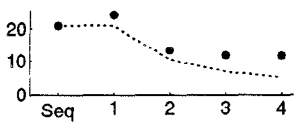
Here are some typical examples of this strategy-compliant parallelism. Displayed are the times in seconds for one to four processors. The theoretically optimal speedup is sketched in as a dashed line.



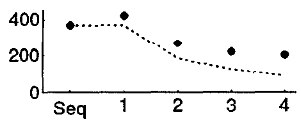
Katsura 5, graded reverse lex



Cohn $\sqrt{2}$, graded reverse lex



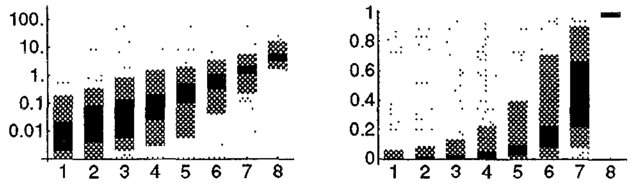
Hoffman 3, graded reverse lex



Cassou, graded reverse lex

The first two examples show almost optimal speedups; the third and forth examples have still a nominal speedup between one and two processors.

For an improvement of the efficieny of the parallelization, we need to take a closer look at the reduction times. In each iteration, the crucial point is how much longer the last terminating reduction takes in relation to the others. Therefore, we measured in each iteration of the algorithm the time for each reduction. For the following figures, we considered those iterations of our examples[2] with longest reduction time greater than one second. (If there are fewer than eight reductions, we fill the list with zeros.)



The Distributions of the Reduction Times in an Iteration from the Shortest (1) to the Longest Reduction (8)

The left figure shows the distribution of the reduction times—in logarithmic scale. In the right figure, the duration of the longest reduction is scaled to one. The dark region covers the significant results.

We notice the enormous spread in reduction times within an iteration, covering orders of magnitude. Thus the figures indicate that—in spite of the overall speedups obtained—a lot of the time in an iteration is used for the reduction that terminates last. In particular, if there are no zero reductions and no deletions in the buffer by the criteria, precisely one new S-polynomial is needed to refill the buffer. Then, as the re-reduction of the former results is usually performed fast, the time needed for the next iteration mainly depends on the reduction of the single newly inserted S-polynomial, and this tends to be a sequential bottleneck as only one job has to be performed (or finished).

One way to overcome this and to introduce more concurrency in the reductions is to increase the buffer width w. (We could even build in every step as many new S-polynomials as there are processors. Then, however, the buffer of the reduced polynomials keeps growing.) Then, it is more likely that several polynomials of the buffer either reduce to zero or are eliminated by the criteria. Thus, we reload more polynomials and the work is better balanced. Disadvantages of this approach are discussed above.

5.2 Experiments with Search Parallelism

As we already pointed out, in search parallelism we do not wait for the termination of all reductions but wait for only NO_OF_IRRED many.

We learned from our experiments that it is advantageous to set the parameters (at run-time) according to the number of processors. We obtain the best results if we do not start more parallel reductions than three times the number of processors. This limits the number of context switches and the need of memory. To prevent starvation, we start new reductions as soon as half of the reductions have terminated, which still guarantees oversaturation. The listed values of NO_OF_IRRED produce an ample choice of reduced polynomials without the disadvantage of causing much superfluous work.

[2]All we computed with work parallelism

These are our actual settings for the parallelization parameters.

Number of Processors	1	2	3	4	n
MAX_CONC_RED	3	6	9	12	$3n$
MIN_CONC_RED	2	3	5	6	$\lceil 3n/2 \rceil$
NO_OF_IRRED	1	2	2	3	$1 + \lfloor n/2 \rfloor$

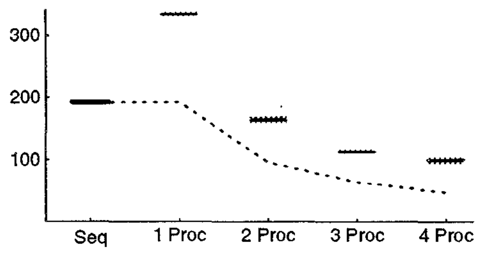
Compared with the sequential algorithm, this setup leads to a selection behavior that depends on the run and cannot be fully simulated sequentially. In spite of having the same selection ordering, the course of the parallel algorithm can be completely different to that of the sequential algorithm. However, as this setup is usually very powerful, we also obtain super-linear speedups for a lot of examples.

