A New Library for Parallel Algebraic Computation*

Wolfgang Schreiner† Hoon Hong†

February 3, 1993

Abstract

We give an overview on PACLIB, a library for parallel algebraic computation on shared memory multiprocessors. PACLIB is essentially a package of C functions that provide the basic objects and methods of computer algebra in a parallel context. The PACLIB programming model supports concurrency, shared memory communication, non-determinism and speculative parallelism. The system is based on a heap management kernel with parallelized garbage collection that is portable among most UNIX machines. We present the successful application of PACLIB for the parallelization of several algebraic algorithms and discuss the achieved results.

1 Introduction

Scientific computing is a rich source of challenging problems such as the solution of systems of partial differential equations. Classical numerical methods operate with efficient finite-precision (floating point) arithmetic and thus quickly yield approximative solutions. However, often one is also interested in certain qualitative aspects like stability properties or the singularities of given systems [5]. In these cases, it is important to compute the *exact* solutions of the given problem, because numerical methods may (by the accumulation of approximation errors) yield qualitatively wrong answers.

Computer algebra is that branch of computer science that aims to provide exact solutions of scientific problems. Research results of this area are e.g. algorithms for symbolic integration, polynomial factorization or the exact solution of algebraic equations and inequalities [3]. All these algorithms have to operate with arbitrary precision arithmetic; they are therefore much more expensive with respect to time and space than the corresponding numerical methods. The parallelization of computer algebra algorithms is therefore of utmost importance in order to extend their application area and to contribute to the further development of scientific computing.

Thus the parallel computation group at RISC-Linz has started a project that pursues the development of PACLIB, a library of parallel algorithms based on the computer algebra library SACLIB [2]. PACLIB has been implemented in the C language on top of a runtime kernel that supports an efficient and high-level parallel programming model [7]. The PACLIB kernel [8] has been implemented on a 20 processor Sequent Symmetry, a MIMD computer with shared memory, but is in principle portable to any UNIX machine.

In this paper, we give an overview on the application of the PACLIB library. First, we sketch the design of its runtime kernel and describe the PACLIB programming model in short. The main section is dedicated to the demonstration of how an algebraic algorithm has been parallelized in PACLIB. An outlook on our further work concludes the paper.

*Funded by the FWF grant S5302-PHY "Parallel Symbolic Computation".

†Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria.

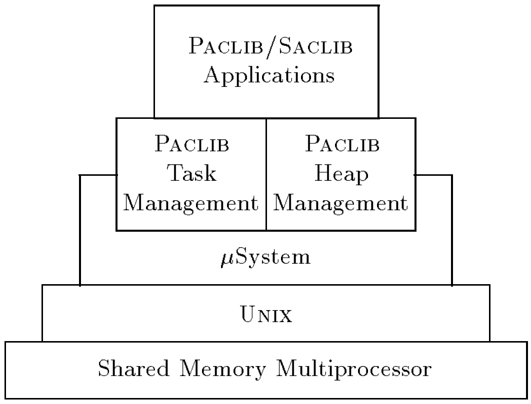

Fig. 1. *The* Paclib *Design*

2 The System Structure

Paclib has emerged from a combination and extension of freely available software packages (see Figure 1):

- Saclib [2] is a library of C functions that are based on a heap management kernel with automatic garbage collection. It provides all the fundamental objects and methods of computer algebra: arbitrary precision integer and rational arithmetic, polynomial arithmetic, linear algebra, polynomial gcd and resultant computation, polynomial factorization, real root calculation and algebraic number arithmetic.

- The μSystem [4] is a library of C functions that supports concurrency on shared memory multiprocessors and Unix workstations. The central concept of the μSystem is the *task* i.e. a light-weight parallel process that may communicate with other tasks via shared memory. The μSystem task management is very efficient and allows the utilization of rather *fine-grained parallelism*.

The Paclib library is built around a runtime kernel that provides the same heap interface as the Saclib kernel i.e. any sequential Saclib application will also run when compiled with the Paclib kernel. However, the internal structure of the Paclib kernel is far more complicated and consists of the following components (see Figure 2):

- **The Heap** that serves as the communication medium between Paclib tasks.

- **The Global Available List** `gavail` that contains all free heap cells.

- **The Ready Queue** that holds all active tasks ready for execution.

- **The Virtual Processors** that execute Paclib tasks.

- **The Local Available Lists** `lavail` from where a task can allocate new heap space.

