PACLIB — A System for Parallel Algebraic Computation on Shared Memory Multiprocessors*

W. Schreiner

schreine@risc.uni-linz.ac.at
RISC-Linz, Johannes Kepler Univ.
A-4040 Linz, Austria

H. Hong

hhong@risc.uni-linz.ac.at
RISC-Linz, Johannes Kepler Univ.
A-4040 Linz, Austria

Abstract

This paper gives an overview on the structure and the use 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 pipelining and a parallelized garbage collection.* Paclib *has been implemented on a* Sequent Symmetry *multiprocessor and is portable to other shared memory machines and workstations. Several algebraic algorithms have been implemented in* Paclib *and demonstrate the effectiveness of the system.*

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 [2]. All these algorithms operate with arbitrary precision arithmetic; they are therefore more expensive with respect to time and memory 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 [7]. Our goal has been to develop a professional tool for the simple and efficient implementation of parallel algorithms in computer algebra and related areas. Paclib is a combination and extension of two free software packages:

- Saclib [1] 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 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 and the Gröbner Bases Algorithm.

- μSystem [3] is a library of C functions 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 give an overview on the 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.

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

- **Data Sharing:** 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 aborted.
- **Streams:** Tasks may be connected by streams of data and may 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 [5]; a comprehensive description of the PACLIB kernel is given in [6]. The development of PACLIB applications is illustrated in [8] and [4].

2 Programming Interface

2.1 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*`)`

```
Word PIPROD(Word a, Word b) {
  Word l, n, i, j, b0, b1;
  Word t, tlist;
  l = LENGTH(b);
  n = l / N + (l % N != 0);
  tlist = NIL;
  for (i = 0; i <= n; i ++) {
    ADVANCE(N, b, &b0, &b1);
    j = i*N;
    t = pacStart(IPRODS, 3, a, b0, j);
    tlist = COMP(t, tlist);
    b = b1;
  }
  for (i = 0; i < n-1; i++) {
    b0 = pacWaitListRm(&tlist);
    b1 = pacWaitListRm(&tlist);
    t = pacStart(ISUM, 2, b0, b1);
    tlist = COMP(t, tlist);
  }
  return(pacWaitOne(FIRST(tlist)));
}
```

Figure 1: Parallel Integer Multiplication

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.

Figure 1 illustrates this model of parallelism by a parallel version of the school algorithm for the multiplication of two long integers a and b. All SACLIB objects are denoted by the standard type `Word`; `COMP` returns a new list cell and thus represents the interface to the heap management kernel. `PIPROD` splits b into subintegers b_0 of N digits and multiplies a with each b_0 in parallel. Each task executes the function `IPRODS` that multiplies a with b_0 and shifts the result by j digits. `PIPROD` iteratively waits for the results delivered by any two tasks and starts a new task that adds the results. Finally, there remains a single task left that returns the total result to the caller.

The non-deterministic semantics of `pacWait` gives the possibility to write programs whose behavior depends on the order in which tasks deliver their result. However, 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

```
Word producer(Word n, pacStream s) {
  Word i;
  for (i = 0; i < n; i++)
    pacPut(&s, F(i));
  pacCloseOne(s);
  return(1);
}
Word consumer(pacStream s) {
  Word v;
  while (pacGetOne(&v, &s))
    G(v);
  return(1);
}
Word pacMain() {
  pacStream s;
  pacTask p, c;
  s = pacOpen();
  p = pacStart(producer, 2, 100, s);
  c = pacStart(consumer, 1, s);
  return(pacWaitOne(c));
}
```

Figure 2: Producer/Consumer Parallelism

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.

2.2 Streams Model

There are several parallel algebraic algorithms (e.g. for *critical pair completion* [2]) for which the functional 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

d `= 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`.

By streams, a *producer/consumer* form of parallelism can be implemented as depicted in Figure 2: the main task creates an empty stream and passes it to two tasks `producer` and `consumer` that generate and process a stream of values, respectively. In general, networks of tasks connected by streams may be constructed. By utilizing the non-deterministic features of `get`, even *manager/worker* schemes of computation are possible [5].

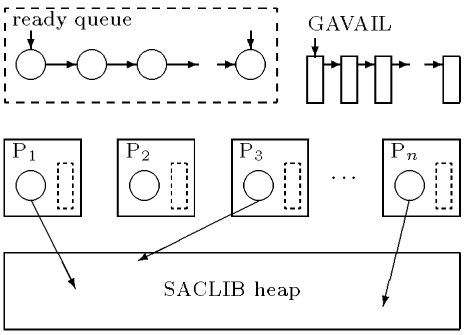

Figure 3: System Design

3 System Design

The Paclib kernel consists of the following main components (depicted in Figure 3):

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 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` is a global list of free lists 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.

3.1 Memory Management

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.

3.2 Task Management

What is the physical implementation 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 alive. After termination, 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.

