The Design of the PACLIB Kernel for Parallel Algebraic Computation*

Wolfgang Schreiner and Hoon Hong

schreine@risc.uni-linz.ac.at hhong@risc.uni-linz.ac.at
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Abstract. This paper describes the runtime kernel of Paclib, a new system for parallel algebraic computation on shared memory computers. Paclib has been developed as a professional tool for the simple design and efficient implementation of parallel algorithms in computer algebra and related areas. It provides concurrency, shared memory communication, non-determinism, speculative parallelism, streams and a parallelized garbage collection. We explain the main design decisions as motivated by the special demands of algebraic computation and give several benchmarks that demonstrate the performance of the system. Paclib has been implemented on a Sequent Symmetry multiprocessor and is portable to other shared memory machines and workstations.

1 Introduction

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 [6]. All these algorithms operate with arbitrary precision arithmetic; they are therefore more expensive with respect to time and space than the corresponding numerical methods. However, they always provide correct solutions and are not influenced by the stability properties of the given problem.

The parallel computation group at RISC-Linz has started a project that pursues the development of Paclib, a new system for parallel algebraic computation on shared memory multiprocessors [20, 21]. Our goal has been to develop a professional tool for the simple and efficient implementation of parallel algorithms in computer algebra and related scientific areas. Paclib is a combination and extension of two free software packages:

- Saclib [5] is a library of C functions that is based on a list processing kernel with automatic garbage collection and supports all fundamental objects and methods of computer algebra: arbitrary precision integer and rational arithmetic, polynomial arithmetic, linear algebra, polynomial gcd and resultant

* Supported by the Austrian Science Foundation (FWF) grant S5302-PHY "Parallel Symbolic Computation".

computation, polynomial factorization, real root calculation and algebraic number arithmetic. Several algebraic applications have been implemented in this library such as the Partial Cylindric Algebraic Decomposition Method [10] and the Gröbner Bases Algorithm [4].
– μSYSTEM [7] is a library that supports concurrency on shared memory multiprocessors and UNIX workstations. The μSYSTEM kernel distributes *tasks* (light-weight processes) among *virtual processors* which are UNIX processes scheduled by the operating system. On a multi-processor, virtual processors are therefore executed by multiple hardware processors truly in parallel. The μSYSTEM task management is quite efficient and allows the utilization of rather *fine-grained parallelism*.

We applied and considerably extended the mechanisms of the μSYSTEM to develop a suitable parallel programming interface for SACLIB. In this paper, we will explain the design of the runtime kernel of this system that in particular supports the following features:

- **Concurrent Tasks:** Any function can be executed as a separate task; the scheduling of tasks and the balancing of load among processors is performed automatically.
- **Data Sharing:** All heap structures can be shared among tasks; arguments and results are passed by reference without any duplication of data.
- **Non-Determinism:** A task may simultaneously wait for the results of various other tasks. The further computation depends on which task delivers its result first.
- **Speculative Parallelism:** If by the result of one task the result of another task is not required any more, the superfluous task may be explicitly aborted.
- **Streams:** Tasks may be connected by streams of data and work in an overlapping fashion (pipelining).
- **Parallel Garbage Collection:** The automatic memory management is based on a mark and sweep scheme collector where both phases are efficiently parallelized.

The PACLIB kernel has been implemented on a SEQUENT SYMMETRY shared memory computer with 20 processors under the Dynix/ptx version of UNIX. The system is portable to any machine supported by the μSYSTEM package (a port to a multi-processor SILICON GRAPHICS is on the way). The PACLIB programming model is explained in large detail in [13]; a comprehensive description of the PACLIB kernel is given in [19]. The development of PACLIB applications is illustrated in [22].

2 Design Requirements

Systems for numerical computation in general do not meet the demands of algebraic applications. The main reason is that numerical programs operate on elementary objects (floating point numbers) and regular structures (vectors and matrices) whose size does not change during the computation. The memory demand of a task is therefore known in advance and its (relative) runtime can be predicted very precisely. Numerical applications can be naturally parallelized using the SPMD model of computation [29].

On the contrary, the objects of algebraic computation are symbolic expressions (arbitrary precision integers and rationals, polynomials, sets of polynomials, ...) that are dynamically allocated on a heap. These objects tend to grow very much in the course of the computation, which also increases the runtime of the operating tasks. The dynamic behavior of algebraic algorithms concerning time and memory is therefore very complex and often unpredictable even for known input values.