When a task calls the Saclib function `comp` to construct a new list cell, a free cell is allocated from the local available list `lavail` of that processor that currently executes this task. If `lavail` runs out of free cells, the processor picks the first list in `gavail` as its new `lavail`. If `gavail` becomes empty, all processors perform in parallel a global garbage collection using a *mark and sweep* scheme: in the "mark" phase, all task stacks are concurrently scanned for accessible heap cells; in the "sweep" phase, different heap portions are concurrently scanned for unused cells and `gavail` is reconstructed.

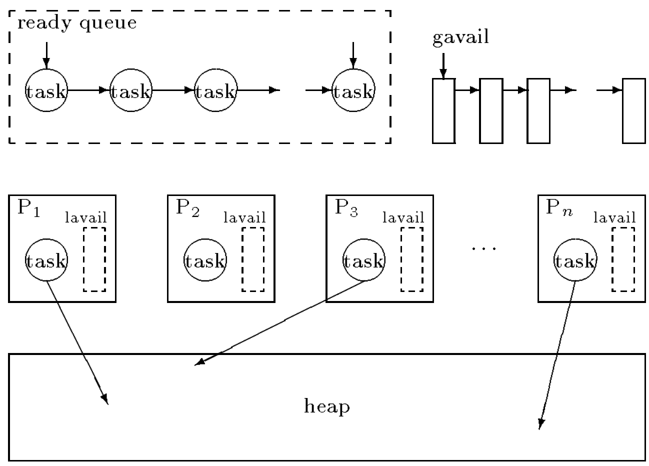

Fig. 2. *The* Paclib *Kernel*

On the one hand, this two-level management scheme allows tasks to allocate cells without synchronization overhead (since the cells are taken from `lavail`). On the other hand, it ensures that no heap space is wasted (since new free lists are taken from `gavail` on demand). The Paclib kernel has been implemented on a Sequent Symmetry computer with 20 processors but is portable to all shared memory multiprocessors supported by the μSystem package (including most Unix workstations).

3 The Programming Model

The Paclib programming model is basically a functional one; each user function can be executed as a concurrent task. Communication and synchronization are mainly based on task results and their availability, respectively. In the following, we give the (abstract) syntax of some of the most important Paclib constructs. The function

task = `pacStart(`*fun*, *args*`)`

creates a new task executing *fun(args)* in parallel with the current task. The arguments *args* are references to Saclib objects in the shared heap. The new task terminates and deallocates its resources as soon as the *fun* call returns with a result r. *task* is a reference to the new task that may be used by other tasks for retrieving r. For instance,

result = `pacWait(`*tptr*, *tasks*`)`

non-deterministically delivers the result r_i of one of the denoted *tasks*. The reference of the delivering task t_i is stored at the location *tptr* and r_i is returned as *result*. If all tasks are still running, `pacWait` temporarily blocks the current task.

A typical task management scheme that applies the non-deterministic features of `pacWait` looks as follows:

```
while (exists(work))
  w, work = split(work)
  t = pacStart(worker, w)
  tasks = comp(t, tasks)
while (!isnil(tasks))
  r = pacWaitList(&t, tasks)
  result = combine(r, result)
  tasks = remove(t, tasks)
return(result)
```

The first loop iteratively splits off some part `w` from the total `work`. It starts a new task `t` executing `worker(w)` and stores `t` in `tlist`. When no more work is left, the second loop iteratively waits for the result `r` of any task and combines `r` to the total `result`. The order in which results are delivered is not fixed in advance; the receiving loop is only blocked if all remaining tasks are still executing. By this mechanism, a significant amount of time can be saved if the combination process is costly and the runtimes of the tasks vary very much (which is often the case in algebraic algorithms).

However, in some applications, by the result of some task the result of all the remaining tasks may have become irrelevant. In this case, the function

`pacStop(`*tasks*`)`

can be applied that prematurely aborts the execution of all denoted *tasks*. In combination with the non-deterministic `pacWait`, this function can be used to express *speculative parallelism*.

Tasks may also communicate via *streams* i.e. buffered (and potentially unbounded) communication channels. Exactly one writer task may use `pacPut` to put values into the stream while a set of reader tasks may use `pacGet` to retrieve these values from there. Streams are a functional concept similar to linked lists: each reader has the same view of the stream and will receive the same sequence of values independently of the other readers. *Buffer interfaces* allow a more general access to streams.

Due to lack of space, we cannot describe the PACLIB programming model in more detail. There are many variations of the above functions that facilitate the most frequent forms of application. For a complete definition of all constructs and their concrete syntax, see [7].

The current implementation of the PACLIB programming model is based on shared memory: all task arguments are references to objects in the shared heap, i.e. data structures are not duplicated even if they are passed as arguments to new tasks. Provided that no PACLIB task destructively updates its input arguments (almost all SACLIB functions stick to this convention), this is a safe and very efficient scheme. However, the programming model itself is totally independent of the underlying architecture and can be realized on distributed memory architectures, too.

