Parallel Computer Algebra on the Desk-Top

Beatrice Amrhein **Wolfgang Kuechlin**

University of Tuebingen
W. Schickard Institute for Informatics
Sand 13, D-72076 Tuebingen, Germany [1]

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

http://www-sr.informatik.uni-tuebingen.de/

Abstract

We report on the development of PARSAC-2, a library of parallel algebraic algorithms designed specifically for networks of multiprocessor workstations. PARSAC-2 is built upon the S-threads system environment for multi-threaded symbolic computation.

S-threads provides virtual parallelism by mapping thousands of very light-weight processes onto the processors of a workstation. It is currently being extended with network functionality, so that heavy-weight processes can be mapped across the network while preserving the S-threads interface.

The current goal of algorithm development in PARSAC is the construction of a parallel polynomial equation solver using Groebner-Bases. We report on the design of a strategy-compliant parallel Groebner-Basis computation with factorization.

Introduction

Symbolic computation is a high-level computational task which makes it comparatively complex and slow. However, it is increasingly applied in science and engineering [FGHK94] and any significant increase in the speed of symbolic computation will expand the range of possible applications.

In Computer Algebra we are dealing with dynamic data structures like arbitrary precision integers or arbitrary degree polynomials. Control flow and run time are extremely data-dependent. For example, the thread of control changes dramatically when a polynomial coefficient is found to be another polynomial instead of an integer. Therefore, it is hard to use hardware like vector units or floating point coprocessors. Fortunately, however, many symbolic and algebraic computations are inherently parallel, and parallel computers are coming into widespread use.

Now, supercomputers with hundreds of processors are certainly very attractive machines, but they are relatively hard to program. It is difficult to balance the computation load, and to organize the communication, between so many processors. It is also often very hard to break

[1] Our research is currently supported by Deutsche Forschungsgemeinschaft under grants Ku 966/1 and Ku 966/2.

the computation down into small enough tasks that will fit into the small local memories of each processor.

Moderately parallel machines which are easier to program are now taking an upswing. Shared memory multiprocessors have already started to appear on the desk-top with up to 4 processors. And a distributed memory component for such desk-top machines can be provided by the network. As workstation networks are ubiquitous, desk-top parallel computation with workstation networks can be used everywhere.

On the other hand, since modern workstations nowadays are parallel multiprocessors, the internal parallelization of computer algebra systems is absolutely necessary. An important challenge of parallel Computer Algebra is now to exploit *at the same time* shared-memory and distributed-memory parallelism available by workstation networks.

Therefore, our target architecture for a parallel Computer Algebra system is a network of shared-memory workstations, as networks of parallel workstations available now and everywhere, and they will be still available in the future.

In parallelizing Computer Algebra systems, we can distinguish three major approaches.

Several standard (sequential) Computer Algebra system kernels are connected by a communication system.

An existing sequential kernel or library (whose source is available) is modified by inserting parallel calls.

A new parallel Computer Algebra system is built from scratch using a new parallel language.

The first approach is most coarse grained and is advantageous in a homogeneous architecture, where all kernels are the same. This is certainly the easiest way to run an application in parallel. However, coarse-grain parallelization alone may leave a few subproblems which dominate the run-time. Only fine-grain parallelism can remove these bottlenecks. Moreover, we must be able to exploit the full power of our future desk top machines, which are starting to be parallel as well.

The second approach makes medium-grain parallelism accessible and is therefore used in multi-threaded environments. Here, we can exploit the additional fine-grained shared-memory parallelism in parallel workstations. It has the advantage of using proven code as a basis, and it saves memory by running only one kernel. The approach is limited by the fact that the parallel language constructs must be an extension of the language of the sequential system.

The third approach leaves complete freedom for expressing the parallel constructs. It has the potential of delivering an integrated, seamless parallel system. The main disadvantage is that it requires a substantial implementation effort.