For parallelizing such algorithms, the SPMD model of parallelization is not suitable. We need the possibility to create concurrent tasks asynchronously and depending on the actual state of the computation. The task creation overhead should be very small since algebraic algorithms can be often parallelized at a medium- or fine-grained level only. Furthermore, flexible synchronization and communication constructs are required to minimize the runtime dependencies between tasks; in particular we need non-deterministic constructs that allow to select the first available result of a set of tasks.

By the dynamic nature of the basic objects, algebraic applications are at least as strongly bound by memory as by time. A garbage collector is required that reclaims no more used objects. In a parallel environment, the issue of garbage collection is even more important, since p processors in general allocate heap space at a rate that is p times higher than a single processor (provided that the total memory size is the same). In order to preserve the ratio of garbage collection time to the actual computation time, the garbage collector itself must be efficiently parallelized.

The high memory consumption also suggests that a distributed memory model of computation may be not appropriate. In such a model, all arguments of a remote task must be explicitly copied into the heap of the executing processor and the results must be copied into the heap of any requesting processor. Eventually, many duplicate structures will exist in the system and even within one heap (since initially shared substructures will be duplicated). The size of solvable problems is therefore drastically smaller than on a single processor machine with the same amount of memory. A global address space that allows the sharing of objects between processors is therefore preferred[2].

From the above discussion, we derive the following demands that we consider essential for systems for parallel algebraic computation:

- Dynamic task creation with low runtime overhead.
- Flexible synchronization constructs that support non-determinism.
- A shared heap with a global address space and parallel garbage collection.

[2] Currently, such a global address space is only efficiently supported by computers with few (≤ 32) processors. However, new *virtual shared memory* architectures promise to fulfill this demand also for massively parallel machines [17].

In the following sections, we will describe our approach to meet these demands and will give the reasons that led to the current design.

3 The Overall Structure

The PACLIB kernel consists of the following main components (depicted in Figure 1):

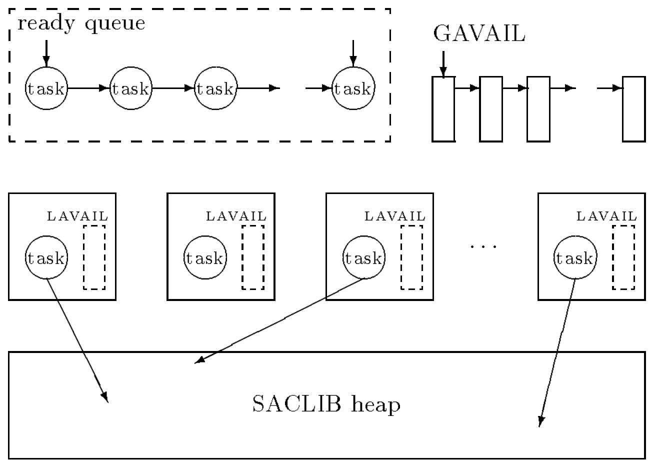

Fig. 1. The System Structure

1. **Ready Queue:** This queue contains all tasks that are *active* but not *running*. The μSYSTEM scheduler selects tasks for execution from the head of this queue; tasks that are preemptively descheduled are inserted at the tail.
2. **Shared Heap:** The SACLIB heap is an array of fixed-size list cells and serves as the only communication medium between tasks. There is no communication bottleneck since all PACLIB tasks may read all heap cells simultaneously.
3. **Virtual Processors:** Each virtual processor is a UNIX process that exclusively owns a local list `LAVAIL` of free heap cells. Each task executed on this processor allocates new heap cells from this list.
4. **Global Available List:** `GAVAIL` denotes a global list of free lists (of fixed but arbitrary length) in which the unused heap cells are linked together. Processors whose local lists run out of free cells receive their new `LAVAIL` from `GAVAIL`.

The next sections describe the task and the memory management in more detail.

4 Task Management

4.1 Functional Model

Algebraic algorithms are mostly based on purely mathematical functions that are entirely defined by their argument/result behavior. The PACLIB model of parallelism reflects this view. A PACLIB task receives certain SACLIB objects as input arguments and returns a SACLIB object as its result. This result may be retrieved later by a handle to this task.

