A Parallel Gröbner Factorizer

Hans-Gert Gräbe , Wolfgang Lassner
Institut für Informatik, Universität Leipzig, Germany

June 24, 1994

Abstract

We report on some experience with a parallel version of the Gröbner basis algorithm with factorization, implemented in the REDUCE package CALI [4]. It is based on a coarse grain parallel master-slave model with distributed memory. This model was realized on an HP workstation cluster both with a disk remote connection based on (ordinary) REDUCE [9] and the special PVM-based parallel REDUCE version of H. Melenk and W. Neun [7].

Our considerations focus on a detailed study of the practical time behaviour of the parallelized improved Gröbner factorization algorithm [5]. For well splitting examples, where the number of intermediate subproblems is large compared to the number of parallel processes available on the system (only for such examples this approach makes sense), we've got almost always a good load balance. Since even for the relative slow disk remote connection the results are encouraging, we conclude that with a fast and stable communication hard- and software one will obtain a serious speed up on such problems compared to the serial implementation.

1 Introduction

In a precedent paper [5] we discussed a new (serial) implementation of the Gröbner factorization algorithm. Factorized Gröbner bases are up to now the most powerful approach to the determination of the solution set of a well splitting system of polynomial equations over an algebraically closed field. A first detailed study appeared in [6]. The main advantage of the new implementation is a careful study of the interconnections between different problems thus allowing cancellation of superfluous computational branches in an early stage of the computation.

The Gröbner factorization algorithm is well suited for a parallelization on almost all parallel architectures, since it produces a lot of mutually independent subproblems that may be treated in parallel by slave sessions started at different nodes. Several such experiments are reported in the literature, see e.g. [2] or [8], but we know of no report about a detailed study of the parallelized implementation.

Such a detailed study is desirable since parallel symbolic algorithms have some special peculiarities compared to parallel numerical applications. The most important special feature is the intermediate problem's growth, that usually can't be predicted by the input data. Thus an efficient automatic load balancing needs much more effort than for most numerical problems. Secondly, (at least advanced) symbolic problems usually rest on a deep knowledge of basic symbolic techniques as simplification, polynomial arithmetic over different base rings, factorization etc. These techniques must be available at each node. Thus parallel symbolic implementations usually need either a (true) shared memory or powerful local nodes to support the necessary software. Moreover, due to the high costs of excessive symbolic communication, usually the latter architecture supports only coarse grain parallelized algorithms.

Below we report on our experience with a parallel implementation of the improved Gröbner factorizer algorithm on a workstation cluster, consisting of 8 HP 9000/735-99 of different memory size ($32 - 144\,MB$), connected through FDDI each other and to a CONVEX server, supporting a

(NFS-)shared file system. It is part of the CONVEX Meta Series installation at the University of Leipzig. We hope to continue these experiments in a near future on a massive parallel system with a better communication hard- and software. Nevertheless especially the PVM-based implementation leads already to a certain speedup compared to the serial version of the same algorithm.

In our experiments a master process manages, updates, and distributes the problems according to the rules described in [5], whereas several slave processes work out the corresponding subtasks.

Since each slave process needs a full REDUCE session, each node can host at most $2 \cdots 3$ of them. Thus the number of slave processes, executable in parallel, is restricted by the hardware.

The problem list management developped in [5] is well suited for such a restricted parallelism, even if, as in our experiments, the software doesn't support the interruption of executing tasks, that turned out to be superfluous, and the master manages only the list of subproblems not yet distributed to the slaves.

Let's now describe the main algorithmic idea of our approach :

- During a *preprocessing* the master interreduces the set B of input polynomials and tries to factor each $f \in B$. If f factors, he replaces B by a set of new problems, one for each factor of f, updates the side conditions, and applies the preprocessing recursively. This ends up with a list of interreduced problems with non factoring base elements. This preprocessing is currently not parallelized.

  Then each of these ideal bases is prepared for further computations, e.g. computing the list of critical pairs. They are collected in a problem list managed by the master, sorted by "importance".