Furthermore, as the selection strategy is so crucial (especially in the sequential case), we often can observe a kind of chaotic behavior as the sequence of the insertion can differ. Therefore, we performed 10 runs for each case; the distribution of the results can be read off by the color coding according to Figure 3. If there is one runaway in a direction, this is indicated in light gray. Furthermore, if one or two values are out of scope (or missing), this is indicated by . If this applies to more values, we use ▨ and ■.

We have investigated Gröbner basis completions of many examples and present here some representative results. Displayed are the runtimes in seconds for one to four processors. For comparison purposes, we display the time for the standard sequential run (cf. Section 4.1) using GRÖBNER [24] with the same selection ordering on the pairs ① = ❶. The theoretically optimal speedups computed from the standard sequential computation is sketched in as a dashed line.

5.2.1 Moderate Speedups

A part of our examples behave as one might predict: we observe the parallelism overhead with one processor (compared with the sequential algorithm) and gain some speedup with more processors.



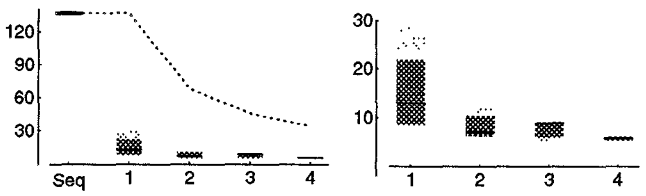
Katsura 5, total degree lexicographic

Here, the change of the strategy has no decisive influence on the computation. In particular, the timings fall within rather narrow bounds. Some of the concurrent reductions are not pertinent (i.e., they are obsolete due to the criteria or would later be performed a lot faster). Moreover, as some suboptimal results are being inserted, we subsequently create new pairs that were not created in the sequential case. Therefore, the timings for one processor are larger than in the sequential case (note that with one processor we start three concurrent reductions and wait for the first result). Still, the computation of these examples can be speeded up by a factor 1.9 using four processors.

5.2.2 Superlinear Speedups

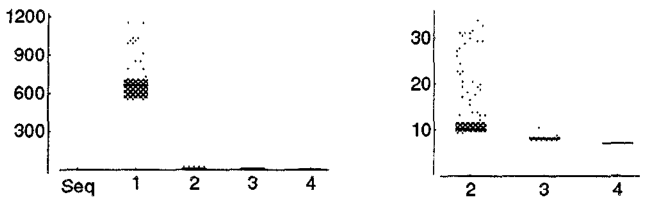
For a lot of examples we observe super-linear speedups, that is, we gain—compared to the sequential algorithm—a factor much larger than the number of processors used.

Remark: there are several selection strategies possible to compute lexicographic Gröbner Bases. For ① we either use the smallest *lcm* or the sugar selection strategy. For ② we use lexicographic ordering or choose the polynomial with the smallest true total degree[3].

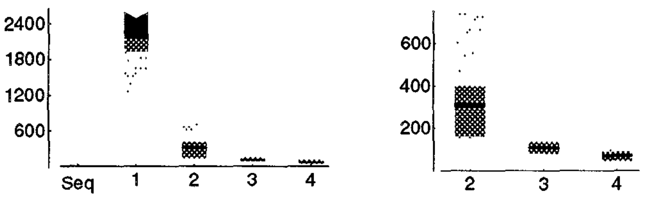


Wood, lexicographic (right: closer look)

This example shows super-linear speedups caused by the new selection strategy. The right figure is a close up of the left, without the time for the sequential algorithm. The speedup between 1 and 4 processors is between 2 and 5, dependent on the run. In addition, we observe slightly chaotic behavior with one processor.

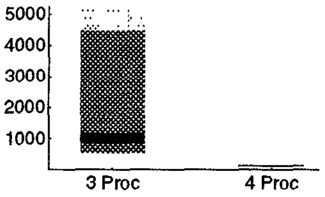


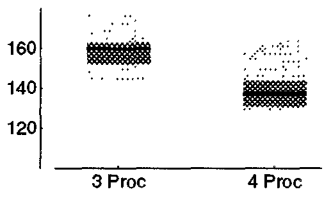
Morgenstern, lexicographic



Cohn $\sqrt{2}$, total degree lexicographic

These two examples did not terminate with the sequential algorithm, either because of memory overflow or because of an exceeded time bound. However, they terminated with the parallel algorithm on one processor. We can read off an additional speedup of up to one hundred between one and four processors.