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

creates a new task that executes *fun(args)* and can be referenced via the handle *task*. *task* is a first-order SACLIB object i.e. it can passed to other functions or tasks or stored in a SACLIB object. The result of this task (which is the value returned by *fun*) may be retrieved (arbitrarily often) by any task that knows *task*.

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

returns the result *value* of one of the *tasks* and writes the handle of the delivering task into the location *tptr*. If all denoted tasks are still active, `pacWait` blocks until one task terminates and then delivers the result of this task. Of course, this non-deterministic construct also supports the deterministic subcase of waiting for the result of a single task.

It may happen that by the result of some task the results of the remaining tasks have become irrelevant (speculative parallelism). In this case, the superfluous tasks should be aborted such that they do not consume any more processor time and memory resources.

`pacStop(`*tasks*`)`

terminates the execution of all denoted *tasks* and of all subtasks created by these tasks. If some task has already delivered its result, it is not affected any more. `pacStop` can be considered as an annotation that does not influence the correctness of a program (modulo termination) but only its operational behavior.

What is the physical interpretation of a task handle? One possibility is to implement it as a *task descriptor* i.e. as a reference to the workspace. Before a task stops execution, it stores its result there for the later retrieval by other tasks. However, this approach has severe disadvantages: either the task descriptor is not deallocated after the corresponding task has terminated and the workspace is never reclaimed. The other possibility (implemented in the μSYSTEM) is to deallocate the task descriptor when the result is received by another task. But then the abstract view of task handles as promises for task results goes lost, since the result of a task may be retrieved only once.

Since we wanted to preserve our abstract view of tasks, we decided to split the description of a task into two separate entities: a *task descriptor* and a *result* *descriptor*. *task* is then a reference to the result descriptor which is a SACLIB list that essentially contains three items: a semaphore for mutual exclusion, a flag that signals whether the result is available and either the result value itself or a queue of the descriptors of those tasks that are waiting for the result.

For each new task, a result descriptor and a task descriptor are allocated and mutually linked. Actually, the task executes a function `pacShell` that calls *fun* with the corresponding arguments *args*. `pacShell` stores the function result in the result descriptor and awakes all tasks blocked on this descriptor. Finally, `pacShell` deallocates its task descriptor and the task terminates.

The large task descriptor therefore occupies space only as long as the task is actually alive. After the termination of the task, only the small result descriptor is left by which other tasks can receive the result. Since this descriptor is a SACLIB object, it is subject to garbage collection and is (together with the result) reclaimed when no task has the *task* handle any more. Hence, task results do not occupy memory longer than necessary.

4.2 Stream Model

There are several parallel algebraic algorithms (e.g. for *critical pair completion* [6]) for which the pure argument/result model is not adequate. These algorithms can be better described in terms of tasks that possess an internal state. Such a task receives a sequence of input objects, processes each object according to its internal state (which is changed by the computation) and forwards a sequence of results to other tasks.

Therefore PACLIB supports an additional form of task communication by *streams*: a stream is a buffered communication channel by which a task can transmit a sequence of values to another task (or even to a set of other tasks). A stream can have three states:

1. *empty*: no element is available in the stream.
2. $\langle$*first, rest*$\rangle$: the stream is a pair of an element *first* and another stream *rest*.
3. *closed*: no element will ever become available in the stream.

Usually, an *empty* stream s_0 is created and passed as an argument to two new tasks. One task (the *producer*) changes the state of the stream from *empty* to $\langle v_0, s_1 \rangle$ where v_0 is the value to be transmitted and s_1 is a new and *empty* stream. The other task (the *consumer*) receives v_0 and s_1 that can be used for the transmission of another value. If the consumer tries to receive a value while s_i is still in the *empty* state, the task is blocked.

In this fashion, a sequence of streams $s_0, s_1, \ldots, s_n$ is generated that transmit the values $v_0, v_1, \ldots, v_{n-1}$. The producer finally sets the state of s_n to *closed*, which terminates the communication. Any task that has a stream handle s_i may receive the associated v_i and that v_i may be extracted from s_i arbitrarily often.

stream = `pacOpen()`

creates an *empty* stream and return its handle.

`pacClose(`*streams*`)`

sets the state of all *streams* to *closed*.