- The master distributes the most important problems to the slaves, one per process. The slaves proceed with the usual Gröbner basis algorithm on the list of pairs not yet processed, trying to factor each reduced (non zero) S-polynomial before it will be added to the polynomial list. If it factors, they split up the problem into as many subproblems as there are (different) factors, add each of the factors to the corresponding subproblem, update the pair list and the side conditions, and return the new problems to the master.

- If the pair list is exhausted, the slave extracts the minimal Gröbner basis of the subproblem and continues with tail reductions. This may cause some of the reduced polynomials to factor anew. Hence he applies the preprocessing once more. If the result is stable he returns it to the master as a result. Otherwise the new problems are returned to the master.

- The master manages both the problem and the result lists according to the rules described in [5].

Obviously this procedure terminates and returns a list of Gröbner bases with the desired properties. For details of this part of the implementation we refer to [5].

2 The Communication Frames

Communication is the heart of any parallelization. Together with the high software requirements of symbolic computations this suggests two approaches :

1. Resting on a standard symbolic package use its built in communication facilities developped for serial purposes to design a parallel communication frame.

   In a first approach we used the standard REDUCE distribution [9] together with the package CALI [4] and its (symbolic mode) disk input/output facilities for communication between different processes.

   Such an approach has the following advantages :

(a) We can use the full power of the diagnostic software supplied with REDUCE for the analysis of the implementation.

(b) The communication protocol can be designed by the user and hence restricted to a necessary minimum.

The main disadvantages are :

(a) The communication frame is slow. Surprisingly enough, it is not as bad as we expected in advance.

(b) Due to the restricted communication protocol there is no error handling. This requires a very secure task handling. After the resolution of certain read/write conflicts caused by the underlying NFS system we had almost no trouble even with large computations.

2. Extend standard symbolic software with an interface for RPC (remote procedure calls) through one of the "classical" communication levels.

   For such an approach we used the original symbolic mode interface of CALI with the PVM-based REDUCE version of H.Melenk and W. Neun, [7], developped at the Zuse center in Berlin. We kindly acknowledge the possibility to use this experimental software and the support provided by the authors installing the system.

   This approach has the following advantages :

   (a) The communication itself is faster and more stable than in the first approach.

   (b) One can use a standard communication RPC interface provided by the software.

   Without doubt, this is the direction in which parallel symbolic software should be developped further. Nevertheless also this approach has some (temporary) disadvantages :

   (a) Transferring arguments of a RPC the corresponding expression tree is searched for repeated entries to minimize the necessary data transfer. This causes a relative great overhead.

   (b) The REDUCE-PVM interface doesn't (yet) support error handling.

   (c) The software doesn't support detailed diagnostics.

Both versions use the RLISP remote interface proposed by Melenk and Neun in [7]. (A subset of) its main features are the following procedures :

remote_process(*initfile*)
: start a *slave process*, load the REDUCE kernel and the necessary environment due to the supplied *initfile*.

remote_call(*function, parameters, slave*)
: start a new *task* on a specified slave executing

```
apply(function, parameters)
```

in the slave's REDUCE shell.

remote_inquire(*task*)
: test, whether the specified task was finished.

remote_receive(*task*)
: get the result of the task's request and close it.

remote_kill(*process*)
: terminate the slave process.

3 The Run Time Experiments and Conclusions

3.1 Preparing the run time experiment

Both communication frames follow the general RPC scenario as e.g. described in [11] :

- A remote request is formulated in the master's REDUCE shell.
- The remote request is submitted to the kernel process.
- The master's kernel transfers the request to the slave's kernel.
- The slave's kernel delivers the request to the slave's REDUCE shell.

Since the internal diagnostic software watches only for the REDUCE shells it is not easy to estimate the true communication overhead.

On the other hand, dividing the communication overhead into two parts, the software and the kernel parts, the internal and the external one, allows us even for a slow interactive medium as the disk remote connection to get an impression about the potential possibilities of our algorithmic ideas in a fast communication frame. Indeed, the internal (to REDUCE) overhead is almost constant and doesn't depend on the externa. The only difference (and deformation of the calculation's graph) comes from the different time that slaves have to wait for calling the "slow" master and vice versa.