Our approach is between the first and the second. The starting point was the library of algebraic algorithms of SAC-2. The original library algorithms are now step by step exchanged by their parallel equivalent. Some examples are root isolation, resultants, gcd computation, and polynomial and integer multiplication.

PARSAC-2

Work on PARSAC-2 began in 1989 [Küc90] based on the SAC-2 library and a Threads programming interface. Development began on a 12 processor SM Encore Multimax, first under Encore UMAX 4.3 with EPT Threads, and later under Mach with C Threads. It has since been ported to multiprocessor Solbourne SPARC servers under Solbourne OS/MP 4.1 and PCR Threads [WDH89], and most recently to multiprocessor Sun SPARCstations under Solaris 2.x and Solaris Threads. For a recent overview see [Küc95].

Parallel System Environment: S-threads

In the shared-memory case, main requirements upon the system environment are to provide (logically) unrestricted parallelism, to map the parallel tasks efficiently onto the processors, and to integrate the parallelism smoothly into the application programming environment above and into the operating system below.

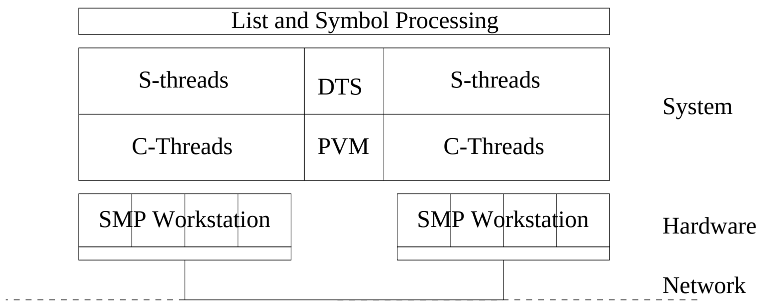

Figure: The S-threads System Environment

The S-threads system allows the algorithms to be executed concurrently as multiple threads. S-threads can be viewed as extensions of C Threads. They have a local list-processing environment, including a local heap fragment with local free cell list [Küc90, Küc91a, Küc91b].

An introduction to S-threads can be found in the reference manual [Küc92] for Release 1; since then, garbage collection support has been much improved [KN91] and the important *virtual threads* concept has been added [KW92].

The original library algorithms are now continuously exchanged by their parallel equivalent. Some examples are root isolation, resultants, gcd computation, and polynomial and integer multiplication. The clear separation into system environment and application provides for modularity and portability. Both parts are developed simultaneously, the S-thread system is enhanced with the needs of the application, and the application is used to test the system.

Distributed Threads System: DTS

Many communication environments for distributed memory have been proposed, but a standard is only slowly emerging. Distributed memory communication is often built upon the TCP/IP protocols accessible through UNIX sockets. Linda is a relatively high-level environment that provides a virtually shared *tuple space* for communication. On a lower level, PVM provides facilities for placing tasks, and passing messages, across an inhomogeneous net. PVM is already available on a wide variety of architectures. With PVM it is therefore possible to develop a distributed application on a workstation net and later to run the same code on a parallel supercomputer.

S-threads system was enhanced by the DTS network component on top of PVM. DTS [Bub93] stands for distributed threads system. It provides an asynchronous remote procedure call to worker nodes across the network in order to fork threads which carry copies of their parameters and do not need shared memory. Each node runs S-threads and may be a multiprocessor. It was implemented by Bubeck and Hiller and first tested with a parallel RSA cryptosystem and parallel polynomial resultant computations [Bub93, Hil93, BHK95]. Now, we have also first experiences with distributed Groebner Bases and Knuth Bendix completions.

In order to work towards a common goal, parallel tasks have to communicate. They have to be created *(forked)*, and their results have to be collected *(joined)* again. They may also have to communicate data and synchronize the access of common data structures.