`pacPut(`*stream*`, `*value*`)`

sets the state of the *stream* to $\langle value,\ new \rangle$ where *new* is a new and empty stream. Then *stream* is replaced by *new*, i.e. any further application of *stream* will actually refer to *new*. Elements may be retrieved from a stream by

done `= pacGet(`*sptr*`, `*vptr*`, `*rptr*`, `*streams*`)`

If all *streams* are *closed*, `pacGet` returns `FALSE`. If some streams are *open* but all of them are *empty*, `pacGet` blocks the current task until the state of some stream has changed. If some stream s has the state $\langle v,\ r \rangle$, s is written into the location *sptr*, v into the location *vptr* and the rest stream r into the location *rptr*. `pacGet` then returns `TRUE`.

The implementation of streams is entirely based on the concepts introduced in the previous section. A stream is just a PACLIB result descriptor *without* an associated task descriptor. The result descriptor receives by `pacPut` a heap cell $\langle v, r \rangle$ where v is the value to be transmitted and r is a reference to a new result descriptor. `pacClosed` writes a `NIL` reference into the result descriptor, which terminates the stream. `pacGet` is implemented by the same algorithm as `pacWait`.

The definition of streams as result descriptors (i.e. synchronization points) without task descriptors (i.e. separate threads of control) again reflects the motivation for the introduction of this concept. The effect of a stream could be also simulated by a sequence of tasks in the following way: each task returns a result cell that is composed of an element and of a reference to a subtask computing the next cell. Actually, the concept of streams is orthogonal to that of function results i.e. the one can be based on the other.

4.3 Timings

The time overhead imposed by the PACLIB kernel is negligible i.e. a sequential program runs as fast as when linked with the SACLIB kernel. We now give several benchmarks that characterize the overall efficiency of the system. All timings were performed on a SEQUENT SYMMETRY with 128 MB main memory and 20 processors i80386 running at a clock rate of 20 MHz. A function call takes on this machine roughly 8 μs (including the passing of arguments on the stack). The creation of a corresponding PACLIB task by `pacStart` takes in average 230 μs i.e. approximately 30 times longer.

The time required for the reception of a task result by `pacWait` depends on whether the receiving task is blocked or not and how many results it waits for. We

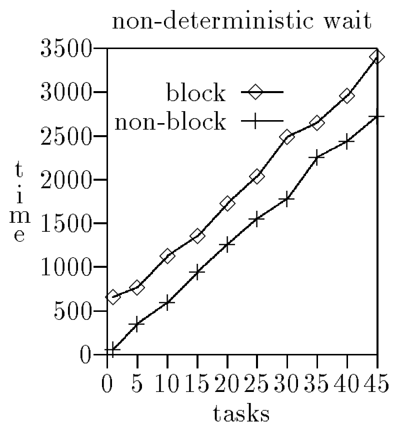

Fig. 2. Task Benchmarks

performed two benchmarks measuring the time that a task spends in `pacWait` with respect to the number of tasks m that it is waiting for (provided that the delivering task is the last one in the list). In the first test, the receiving task finds the result of the last task already available and is therefore not blocked. In the other test, none of the results is available yet; the current task is blocked and will be awaked by the last task in the list.

The results depicted in the left picture of Figure 2 show that the time t required for the delivery of a result is linear in the number of tasks m that are waited for. Roughly, we have $t = m * 60 + b$ μs where $b = 0$, if the receiving task is not blocked, and $b = 600$, if the task is blocked and awaked[3]. Waiting for a single task therefore costs 60 μs minimum.

In order to estimate the minimum granularities for Paclib applications, we ran the following benchmark (see the right picture in Figure 2): a main task started 100 subtasks (that were calibrated to execute for a certain amount of time) and received the corresponding results deterministically in sequence. The time from the start of the first task until the reception of the last result was measured. This test therefore represented the worst case by including most of the synchronization overhead into a single sequential task.

Granularities up to 10 ms were to small to to outweigh the parallelization overhead. This shows that just considering the task creation time of 24 μs gives a too optimistic picture by neglecting the times for context switches, reception of results and other effects. However, for tasks running 100 ms, almost linear speedups were achieved that could be not significantly improved by larger granularities.

[3] Actually, these 600 μs include 260 μs cost for two context switches, since the benchmark was executed on a single virtual processor.