3.3 Stream Management

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 An Application Example

It is beyond the scope of this paper to present all the algorithms that have been implemented in PACLIB. Instead, we will summarize the results we gained by parallelizing an algorithm for the *exact solution of linear equation systems [8]:* 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 sizes are not limited by the word length of any computer. These numbers have therefore to be represented by sequences of computer words. Furthermore, each x_i shall be the *exact* solution of the given system; therefore it has to be represented by a pair of integers (the numerator and the denominator of the rational, respectively).

One of the most efficient methods for this problem is based on *Cramer's Theorem* which says that $x_i = y_i/d_i$ where $d_i = \det(A)$ 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 performed using *modular arithmetic:* We take k prime numbers p_j (a bound for k can be determined from (A, b)) and map the system into k systems (A^j, b^j) over the finite fields $\mathbb{Z}_{p_j}$. Assuming that the p_j fit into single computer words, the determinant families (d_i^j, y_i^j) can be computed very fast. Applying the *Chinese Remaindering Theorem*, we derive d from the d^j and all y_i from the y_i^j and finally compute $x_i = y_i/d$.

The PACLIB program that implements a parallel version of this algorithm starts k `det` tasks, one `cra_d` task and n `cra_y` tasks. Each `det` task maps (A, b) into an image (A^j, b^j) and computes the determinant family (d^j, y_i^j). The single `cra_d` task receives the d^j from the `det` tasks and computes d. Similarily, each `cra_y` task computes y_i, receives d from the `det` task and performs the final reduction y_i/d. The commu-

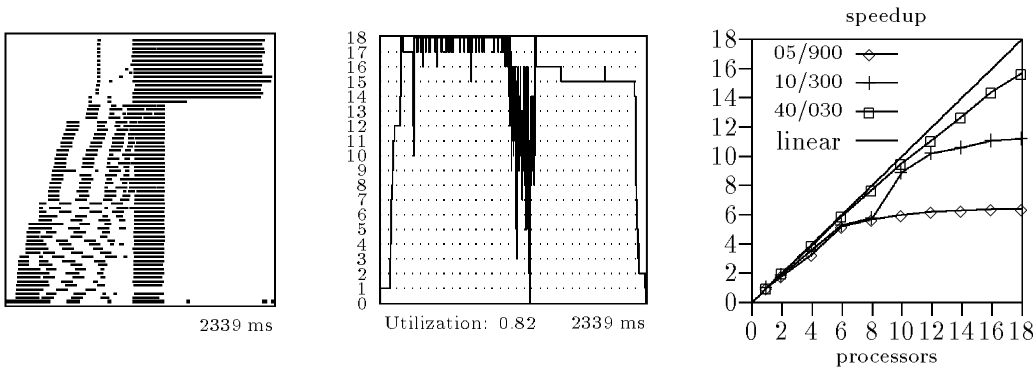

Figure 4: Exact Solution of Linear Equation Systems

nication between tasks is entirely based on the functional model of parallelism by passing (lists of) task handles as task arguments and applying `pacWait` to receive the required results. Since the results of the various `det` tasks are delivered in a non-deterministic order, a significant part of the Chinese Remaindering phase can be overlapped with the computation of the determinants.

Figure 4 shows the dynamic behavior of the PACLIB program for an equation system of dimension 15 with 18 virtual processors. The left picture visualizes trace information that was generated by the program: each line represents one task and displays the time when the task was active. The medium picture shows the utilization of the processors during the program run. The right picture displays the speedups that could be achieved compared to the sequential SACLIB implementation of the algorithm for three equation systems of dimension 5, 10 and 40 and initial integer lengths of 900, 300 and 30 bit. The maximum speedup is 16 for the largest system but bounded by the dimension of the smaller systems (which restricts the number of `cra_y` tasks). For these cases, we have developed more efficient variants of the algorithm [8].

Acknowledgements

The authors would like to thank Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl and Tudor Jebelean who helped to develop the PACLIB concept. Volker Stahl, Mark Encarnacion, Carla Limongelli, Wolfgang Stöcher and Werner Landerl implemented several PACLIB algorithms. This work was supported by the Austrian Science Foundation (FWF) grant S5302-PHY "Parallel Symbolic Computation".

References

[1] B. Buchberger, G. Collins, et al. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[2] B. Buchberger, G. Collins (editors), et al. *Computer Algebra — Symbolic and Algebraic Computation.* Springer, Vienna, New York, 1982.

[3] P. Buhr and R. Stroobosscher. The μSystem: Providing Light-weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software — Practice and Experience*, 20(9):929–964, September 1990.

[4] H. Hong. Parallel Computation of Modular Multivariant Polynomial Resultants on Shared Memory Machine. Submitted to the *2nd International Conference of the Austrian Center for Parallel Computation*, Gmunden, Austria, October 4–6, 1993.

[5] H. Hong, W. Schreiner, A. Neubacher, et al. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992.

[6] W. Schreiner. The Design of the PACLIB Kernel. Technical Report 92-33, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[7] W. Schreiner and H. Hong. A New Library for Parallel Algebraic Computation. In *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, Virginia, March 22-24, 1993.

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