4 An Application Example

As an example for the application of PACLIB, we describe in some more detail the design of a parallel algorithm for the exact solution of linear equation systems with arbitrarily large integer coefficients. This is one of the basic problems in computer algebra, its solution is e.g. required for the manipulation of multi-variate polynomials. Handling such polynomials is of importance in several practical applications such as geometric modelling [6].

The problem can be formalized as follows. Let A be a regular $n \times n$ matrix over $\mathbb{Z}$ and b a vector of length n over $\mathbb{Z}$. We want to find the vector x of length n over $\mathbb{Q}$ such that

$$A * x = b.$$

Please note that A and b contain arbitrary integer numbers whose size is not limited by the word length of any computer; these numbers therefore have to be represented by sequences of computer words. Furthermore, the result vector x shall consist of rational numbers that are the *exact* solutions of the system; each rational therefore has to be represented by a pair of integers (the nominator and the denominator, respectively).

Since we have to perform exact arithmetic, the size of the involved integers steadily increases during the computation and arithmetic becomes more and more time consuming.

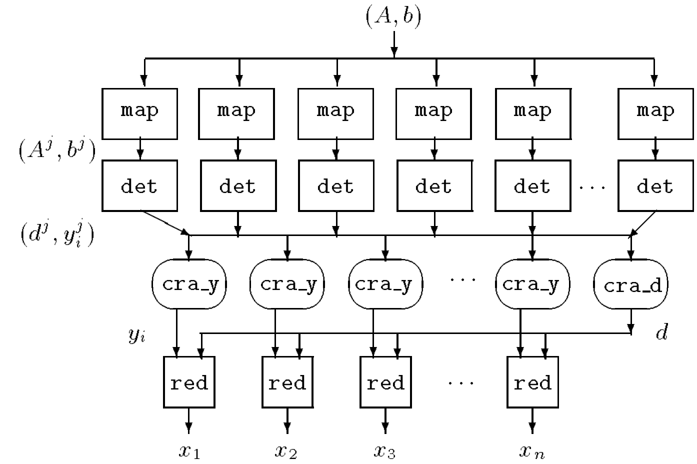

FIG. 3. *The Modular Algorithm*

Thus the complexity of the standard Gaussian Elimination algorithm is $O(n^5 l^2)$ where l is the maximum length of the entries in A and b [13].

The most efficient sequential approach for solving the equation system is illustrated in Figure 3: We apply Cramer's Theorem which says that the solutions can be computed by $x_i = y_i/d$ where $d = \det(A)$ and $y_i = \det(A_i)$ (A_i is A with the i-th column replaced by b). Thus we transform the problem of solving an equation system into a problem of determinant computation that can be efficiently solved using *modular arithmetic*:

We take k prime numbers[1] p_j and map the given system (A, b) over $\mathbb{Z}$ into k systems (A^j, b^j) over the finite fields $\mathbb{Z}_{p_j}$. Provided that the p_j fit into single computer words, arithmetic can in these fields performed in constant time (since all elements are bounded by p_j). Then we compute the determinants $d^j = \det(A^j)$ and $y_i^j = \det(A_i^j)$.

Since the systems (A^j, b^j) are *homomorphic images* of (A, b) with respect to determinant computation, we can apply the *Chinese remaindering algorithm* [1] to determine the original determinants y_i and d. Finally, we compute $x_i = y_i/d$ and reduce the result to normal form applying the *Euclidean algorithm* for the computation of the greatest common divisor. The total complexity of the algorithm is then $O(n^3 l^2 + n^4 l)$.

The basic idea for the parallelization of this algorithms is as follows;

- **Mapping:** The images (A^j, b^j) can be computed in parallel.
- **Determinants:** The determinants d^j and y_i^j can be computed in parallel.
- **Chinese Remaindering:** $n + 1$ tasks may compute d and each y_i in parallel.
- **Reduction:** The n reductions $x_i = y_i/d$ can be performed in parallel.

In the PACLIB implementation of the algorithm, there are actually the following tasks:

- A set of `det` tasks that perform the mapping and the determinant computation.
- One `cra_d` task that computes d from the d^j.
- n `cra_y` tasks that compute the y_i from the y_i^j and finally perform the reduction y_i/d.

[1] A lower bound for k can be determined from the entries of (A, b).