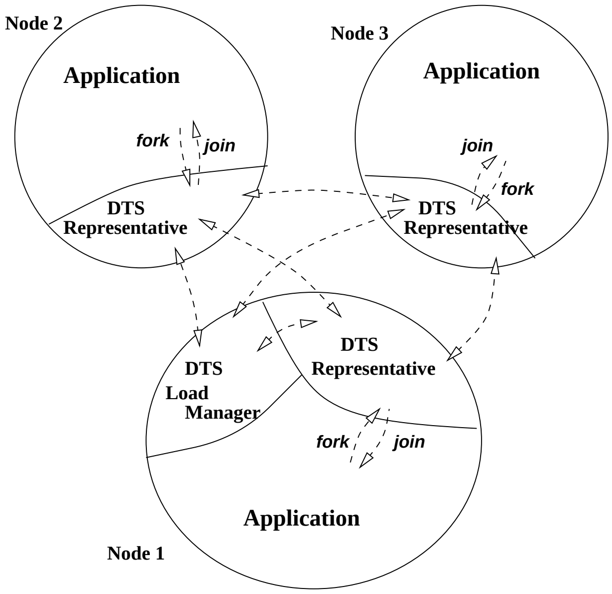

Figure: Architecture of DTS

The creation of these communication primitives must be high-level enough to be usable and easily understandable. But they also must be low-level enough to be efficient.

One more major issue is the placement of tasks across the network so that the system load is evenly balanced. DTS provides load balancing and error recovery. Up to now, this is done by a central manager. Current development is concerned with a decentralization of DTS and with a smooth integration into the S-threads system, which is used on each node.

In the future, the decision whether a fork has to be a high level DTS-fork or a lower level S-thread-fork will be postponed until runtime instead of hard coded into the program.

The PARSAC Philosophy

One of the most important problems in parallel Computer Algebra is the *grain-size* issue. The question is, how large are the pieces of logical parallelism which can be profitably executed with real parallelism on a given architecture.

Since Computer Algebra codes are so large, we cannot afford to rewrite them every time we move to a new machine. We must be able to write portable code which gives good speed-ups on a variety of machines under a changing number of processors and changing system load without any major change in the code. In addition, the system must neither be starved for speed-up on an empty machine nor drown in processes on a loaded machine.

A related problem is the *integrability* of code. We need to be able to build parallel libraries in the same way as we are building sequential libraries. However, code cannot be integrated into a larger system if it makes assumptions on the number of free processors or the number of processes and their grain-sizes. Therefore, in PARSAC-2 we concentrate on developing and evaluating the divide-and-conquer style of parallelization.

The System Environment delivers the real and the virtual parallelism and the scheduling of the user forks. Furthermore, the S-threads system provides for the integration with the underlying operating system. Lists with futures or tasks can be processed in the ordinary way; however, the computation will implicitly wait for the actual result to be delivered if the content of some special future is to be inspected.

An optimization which is of prime importance in practice is the lazy generation of futures, first developed as *Work Crews* in the context of Modula-2+.

The approach of PARSAC-2 is to generate a hierarchy of parallel tasks and to dynamically fuse small (virtual) threads into bigger (real) ones according to the machine load at run-time. In PARSAC-2, a lazy future is called a *virtual thread.*

What we want to achieve is a parallel library, which is as flexible as possible. On the other hand, the programming constructs must be clear and easy to apply so that it is possible to keep the overall view, which is not trivial and of prime importance in parallel programming.

Parallel Algebraic Equation Solving

Groebner Basis Completion

Many approaches have already been performed to parallelize Groebner Basis computation . The parallelization of the (non-factoring) Groebner Basis computation has attracted much attention, going back to the mid 1980's [Wat86, Buc87]. However, it has posed considerable difficulties in practice [Pon89]. Firstly, as Groebner Basis completion both creates new polynomials and deletes old polynomials, there is much demand for synchronization. Vidal [Vid90], Melenk and Neun [MN89] were among the first who used shared memory parallelism and achieved speed-ups in practice; their work was combined by Schwab [Sch92].