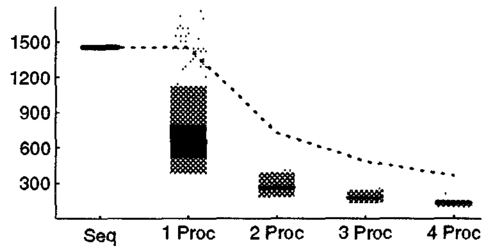
Hunecke, lexicographic Cyclic 6, graded rev. lexicogr.

Both examples terminated only with three and four processors. (Left example with four processors: 120–125 seconds.)

[3]Sum of the total degrees of the terms, refined by the lexicographic ordering.

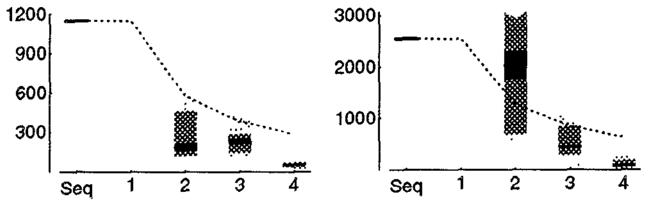
5.2.3 Chaotic Behavior

The main disadvantage of the search parallelism is its non-deterministic behavior. This can lead to runs which sometimes terminate after a few seconds and otherwise must be killed due to memory overflow after hours, with exactly the same input and configuration. However, this also opens the chance to receive results that could not be computed otherwise.



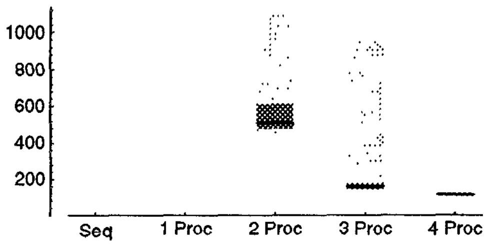
Cohn $\sqrt{2}$, graded reverse lexicographic

For this example, we obtained runtimes between 400 and 1800 seconds with one processor. The majority of the runs are much faster than in the sequential case. With four processors, the bulk of the runtimes lie in the 100–150 second range.



Left: Cassou graded reverse lexicographic
Right: Auxiliary in Cyclic 7 (Arnborg-Lazard), lexicographic

In both examples, none of the 10 runs terminated with one processor. We observe a chaotic behavior, decreasing with the number of processors, and large super-linear speedups. That means, the strategy is much improved with increasing number of processors (i.e., with the corresponding setup). With four processors we obtain timings of 26–76 seconds for the left example and of 40–200 seconds for the right example.



Canny 2, lexicographic

This example is representative for many others. The improved selection strategy enables us to compute, in a few minutes, Gröbner Bases that we could not obtain before. Furthermore, the runtimes are much more stable with four processors.

6 Conclusion

We have demonstrated that large super-linear speedups are possible with a parallel Gröbner Basis installation on standard, low-cost workstations with a few processors.

Our parallel Gröbner Basis algorithm retains the well-understood basic structure of the traditional implementation and is compatible with all important selection strategies and deletion criteria. Its practical behavior, ranging from pure work parallelism to search parallelism of adjustable width, can be influenced by setting just a few parameters. It also admits a refined selection strategy with new opportunities for better control.

On the system side, implementation of the algorithm design is supported by a single programming construct (the thread group) with proven utility. A parallel environment need not require a radical break with existing coding techniques. Just a few new system concepts (threads with thread groups and fork / join) make it possible in practice to realize new algorithmic concepts even for code that is to be executed on uniprocessors.

We see as one of the main problems for the immediate future to gain more experience with the new flexibility. The search parallel algorithm should be further tuned to produce results in a more stable and more easily reproducible way. As a result of our experimentation, we are confident that our master implementation is a solid first step towards making parallelism the method of choice for large Gröbner Basis computations.

References

[1] ATTARDI, G., AND TRAVERSO, C. A strategy-accurate parallel Buchberger algorithm. In Hong [16], pp. 12–21.

[2] BECKER, T., AND WEISPFENNING, V. *Gröbner Bases, a Computational Approach to Commutative Algebra.* Springer-Verlag, 1993.

[3] BUBECK, T., HILLER, M., KÜCHLIN, W., AND ROSENSTIEL, W. Distributed symbolic computation with DTS. In *IRREGULAR'95*, A. Ferreira and J. Rolim, Eds., vol. 980 of *LNCS*, pp. 231–248.

[4] BUCHBERGER, COLLINS, ENCARNACIÓN, HONG, JOHNSON, KRANDICK, LOOS, MANDACHE, NEUBACHER, AND VIELHABER. SACLIB User's Guide, 1993. On-line software documentation.