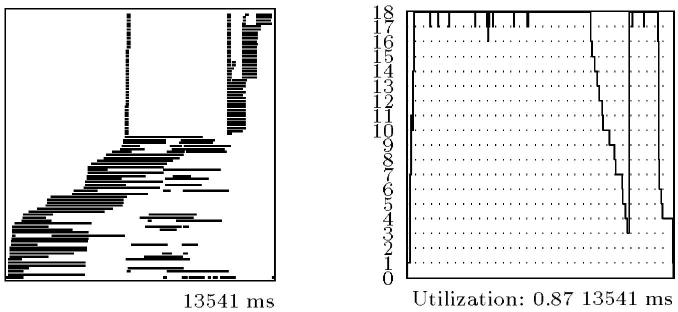

FIG. 4. *Visualization and Utilization*

The PACLIB program first creates the required number of `det` tasks and stores their handles in a variable `tasks`. Then the single `cra_d` task is started that takes `tasks` as argument and computes d from the results of the `det` tasks. In the meanwhile, also n `cra_y` tasks have been started with `tasks` and the handle of the `cra_d` task as arguments. The `cra_y` tasks compute the various y_i from the results of the `det` tasks, receive d from the `cra_d` task and perform the final reduction y_i/d.

Figure 4 displays the behavior of the algorithm for an equation system of dimension 40 using 18 processors. The left picture was created by the tool `pacgraph` that visualizes the profiling data that were generated during the run of a PACLIB program. Each horizontal line represents one task and shows in the horizontal extension the times during which the task was scheduled for execution. The right picture was made by the program `pacutil` and shows the utilization of the processors during the program run.

These pictures show that the program performed quite well for the given equation system. During most of the time, all processors were satisfied with work. There was only one essential synchronization point, when the `cra_y` tasks had to wait for the result of some `gauss` task. Figure 5 shows the speedups that could be achieved (compared to the sequential SACLIB implementation of the modular method) for three different equation systems of dimension 5, 10 and 40, respectively.

It turns out that the maximum speedup is almost 16 for the large system but bounded by the dimension n of the smaller systems (due to the fact that there are only n `cra_y` processes). For handling these cases, we have developed more efficient variants of the algorithm whose description is beyond the scope of this paper. Furthermore, there are several technical details that are essential for an efficient implementation. For a detailed discussion of these issues, see [10] and [12].

5 Contents of the Library

Among its algorithms, the PACLIB library currently contains parallel methods for

- **Gröbner Bases Computation:** Buchberger's Gröbner Bases Algorithm is one of the most fundamental and most powerful problem solving methods in computer algebra. An important application of this algorithm is e.g. the exact solution of systems of multivariate polynomial equations. The PACLIB implementation of this algorithm uses a bidirectional pipeline of tasks connected by streams through which polynomials "flow" in both directions [11]. With this implementation (which essentially relies on the PACLIB support of non-determinism), a maximum speedup of 10 could be achieved on our machine.

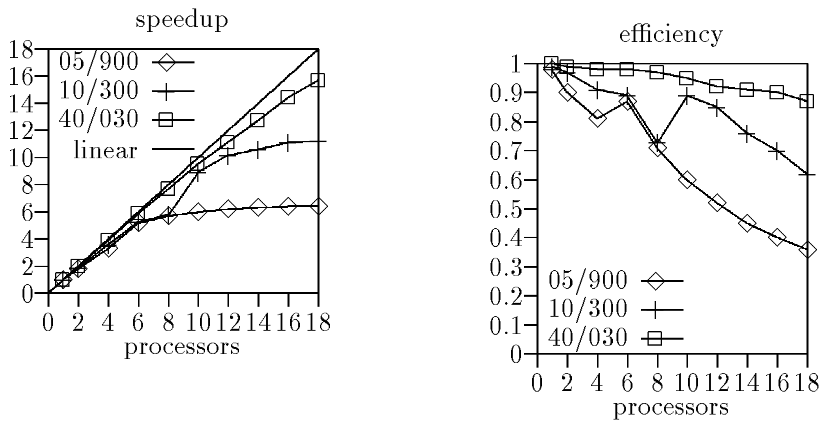

FIG. 5. *Speedup and Efficiency*

- **Resultant Computation:** The resultant of two polynomials is the determinant of a matrix that is in a certain way constructed from the coefficients of the polynomials. Resultants are used for solving non-linear systems of polynomial equations and inequalities. The PACLIB implementation of the algorithm computes resultants of multivariate integer polynomials using a modular approach: The resultants are concurrently computed in several homomorphic images. The computation in each image is also parallelized in a divide and conquer fashion by reducing the computation of a resultant of degree n to the computation of several resultants of degree $n-1$. A maximum speedup of 10 could be achieved.