5 Memory Management

5.1 The Garbage Collector

The PACLIB memory subsystem is transparent with respect to parallelism, i.e. it provides the same interface as the sequential SACLIB. The operation `COMP(h,t)` allocates a new heap cell and fills the head field with `h` and the tail field with `t`. The new cell is allocated from the local available list `LAVAIL` of that processor that currently executes the task. If `LAVAIL` runs out of free cells, the processor picks the first available list in `GAVAIL` as its new `LAVAIL`. If `GAVAIL` is also empty, a global garbage collection is triggered. This two-level memory management scheme of PACLIB is a pragmatic compromise between two extremes that both have significant disadvantages:

1. **Global Available List Only:** In order to keep `GAVAIL` consistent, it must not be simultaneously updated by different processors. Without local available lists, each call of `COMP` would require a semaphore operation that would considerably increase the runtime overhead of the function. Moreover, due to the serial access to `GAVAIL`, this shared resource would represent a significant bottleneck for parallelization.
2. **Local Available Lists Only:** Without a global available list, there arises the danger of wasting heap space. Some processor might consume cells from its local available list `LAVAIL` faster than the other processors. It would then trigger garbage collection even if there were still plenty of free cells available (since these cells were attached to the local available lists of other processors).

In order to trigger garbage collection, the processor puts the currently executed task back into the ready queue and interrupts by a signal the execution of the other virtual processors (actually UNIX processes). After all processors have acknowledged the interruption, garbage collection takes place in two phases:

1. **Mark:** All cells that can be referenced by any task are marked. Since all user tasks are descheduled, all heap references are on the stacks of these tasks. Each virtual processor picks a task descriptor, scans the associated stack and marks all heap cells that can be reached.
2. **Sweep:** All unmarked cells are reclaimed to construct a new list `GAVAIL`. Each processor scans a different portion of the heap, unmarks the marked cells and reclaims the unmarked cell. When a new free list of the desired length is built, the list is linked into `GAVAIL`.

The work of the sweep phase is well balanced among all processors since they operate on different heap sections of the same size. The balance of load in the mark phase depends on the number of available tasks and the number of cells that can be reached by each task. The worst case is that of a single task (since its stack is scanned by a single processor). However, if we assume that there are substantially more tasks than processors, also this phase is performed in parallel.

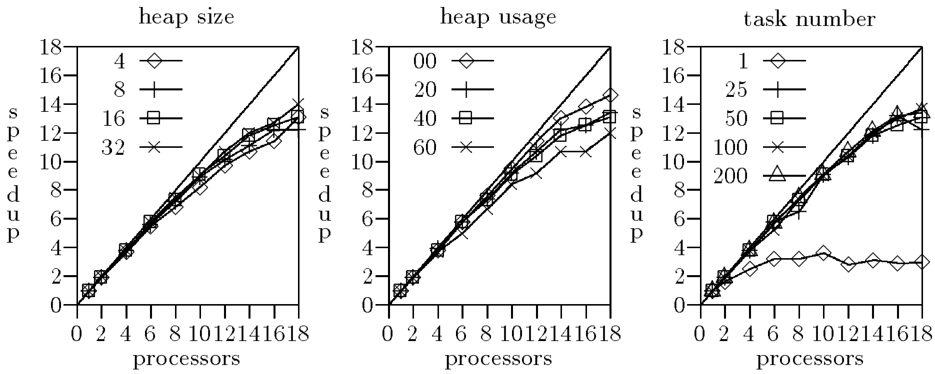

Fig. 3. Garbage Collection

5.2 Timings

We ran a set of benchmarks that measured the speedups that could be achieved for the garbage collector by starting a number of dummy tasks that evenly allocated heap cells. These programs tested the influence of three different parameters (see Figure 3):

- **Heap Size:** The larger the heap is, the longer the sweep phase takes (where all virtual processors can work independently in parallel). However, the total runtime differences are not very significant.
- **Heap Load:** The more heap cells are in use, the more time the mark phase takes. For an almost empty heap, a speedup of more than 14 could be achieved compared to a speedup of 12 for a heap with 60% cells in use.
- **Task Number:** The more tasks are active, the better the load balancing in the mark phase becomes. For a single task, only a speedup of 3 could be achieved; in all other cases a maximum speedup of 13 was possible.