[5] BUCHBERGER, B. Basic features and development of the Critical-Pair/Completion procedure. In *RTA'85*, J.-P. Jouannaud, Ed., vol. 202 of *LNCS*, pp. 1–45.

[6] BUCHBERGER, B. The parallel L-Machine for symbolic computation. In *Eurocal'85*, B. F. Caviness, Ed., vol. 204 of *LNCS*, pp. 541–542.

[7] BUCHBERGER, B. The parallelization of critical-pair / completion procedures on the L-Machine. TR 87-12, RISC, Linz, Austria, 1987.

[8] BUCHBERGER, B., AND JEBELEAN, T. Parallel rational arithmetic for computer algebra systems: Motivating experiments. TR 92-29, RISC, Linz, Austria, 1992.

[9] BÜNDGEN, R., GÖBEL, M., AND KÜCHLIN, W. A fine-grained parallel completion procedure. In *ISSAC'94*, J. von zur Gathen and M. Giesbrecht, Eds., ACM Press, pp. 269–277.

[10] BÜNDGEN, R., GÖBEL, M., AND KÜCHLIN, W. Multi-threaded AC term rewriting. In Hong [16], pp. 84–93.

[11] CHAKRABARTI, S., AND YELICK, K. Implementing an irregular application on a distributed memory multiprocessor. In *POPP'93*, ACM Press, pp. 169–178.

[12] CZAPOR, S. R. A heuristic selection strategy for lexicographic Gröbner bases? In Watt [23], pp. 39–48.

[13] FAUGÈRE, J. Parallelization of Gröbner basis. In Hong [16], pp. 124–132.

[14] GEBAUER, R., AND MÖLLER, H. On an installation of Buchberger's algorithm. *JSC 6*, 2 & 3 (1988), 275–286.

[15] GIOVINI, A., MORA, T., NIESI, G., ROBBIANO, L., AND TRAVERSO, C. "One sugar cube, please." In Watt [23], pp. 49–54.

[16] HONG, H., Ed. *First Intl. Symp. Parallel Symbolic Computation PASCO'94* vol. 5 of *Lecture Notes Series in Computing*, World Scientific.

[17] KÜCHLIN, W. PARSAC-2: A parallel SAC-2 based on threads. In *AAECC-8*, S Sakata, Ed., vol. 508 of *LNCS*, pp. 341–353.

[18] KÜCHLIN, W. The S-threads environment for parallel symbolic computation. In *CAP'90*, R. Zippel, Ed., vol. 584 of *LNCS*, pp. 1–18.

[19] KÜCHLIN, W. PARSAC-2: Parallel computer algebra on the desk-top. In *Computer Algebra in Science and Engineering*, J. Fleischer, J. Grabmeier, F. Hehl, and W. Küchlin, Eds., World Scientific, 1995, pp. 24–43.

[20] POSSO. Polynomial systems library. `ftp: posso.dm.unipi.it`.

[21] SAWADA, H., TERASAKI, S., AND AIBA, A. Parallel computation of Gröbner Bases on distributed memory machines. *JSC 18*, 3 (1994), 207–222.

[22] VIDAL, J.-P. The computation of Gröbner bases on a shared memory multiprocessor. In *DISCO'90*, A. Miola, Ed., vol. 429 of *LNCS*, pp. 81–90.

[23] WATT, S. M., Ed. *ISSAC'91* (Bonn, Germany, July 1991), ACM Press.

[24] WINDSTEIGER, W., AND BUCHBERGER, B. *GRÖBNER: A Library for computing Gröbner Bases based on SACLIB. Manual for Version 2.0*, 1993.

BEATRICE AMRHEIN is a Research Associate at the Universität Tübingen in Tübingen, Germany. Dr. Amrhein holds a Dr. sc. math. (Ph.D.) degree from the Swiss Federal Institute of Technology (ETH). Her current research is in the areas of parallel and symbolic computation, and combinatory logic.

OLIVER GLOOR is a Research Associate at the Universität Tübingen in Tübingen, Germany. Dr. Gloor holds a Dr. sc. math. (Ph.D.) degree from the Swiss Federal Institute of Technology (ETH). His current research is in the areas of parallel and symbolic computation, and combinatory logic and the use of computer algebra systems in education.

WOLFGANG W. KÜCHLIN is an Associate Professor at the Universität Tübingen in Tübingen, Germany. Prof. Küchlin holds a Dipl.-Inform. (M.Sc.) degree from the Universität Karlsruhe, Germany and a Dr. sc. techn. (Ph.D.) degree from the Swiss Federal Institute of Technology (ETH). His current research is in the areas of parallel computation and and symbolic computation.