However, with the discovery of different criteria and better selection strategies, many of the previous approaches have to be re-evaluated.

We can distinguish four different approaches in parallelizing Groebner Basis computation.

Coarse grained Several processes compute different Groebner Bases according to different term orderings or different strategies. For example, the factoring Groebner Basis completion is of this type, as each worker computes some of the Groebner Bases.

Medium grained Several polynomials are selected for reduction and inserted in the basis concurrently. Afterwards, new critical pairs are built from all of these polynomials. One drawback of this method may be its high complexity, which makes it difficult to ensure correctness.

Fine grained The different polynomials in the basis act as individual workers and reduce concurrently each of the polynomials of the critical pair list. In this surrounding, we also speek of a *reduction pipe*.

Very fine grained This would be the case, when we even tried to proceed the reductions of the single polynomials in parallel (e.g. using parallel integer arithmetic).

Solving Systems of Algebraic Equations

In the context of system solving, one can generate very coarse-grain parallelism by a *factoring Groebner Basis algorithm.* If the Groebner Basis computation generates a polynomial which factors, we can split the computation into the concurrent computation of two or more Groebner Bases.

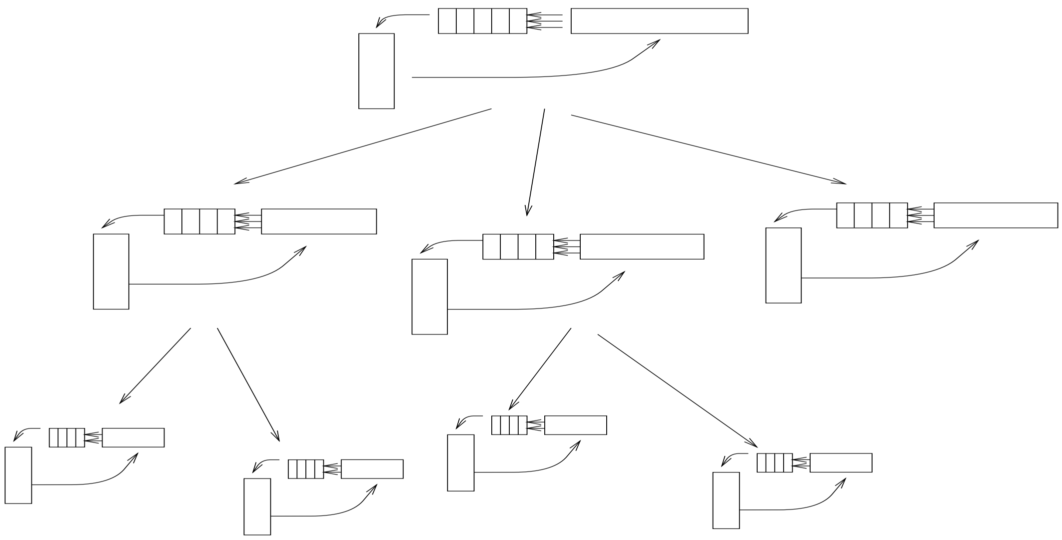

Figure: Groebner Factorizer

This method was already proposed by Czapor [Cza89] and by Melenk and Neun[NM92] and is already profitable sequentially because the resulting systems are much smaller (see also [GL94]).

In a parallel system solver we can discover parallelism on different levels. For example, we can apply a parallel *gcd* algorithm for making parallel irreducible tests, but also for factoring polynomials in parallel. When we have computed the Groebner Basis (with respect to any term-ordering), we apply basis conversion and perform parallel root isolation and back substitution.