Summarizing, in most cases the speedup is linear up to 10 processors and achieves a maximum of 14 with 18 processors. Each processor reclaims 1.5 MB of heap memory per second which is identical to the sequential SACLIB garbage collection rate. For typical applications, the garbage collection overhead is less than 15% of the total execution time.

6 An Application Example

As an example for the application of PACLIB, we sketch the implementation of a parallel algorithm for the exact solution of linear equation systems. 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 $A * x = b$. A and b may 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 has to be represented by a pair of integers (the nominator and the denominator, respectively).

Since we have to perform exact computations, the size of the involved integers steadily increases and arithmetic becomes more and more time-consuming. Thus the complexity of the Gaussian Elimination algorithm is $O(n^5 l^2)$ where l is the maximum length of the entries in A and b [28]. A more efficient approach is based on Cramer's Theorem saying that each x_i can be computed as $x_i = y_i/d$ where $d = \det(A)$ and $y_i = \det(A_i)$ (A_i is A with the i-th column replaced by b). Thus we may transform the problem of solving an equation system into the problem of computing determinants which can be efficiently done by *modular arithmetic*:

We take k prime numbers[4] p_j and map the system (A, b) 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. Then we compute the determinant families $d^j = \det(A^j)$ and $y_i^j = \det(A_i^j)$ and apply the *Chinese remaindering algorithm* [1] to derive the original determinants y_i and d. Finally, we reduce the rationals $x_i = y_i/d$ to normal form. The total complexity of the algorithm is then $O(n^3 l^2 + n^4 l)$ [28]. The basic idea for the parallelization of this algorithms is as follows;

- **Mapping:** The k images (A^j, b^j) can be computed in parallel.
- **Determinants:** The k determinant families (d^j, 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.

The PACLIB program first creates k `det` tasks, one `cra_d` task and n `cra_y` tasks. Each `det` task maps the given system into a domain $\mathbb{Z}_{p_j}$ and computes the determinant family (d^j, y_i^j). The `cra_d` task takes the list of `det` tasks and receives in a non-deterministic order their results to compute d. In the same way, each `cra_y` task computes its y_i, waits for the result of the `cra_d` task and performs the final reduction y_i/d.

The first two pictures in Figure 4 visualize the profiling information that was generated by a program run solving an equation system of dimension 15 with 18 processors. In the left picture, each horizontal line represents the execution time of a task; the medium picture displays the utilization of the processors. The right picture in this figure shows how much the PACLIB program was faster than the sequential SACLIB program for three equation systems with various dimensions (5, 10 and 40) and maximum lengths of the matrix entries (30, 300 and 900 bit).

Since k is usually very large, all processors are saturated with work during the determinant computation phase. A significant part of this phase is overlapped

[4] An upper bound for k can be determined from the entries of (A, b).

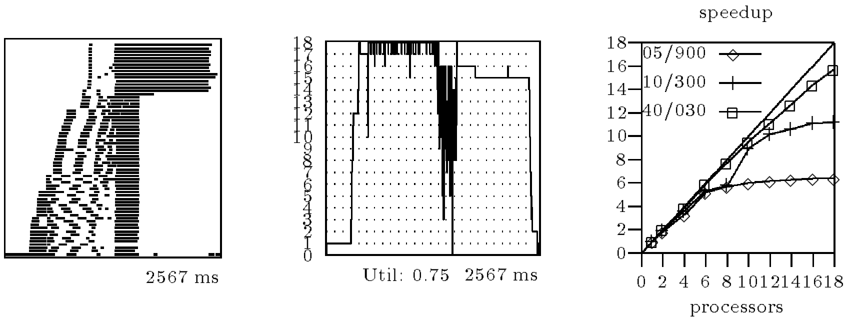

Fig. 4. Exact Solution of Linear Equation Systems

with the execution of the `cra_d` and the `cra_y` tasks, because the `det` results are delivered in a non-deterministic order. The final reduction of the solutions is performed by the `cra_y` tasks, therefore the maximum speedup is bounded by the dimension of the system. However, the speedup is very good, if if the dimension of the system is larger than the number of available processors. For the case of small dimensions, we have developed somewhat more efficient variants of the algorithm [22].