Since awakening sleeping processes is not supported by our communication frames, unbusy slaves and master are sleeping a constant time (of 1s., calling `sleep` from inside REDUCE). These (in)activities can be watched with the REDUCE diagnostic software (counting the calls to `sleep`). This allows to predict in a very precise manner a fast communication scenario of the same calculations. We will discuss this below in more detail.

Note that slaves are sleeping during the (not yet parallelized) preprocessing.

To develop a good approximation of the true picture of the communication we have to collect different CPU times and real times. Here *CPU time* measures only activities of the corresponding process, not of derived kernel activities. This applies to both the PVM-based communication and the disk remote communication frames. Using the *qualtime* package of REDUCE we can count even the CPU time spent in single procedures of both the master's and the slave's (the latter only in the disk remote version) REDUCE shells. *Real time* is the real time difference between two events and thus depends not only on the kernel processes responsible for the external communication, but also on the general load factor of the machine. Comparing CPU and real time of a serial symbolic implementation for reference we've got a factor in the range $2 \ldots 5$ between both timings.

The master's CPU time mt may be divided essentially in two parts,

> the call/receive time cr spent during `remote_call` and `remote_receive`, e.g. allocating memory for the transferred data (excluding the file management part of the kernel process),
>
> and the master's proper contribution $prog$ to the Gröbner bases computation (preprocessing, list management etc.).

The same applies to the slave's CPU time st, that can be divided (and accessed) into two parts in the same way :

> the slave's contribution $prog$ to the Gröbner bases computation
>
> and the slave's receive/send time cr.

3.2 The Examples

We tested our implementation on several big polynomial systems arizing in the computation of automorphism groups of (complex) Lie algebras given by their structure constants. For a d-dimensional Lie algebra we obtain this way $d \times \binom{d}{2}$ quadratic equations in d^2 variables (a^i_j) constituting the matrix of the automorphism, see [3, cor. to thm. 2]. Due to the structure theory of these automorphism groups (CALI's *varopt* suggests the same) we consider the corresponding interreduced system of polynomials with respect to the pure lexicographic term order, where variables with greater $|i-j|$ are counted first.[1] In table 1 we collected all results, not only those with $det(a^i_j) \neq 0$.

As result we obtain usually a prime component through the unit matrix and several other components lying on the hypersurface $det(a^i_j) = 0$.

Our examples arose from the following Lie algebras

1. **a5.x** : the 5-dimensional Lie algebra $a5_x$ in [10].
2. **o4** : the 6-dimensional Lie algebra $o_4 = so_3 \oplus so_3$.
3. **heat** : the 6-dimensional Lie symmetry algebra of the heat equation, cf. [1, p. 178].

In table 1 we collected some characteristic data of these problems and the serial execution (CPU)-time with REDUCE 3.5 and CALI in sec. on an HP-9000/735.

example	vars	eqns	subsolutions	time	dimension of the solutions
a5.37	25	46	18	122	7 5x6 10x5 2x4 (2x5 with $det \neq 0$)
a5.39	25	46	6	73	8 2x7 3x6 (2x6 with $det \neq 0$)
a5.40	25	46	2	83	6 0
o4	36	90	9	1650	4x6 (2 with $det \neq 0$) 4x3 0
heat	36	90	2	911	6 0

Table 1 : The examples

3.3 The Disk Remote Version

Below in table 2 we collected our experimental results on the given examples, i.e.

the total serial (real) time *tst*,

the sum of the *prog* part contributions of the master and all slaves,

the number of tasks, i.e. subproblems sent to the slaves,

the master's mean call/receive communication (CPU) time per task,

the master's call/receive (CPU) time *cr*,

the master's CPU time *mt*,

the minimun and maximum slave's CPU time st_{min} and st_{max}

and for comparison the (master's) total (real) time *tpt* of the parallel computation.

These timings were obtained with 8 slaves on 8 nodes. We did also experiments with other configurations. The results are similar.

All times are given in sec.