Our approach towards a parallel Groebner Factorizer is a follows: we reduce some critical pairs (or s-polynomials) selected according to the *sugar strategy* [GMN+91], perform irreducibility tests and, in the positive case, we try to factor them concurrently on the same node. If a polynomial factors, we can distribute the different Groebner Basis computations over the network.

```
GB_Factor(basis, critical_Pairs , s_Poly_set)
while ( Length(s_Poly_set) > 0)
      Split s_Poly_set into s_Poly and (s_Poly_set \ s_Poly);
      dts_fork(dtsGB_Factor, basis, critical_Pairs, Term_order, s_Poly);
```

```
dtsGB_Factor(basis, critical_Pairs , s_Poly)
 if ( s_Poly is not equal to zero ) then
      insert s_poly into basis;
      build new pairs (with criteria);
      concurrently reduce the small pairs in the critical_Pairs list and try to factor them;

 while (critical_Pairs is not empty)
      apply criteria;
      find smallest s_Poly_set in critical_Pairs according to the termorder (or sugar-order);

        if Length(s_Poly_set) > 1 then      // this critical pair has factored
             GB_Factor(basis, critical_Pairs , s_Poly_set);      // split computation
        else
             insert the polynomial in s_Poly_set into the basis;
             build new pairs with criteria;
             concurrently reduce the small pairs in the critical_Pairs list and try to factor
             them;

 return (basis);
```

Summary

As parallel workstations will (soon) be found on every desk top, the internal parallelization of computer algebra systems becomes important.

In the future, using parallelism for applications will be made significantly easier as parallel systems provide more functionality and better support for the decomposition of tasks.

With parallelism, we will reach new levels of applications through more computational power and (equally important, especially in symbolic computation!) more memory.

The challenge we see is to exploit the parallelism that is provided by the most widespread machines as hardware support for Computer Algebra.

References

[BHK95] Tilmann Bubeck, Martin Hiller, and Wolfgang Küchlin. Distributed symbolic computation with DTS. In *Proc. IRREGULAR'95*, LNCS, 1995. (To appear).

[Bub93] Tilmann Bubeck. Eine Systemumgebung zum verteilten funktionalen Rechnen. Technical Report WSI-93-8, W.-Schickard-Institut für Informatik, Universität Tübingen, D-72076 Tübingen, 1993.

[Buc87] Bruno Buchberger. The parallelization of critical-pair / completion procedures on the L-Machine. Technical Report TR 87-12, RISC, Linz, Austria, 1987. (Also in: *Japanese Symp. Functional Programming*, pp. 54–61, Feb. 1987).

[Cza89] S. R. Czapor. Solving algebraic equations: Combining Buchberger's algorithm with multivariate factorization. *J. Symbolic Computation*, 7:49–53, 1989.

[DDF89] J. Della Dora and J. Fitch, editors. *Computer Algebra and Parallelism*. Computational Mathematics and Applications. Academic Press, London, 1989. (Proc. CAP'88, Grenoble, France, June 1988).

[FGHK94] J. Fleischer, J. Grabmeier, F. Hehl, and W. Küchlin, editors. *Computer Algebra in Science and Engineering*, Bielefeld, Germany, August 1994. World Scientific. (Proceedings of a ZiF workshop. To appear).

[GL94] Hans-Gert Gräbe and Wolfgang Lassner. A parallel Gröbner factorizer. In Hong [Hon94], pages 174–180.

[GMN+91] Alessandro Giovini, Teo Mora, Gianfranco Niesi, Lorenzo Robbiano, and Carlo Traverso. "one sugar cube, please" OR selection strategies in the Buchberger algorithm. In Watt [Wat91], pages 49–54.

[Hil93] Martin Hiller. Verteiltes symbolisches Rechnen mit PARSAC-2. Master's thesis, W.-Schickard-Institut für Informatik, Universität Tübingen, September 1993.

[Hon94] Hoon Hong, editor. *First Intl. Symp. Parallel Symbolic Computation PASCO'94*, volume 5 of *Lecture Notes Series in Computing*, Linz, Austria, September 1994. World Scientific.