7 Related Work

Finally, we want to relate the concepts of PACLIB to other parallel systems:

Functional Programming: The PACLIB task model is influenced by functional programming [25] and related concepts as dataflow [16] and guarded horn clause languages [11]. PACLIB tasks are similar to the "futures" of MULTILISP [12]. However, PACLIB uses an explicit synchronization function to access task results, while futures are transparent to the program. This is possible since in MULTILISP all operand types are checked at runtime. The `eval` construct of LINDA [8] also resembles the functional model of parallelism.

Non-Determinism: There exist approaches to incorporate non-determinism into a functional framework [3]. Guarded horn clause languages [11] offer committed choice non-determinism. LINDA [8] allows the non-deterministic selection of tuples that match a given pattern. Message passing languages like OCCAM [14] offer the non-deterministic reception of messages from different sources.

Streams: Streams are a concept of non-strict functional languages [25], data flow languages [16] and guarded horn clause languages [11]. In contrast to message passing channels, streams are first-order objects that can be accessed by many tasks simultaneously. There also exist more general forms of non-strict data structures (e.g. I-STRUCTURES [2]).

Garbage Collection: There is a large amount of literature on garbage collection in a parallel environment (for a bibliography, see e.g. [18]). For the PACLIB garbage collector, we chose a relatively simple solution which is a natural extension of the Saclib mark and sweep collector and provides a comparative efficiency.

Parallel Computer Algebra Systems: Most approaches to parallel computer algebra combine sequential computer algebra systems with parallel programming models:

- **PARSAC** [15] is a parallel variant of the computer algebra library SAC-2 based on Mach threads. The system supports a shared heap where each thread owns a set of memory pages from which it allocates new cells. A terminating thread copies its result into the page set of the parent thread and frees its own page set.
- **Linda-Maple** [9] is a combination of the computer algebra system Maple with the Linda model. Each Maple process has a separate heap. Interprocess communication is based on the Linda tuple space by character strings into which Maple arguments and results are converted.
- ||**MAPLE**|| [23] is a combination of Maple with the guarded horn clause language Strand. Parallel programs are written in Strand while sequential algebraic computations are performed by Maple processes (each with a separate heap). The communication between both layers is based on Strand character strings.

Light-Weight Concurrency: Packages similar to the μSystem are incorporated into Mach [26], the Sun version of Unix [24] and the Charm Kernel [27].

8 Conclusions and Further Work

Paclib is now a reliable and efficient tool for the implementation of algebraic algorithms on shared memory multi-processors. Our current work concentrates on the extension of the Paclib library that now includes parallel algorithms for resultant computation, the exact solution of linear equation systems, the Gröbner bases method and others. We will also continue to introduce new features (e.g. task priorities) into the Paclib kernel. Furthermore, we work on the development of an X11-based performance analysis environment and on the design of a high-level programming interface that generates parallel Paclib code from functional specifications.

9 Acknowledgements

The authors would like to thank Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl and Tudor Jebelean who helped to design the Paclib concept. Volker Stahl, Werner Landerl, Wolfgang Stöcher, Mark Encarnacion, Carla Limongelli and others designed and implemented several Paclib programs. This work was supported by the Austrian Science Foundation (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. Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. Computation Structures Group Memo 269, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1987. Also in: Proceedings of the Workshop on Graph Reduction, Los Alamos, New Mexico, September 28 – October 1, 1986.
3. P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Functional Programming Languages and Computer Architectures*, volume 523 of *Lecture Notes in Computer Science*, pages 538–568, Harvard, Massachusetts, USA, 1991. Springer, Berlin.
4. B. Buchberger. Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. In N. K. Bose and D. Reidel, editors, *Recent trends in Multidimensional Systems*, chapter 6, pages 184–232. D. Reidel Publishing Company, Dordrecht-Boston-Lancaster, 1985.
5. B. Buchberger, G. Collins, M. Encarnation, H. Hong, J. Johnson, W. Krandick, R. Loos, A. Mandache, A. Neubacher, and H. Vielhaber. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.
6. B. Buchberger, G. E. Collins, R. Loos, and R. Albrecht, editors. *Computer Algebra — Symbolic and Algebraic Computation.* Springer, Vienna, New York, 1982.
7. 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(9):929–964, September 1990.
8. N. Carriero and D. Gelernter. *How to Write Parallel Programs.* MIT Press, 1990.
9. B. W. Char. Progress Report on a System for General-Purpose Parallel Symbolic Algebraic Computation. In *Proceedings of the ISSAC'90, Tokyo, Japan, August 20–24*, pages 96–103, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, 1990. ACM Press, New York.
10. G. E. Collins and H. Hong. Partial CAD Construction in Quantifier Elimination. Technical Report OSU-CISRC-10/89 TR 45, Computer and Information Science Research Center, Ohio State University, Columbus, OH, 1987.
11. I. Foster and S. Taylor. *Strand — New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.
12. R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Prog. Lang. Syst.*, 7(4):501–538, October 1985.
13. H. Hong, W. Schreiner, A. Neubacher, K. Siegl, H.-W. Loidl, T. Jebelean, and P. Zettler. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992. Also: Technical Report ACPC/TR 92-9, ACPC Technical Report Series, Austrian Center for Parallel Computation, July 1992.
14. Inmos. *Occam 2 Reference Manual.* Prentice Hall, New York, 1988.
15. W. Küchlin. The S-Threads Environment for Parallel Symbolic Computation. In *Computer Algebra and Parallelism, Second International Workshop*, pages 1–18, Ithaca, USA, May, 1990. Springer, Berlin. Volume 584 of Lecture Notes in Computer Science.