[1] For *heat* we took the deglex. term order.

example	tst	sum of progs	# tasks	$\frac{cr}{task}$	cr	mt	st_{min}	st_{max}	tpt
a5.37	293	106.99	166	0.15	28	49.82	11.16	16.79	450
a5.39	191	112.56	243	0.12	37	53.94	11.27	18.51	575
a5.39	191	116.71	256	0.14	37	55.62	11.71	18.19	528
a5.40	244	79.55	143	0.20	33	46.14	6.92	19.25	349
a5.40	244	82.86	144	0.20	33	46.22	7.70	15.31	394
heat	2081	809.83	82	0.70	62	86.99	12.30	485.71	1101
o4	2169	1166.94	760	0.30	289	580.76	111.26	168.33	2032

Table 2 : Experimental results with the disk remote version. The global picture.

Comparing the second and the last columns of our table we see that the timings obtained by the parallel disk remote implementation don't differ as much from the (well tuned) serial implementation as we were feared in advance. They report even a slight speedup for the larger examples. Some examples we were running several times to see how far the output depends on the temporary load situation of the cluster.

Although we reported in [5] a great influence of the strategy choosing next subproblems on the run time of the Gröbner factorizer the total effort of both the serial and the parallel versions (using the same strategy) for the computation of the Gröbner bases should be comparable. This yields an independent criterion for the adequateness of our scenario : the *prog* values of the slaves and the master should more or less sum up to the serial CPU time given in table 1. These sums are collected in the third column.

The mean call/receive communication time per task gives an impression about the average range of a subproblem that had to be transferred between the master and the corresponding slave process. Of course, the size of the problem varies not only between different examples but also between different pieces of the same computation. Nevertheless it turned out, that the average value of $\frac{cr}{task}$ for each single slave doesn't differ much from the average value over the whole computation. This is a first indication for a good load (self)balance. This means that cr is a good approximation for the total transfer amount (given in *transfer units* of about $0.8\,MB$, comparing the entries in the column $\frac{cr}{task}$ with the real disk file size of the transferred requests).

A second indication for such a good balance is the small ratio between the maximal and minimal slave's CPU time in almost all examples.

Comparing CPU times we see that in all cases but *heat* the master's time dominates. For *heat* there are several "thick hunks" to be treated by some of the slaves thus forcing the master to wait for them.

In table 3 we collected for the master's CPU time

the percentage of the contribution of the preprocessing $\%\ pre$,

the percentage of the contribution of the call/receive part $\%\ cr$,

the number of `sleep` calls,

and for the slaves' CPU times

the range of the percentage of the slave's *prog* part,

and the range of the number of `sleep` calls.

These parameters give a good impression about the local distribution of the computations.

example	Master			Slaves	
	% pre	% cr	# sleep	% prog	# sleep
a5.37	3	57	10	71..84	123..329
a5.39	< 1	62	27	68..76	171..439
a5.39	< 1	66	11	72..80	122..403
a5.40	17	71	7	62..73	72..228
a5.40	17	70	3	61..74	270..371
heat	< 1	83	774	53..98	128..919
o4	30	50	64	74..81	86..105

Table 3 : Experimental results with the disk remote version. The local picture.

Let's give an interpretation of the number of `sleep` calls. In almost all examples the real bottleneck was the slow input/output of the master process. But, comparing master's and slave's CPU times, we conclude that even for a superfast communication the picture will not change much. The only difference will be the range of `sleep` calls to the slaves. The great dispersion in table 3 rests mainly on the circumstance whether the slave had to handle a lot of small problems (high sleeping rate standing in the master's queue to deliver the answer) or a smaller number of more serious problems. In a faster frame these unsimilarities should disappear.

Let's try to predict the real time needed for the same computations, but with a communication that is 10 times faster that our disk remote connection. With a load factor (real time vs. CPU time) of about 3 our timings yield a *real call/receive communication time* of about 10 s. per transfer unit. Altogether we get for the expected total parallel time $etpt = tpt - 9cr$ the values in table 4 (*o4* doesn't meet these assumptions, see table 1) :

example	etpt
a5.37	198
a5.39	217
a5.40	73
heat	543

Table 4 : Expected real run time with fast communication.