[KN91] Wolfgang W. Küchlin and Nicholas J. Nevin. On multi-threaded list-processing and garbage collection. In *Proc. Third IEEE Symp. on Parallel and Distributed Processing*, pages 894–897, Dallas, TX, December 1991. IEEE Press.

[Küc90] Wolfgang W. Küchlin. PARSAC-2: A parallel SAC-2 based on threads. In Sakata [Sak90], pages 341–353.

[Küc91a] Wolfgang W. Küchlin. On the multi-threaded computation of integral polynomial greatest common divisors. In Watt [Wat91], pages 333–342. (Also OSU-CISRC-1/91-TR2).

[Küc91b] Wolfgang W. Küchlin. On the multi-threaded computation of modular polynomial greatest common divisors. In Zima [Zim91], pages 369–384. (Proc. 1st ACPC).

[Küc92] Wolfgang W. Küchlin. The S-threads environment for parallel symbolic computation. In Zippel [Zip92], pages 1–18. (Proc. CAP'90, Ithaca, NY, May 1990).

[Küc95] Wolfgang W. Küchlin. PARSAC-2: Parallel computer algebra on the desk-top. In Fleischer et al. [FGHK94]. (Proceedings of a ZiF workshop. To appear).

[KW92] Wolfgang W. Küchlin and Jeffrey A. Ward. Experiments with virtual C Threads. In *Proc. Fourth IEEE Symp. on Parallel and Distributed Processing*, pages 50–55, Dallas, TX, December 1992. IEEE Press.

[Mio90] A. Miola, editor. *Design and Implementation of Symbolic Computation Systems*, volume 429 of *LNCS*, Capri, Italy, April 1990. Springer-Verlag. (Int. Symp. DISCO'90).

[MN89] Herbert Melenk and Winfried Neun. Parallel polynomial operations in the large Buchberger algorithm. In Della Dora and Fitch [DDF89], pages 143–158. (Proc. CAP'88, Grenoble, France, June 1988).

[NM92] W. Neun and H. Melenk. Very large Groebner basis calculations. In Zippel [Zip92], pages 89–100. (Proc. CAP'90, Ithaca, NY, May 1990).

[Pon89] Carl G. Ponder. Evaluation of "performance enhancements" in algebraic manipulation systems. In Della Dora and Fitch [DDF89], pages 51–73. (Proc. CAP'88, Grenoble, France, June 1988).

[Sak90] S. Sakata, editor. volume 508 of *LNCS*, Tokyo, Japan, August 1990. Springer-Verlag.

[Sch92] Stephen A. Schwab. Extended parallelism in the Gröbner Basis algorithm. *Int. J. of Parallel Programming*, 21(1):39–66, 1992.

[Vid90] Jean-Philippe Vidal. The computation of Gröbner bases on a shared memory multiprocessor. In Miola [Mio90], pages 81–90. (Int. Symp. DISCO'90).

[Wat86] Stephen M. Watt. *Bounded Parallelism in Computer Algebra*. PhD thesis, University of Waterloo, Waterloo, Canada, 1986. Also Technical Report CS-86-12.

[Wat91] Stephen M. Watt, editor. *Proc. 1991 International Symposium on Symbolic and Algebraic Computation: ISSAC'91*, Bonn, Germany, July 1991. ACM Press.

[WDH89] Mark Weiser, Alan Demers, and Carl Hauser. The portable common runtime approach to interoperability. In *12th ACM SOSP*, pages 114–122, 1989.

[Zim91] H.P. Zima, editor. volume 591 of *LNCS*, Salzburg, Austria, October 1991. Springer-Verlag. (Proc. 1st ACPC).

[Zip92] Richard Zippel, editor. *Computer Algebra and Parallelism*, volume 584 of *LNCS*, Ithaca, NY, March 1992. Springer-Verlag. (Proc. CAP'90, Ithaca, NY, May 1990).