- **Polynomial Factorization:** The classic method for polynomial factorization is Berlekamp's algorithm. This method computes the complete factorization of a polynomial by computing the greatest common divisors of certain other polynomials. The basic idea for parallelization is to compute these greatest common divisors in parallel. In PACLIB, a variant of the algorithm for the factorization of univariate polynomials over finite prime fields has been implemented. A maximum speedup of 12 could be achieved for certain examples.

- **P-adic Arithmetic:** p-adic arithmetic can be used for the efficient evaluation of arithmetic expressions over big rational numbers. Instead of a direct evaluation, the expression is mapped into several simple domains. In these domains, the required computations can be efficiently solved using some sort of "truncated" arithmetic. The result in the original domain is then constructed from the results in the simple domains. The PACLIB implementation of p-adic arithmetic achieves a speedup of 3 for medium-sized examples.

A more detailed description of most of these methods can be found in [3]. Currently, the library is constantly extended by new algorithms.

6 Conclusions and Further Work

We presented a new library PACLIB for parallel algebraic computation on shared-memory multiprocessors. PACLIB is based on an efficient runtime kernel for heap and task management that allows to utilize rather fine-grained parallelism. A high-level parallel programming model is used to implement parallel algorithms on the basis of the C language.

The dynamic behavior of PACLIB programs can be visualized by various tools for further optimization. Several important algebraic algorithms have been parallelized in this library and we will continue our efforts in this direction. Furthermore, we work on a formal verification of the runtime kernel [9] and on the the introduction of new features such as task priorities and virtual tasks. A compiler translating a functional programming language into PACLIB code is currently under development as well as an interactive X11-based visualization environment.

7 Acknowledgements

PACLIB has been developed by the RISC-LINZ parallel computation group on an initiative of its leader H. Hong. W. Schreiner performed the detailed design and the implementation of the runtime kernel. A. Neubacher, K. Siegl, H.-W. Loidl and T. Jebelean helped in many discussions to clarify the main concepts. The algorithms described in this paper were designed and implemented by V. Stahl, W. Schreiner, K. Siegl, C. Limongelli, M. Encarnacion, M. Minimair and H. Hong. This work was funded by the FWF grant S5302-PHY "Parallel Symbolic Computation".

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] B. Buchberger, G. Collins, M. Encarnation, H. Hong, J. Johnson, W. Krandick, R. Loos, and A. Neubacher, *A SACLIB Primer*, Tech. Rep. 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[3] B. BUCHBERGER, G. E. COLLINS, R. LOOS, AND R. ALBRECHT, eds., *Computer Algebra — Symbolic and Algebraic Computation*, Springer, Vienna, New York, 1982.

[4] P. A. Buhr and R. A. Stroobosscher, *The μSystem: Providing Light-weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX*, Software — Practice and Experience, 20 (1990), pp. 929–964.

[5] V. G. Ganzha, E. V. Vorozhtsov, and J. A. Hulzen, *A New Symbolic-Numeric Approach to Stability Analysis of Difference Schemes*, in ISSAC 92, International Symposium on Symbolic and Algebraic Computation, Berkeley, California, July 27–29, 1992, ACM Press.

[6] C. M. Hoffman, *Implicit Curves and Surfaces in Computer Aided Geometric Design*, Tech. Rep. CER-92-002, Department of Computer Science, Purdue University, 1992.

[7] H. Hong, W. Schreiner, A. Neubacher, K. Siegl, H.-W. Loidl, T. Jebelean, and P. Zettler, *PACLIB User Manual*, Tech. Rep. 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992. Also: Technical Report 92-9, ACPC Technical Report Series, Austrian Center for Parallel Computation, July 1992.

[8] W. Schreiner, *The Design of the PACLIB Kernel*, Tech. Rep. 92-33, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[9] W. Schreiner, *The Correctness of the PACLIB Kernel — A Case Study in Parallel Program Verification by Temporal Logic*, tech. rep., RISC-Linz, Johannes Kepler University, Linz, Austria, 1993. To appear.

[10] W. Schreiner and V. Stahl, *The Exact Solution of Linear Equation Systems on a Shared Memory Multiprocessor*, in Submitted to the PARLE 93, Munich, Germany, June 14–18, 1993.

[11] K. Siegl, *Parallelizing Algorithms for Symbolic Computation Using* $||MAPLE||$, in Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, May 19-21, 1993. To appear.

[12] V. Stahl, *Solving a System of Linear Equations with Modular Arithmetic on a MIMD Computer*, Tech. Rep. 92-62, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[13] F. Winkler, *Computer Algebra I*, Tech. Rep. 88-88, RISC-Linz, Johannes Kepler University, Linz, Austria, 1988.