3.4 The PVM-based Version

For the PVM based version we had access only to the master's time. In table 5 we compared these timings with the corresponding disk remote values obtained earlier. We've got already a slight speed up for almost all our examples. The number of tasks necessary to complete a problem differs heavily between both versions (and also between runs at different time of the same version). This is due to the sensitivity of the Gröbner factorizer to the subproblem selection strategy as reported in [5].

		PVM				disk remote			
example	tst	mt	cr	# tasks	tpt	mt	cr	# tasks	tpt
a5.37	293	25.40	3.95	186	236	49.82	28	166	450
a5.39	191	17.80	3.20	175	138	55.62	37	256	528
a5.40	244	21.12	5.38	184	105	46.22	33	144	394
heat	2081	24.05	6.38	82	1312	86.99	62	82	1101
o4	2169	437.57	32.16	816	2091	580.76	289	760	2032

Table 5 : PVM based vs. disk remote version.

Taking into account the sensitivity mentioned above one should agree that these timings fit very close into the expected run time scenario derived from the detailed consideration of the disk remote connection above. Denote, on the other hand, that the huge PVM-software providing the communication changes the load situation on the cluster drastically (even with one slave per node its CPU usage is seldom more that 30%), so that it needs some care to compare the timings in the table.

3.5 Conclusions

Due to the coarse granularity of our approach the number of available tasks is not very high and one cannot expect a good load balance if the number of slave processes is beyond that number. Nevertheless the results collected in table 2 – 5 show, that we obtain a good load balance and a good ratio between the communication overhead of the slaves and the real work they have to do.

We tried even larger examples than the *o4* and found that up to 24 slave processes (3 per node) are well suited for our cluster. Our approach gives an almost equal distribution to the slaves, provided the problem does well split into enough pieces. The smaller examples produced enough subproblems only for $5 \ldots 8$ slave processes. Hence the load balance in a given configuration of a medium range number ($5 \ldots 20$) of powerful (≥ 25 MB heap size) slave processes depends strongly on the examples. Our approach is very well suited for examples that really admit factorization. Many problems occuring in "real life" are of this kind. The performance will increase with a fast communication frame that doesn't influence the load factor so "heavily" as PVM does.

Some of the examples factored very well, thus producing a great amount of communication overhead, whereas other examples factored into a small number of large Gröbner basis computation pieces only. In the former class the slow communication became a real bottleneck and slaves had to wait a great amount of time. In the latter examples (e.g. in *heat*) the master was unbusy some time, thus indicating a good load balance for the slaves.

References

[1] G. W. Bluman, S. Kumei : Symmetries and Differential Equations. Springer, New York, 1989.

[2] R. Bradford : A parallelization of the Buchberger algorithm. In : Proc. ISSAC'90, Tokyo, ACM Press, New York 1990, 296.

[3] V. P. Gerdt, W. Lassner : Isomorphism verification for complex and real Lie algebras by Gröbner basis technique. In : Modern group analysis : Advanced analytical and computational methods in mathematical physics, (Ibragimov et al., eds.), Kluwer Acad. Publ. (1993), 245 - 254.

[4] H.-G. Gräbe : CALI – A REDUCE package for commutative algebra. Version 2.1., Oct. 1993. Available through the REDUCE library e.g. at `redlib@rand.org`.

[5] H.-G. Gräbe : Solving polynomial systems with factorized Gröbner bases. To appear.

[6] H. Melenk, H.-M. Möller, W. Neun : Symbolic solution of large chemical kinematics problems. *Impact of Computing in Science and Engineering* **1** (1989), 138 - 167.

[7] H. Melenk, W. Neun : RR – Parallel symbolic algorithmic support for PSL based REDUCE. Draft version, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Oct. 93.

[8] W. Neun, H. Melenk : Very large Gröbner basis calculations. In : Computer algebra and parallelism (ed. R. Zippel), LNCS 584 (1992), 89 - 99.

[9] REDUCE 3.5, Rand Corp., Santa Monica, Oct. 1993.

[10] J.Patera *et al* : Invariants of real low dimension Lie algebras. *J. Math. Phys.* **17** (1976), 986 - 993.

[11] Tanenbaum : Moderne Betriebssysteme. Carl Hanser Verlag München, 1994.