16. R. S. Nikhil. ID (Version 88.0) Reference Manual. Computation Structures Group Memo 284, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988.
17. J. Rothnie. Kendall Square Research Introduction to the KSR1. In H.-W. Meuer, editor, *Supercomputer '92 — Anwendungen, Architekturen, Trends (Applications, Architectures, Trends)*, pages 104–113, Mannheim, Germany, June 25–27, 1992. Springer, Berlin.
18. M. Rudalics. *Multiprocessor List Memory Management.* PhD thesis, Johannes Kepler University, Linz, Austria, 1988. Also: Technical Report 88-87, RISC-Linz, Johannes Kepler University, Linz, Austria, 1988.
19. W. Schreiner. The Design of the PACLIB Kernel. Technical Report 92-33, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992. Also: Technical Report ACPC/TR 93-4, ACPC Technical Report Series, Austrian Center for Parallel Computation, February 1993.
20. W. Schreiner and H. Hong. A New Library for Parallel Algebraic Computation. In R. F. Sincovec et al., editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume II, pages 776–783, Norfolk, Virginia, March 22–24, 1993. SIAM. Also: Technical Report 92-73, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992. Also: Technical Report ACPC/TR 93-9, ACPC Technical Report Series, Austrian Center for Parallel Computation, February 1993.
21. W. Schreiner and H. Hong. PACLIB — A System for Parallel Algebraic Computation on Shared Memory Computers. In H. M. Alnuweiri, editor, *Parallel Systems Fair at the Seventh International Parallel Processing Symposium*, pages 56–61, Newport Beach, CA, April 14, 1993. IPPS '93. Also: Technical Report 93-03, RISC-Linz, Johannes Kepler University, Linz, Austria, February 1993. Also: Technical Report ACPC/TR 93-12, ACPC Technical Report Series, Austrian Center for Parallel Computation, February 1993.
22. W. Schreiner and V. Stahl. The Exact Solution of Linear Equation Systems on a Shared Memory Multiprocessor. Technical report, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.
23. K. Siegl. Parallelizing Algorithms for Symbolic Computation Using ‖MAPLE‖. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–186, San Diego, California, May 19-22, 1993. ACM Press. Also: Technical Report 93-08, RISC-Linz, Johannes Kepler University, Linz, Austria, February 1993.
24. Sun Microsystems. *System Services Overview, Lightweight Processes*, May 1988.
25. B. K. Szymanski, editor. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, 1991.
26. A. Tevanian, Jr. and R. F. Rashid. MACH: A Basis for Future UNIX Development. Technical Report CMU-CS-87-139, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, June 1987.
27. University of Illinois. *The CHARM (3.2) Programming Language Manual*, December 1992.
28. F. Winkler. Computer Algebra I. Technical Report 88-88, RISC-Linz, Johannes Kepler University, Linz, Austria, 1988.
29. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison Wesley, New York, 1990.

This article was processed using the LaTeX macro package with LLNCS style