The Design of the PACLIB Kernel*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
email: `schreine@risc.uni-linz.ac.at`

July 3, 1992

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 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.

*Sponsored by the Austrian Science Foundation (FWF) in the frame of the project S5302-PHY "Parallel Symbolic Computation".

1 Introduction

The current situation for designers of parallel algorithms in computer algebra and related areas is rather dissatisfying. The few existing systems for parallel algebraic computation are either bound to special purpose hardware [Roch, 1988] or not wide-spread operating systems [Kuechlin, 1990] or they have an experimental nature [Char, 1990]. The situation is also caused by the fact that there are currently no accepted programming standards for parallel computers.

This paper describes the runtime kernel of PACLIB, a new system for parallel algebraic computation on shared memory computers. Shared memory computers and multi-processor workstations are currently the most wide-spread parallel computer architectures and run unter standard operating systems (derivates of UNIX). The primary motivation for the development of PACLIB was to create a professional tool for parallel computer algebra with the following characteristics:

1. **Efficiency:** We aimed to create a system that most efficiently manages a large shared heap with automatic garbage collection which allows all tasks to communicate by references to shared data structures.
2. **Usability:** We wanted to provide a convenient user interface that allows researchers in algebraic computation to express and implement their algorithms in a natural way without carrying for low-level system features.
3. **Portability:** The system should be portable to other machines with shared memory running under UNIX. An extension of the model to networks of such machines was also considered but deferred for the moment.

PACLIB is a combination and extension of two public domain software systems:

1. **Saclib** [Collins *et al.*, 1992] is a large library of algebraic algorithms written in the C language. These algorithms are based on a list-processing kernel that provides automatic garbage collection. Furthermore, the package supports exact arithmetic with arbitrary precision integers and rationals.
2. The **uSystem** [Buhr and Stroobosscher, 1990] is a package that provides light-weight concurrency on mono-processor workstations and shared memory multi-processors running under the UNIX operating system. The μSYSTEM provides an efficient management of *tasks* i.e. light-weight parallel processes.

PACLIB combines the list-processing and algebraic facilities of SACLIB with the concurrency possibilities of the μSYSTEM. In particular, PACLIB supports the following features:

- **Concurrent Tasks:** Any program function can be executed as a separate task; the scheduling of tasks and the balancing of load among processors is performed automatically.

- **Shared Memory Communication:** All data can be *shared* among tasks, i.e. task arguments and task results are passed by reference. No copying of data structures is required.
- **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.
- **Task Abortion:** 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 and Task Pipelines:** Tasks may be connected by *streams* of data and work in an overlapping fashion (pipelining).
- **Parallel Garbage Collection:** The garbage collector is based on a *mark and scan* scheme where the marking of of used cells and the reclamation of free cells are both parallelized.

PACLIB is a system still under evolution but we believe that the main features have already been stabilized. The system has been implemented on a Sequent Symmetry shared memory computer with 20 processors running under the Dynix/ptx version of UNIX System V.3. However, it is portable to any other machine supported by the μSYSTEM package [Buhr and Stroobosscher, 1990].

This report is intended as a detailed description of the internal structure of the PACLIB kernel. It explains how the system has been *designed*, not how it can be *used*. The actual user interface of PACLIB is described in [Hong *et al.*, 1992].

2 SACLIB

SACLIB is a library of algebraic algorithms that supports

- Arbitrary precision integer and rational arithmetic.
- Polynomial arithmetic.
- Linear algebra.
- Polynomial GCD and resultant computation.
- Polynomial factorization.
- Real root calculation.
- Algebraic number arithmetic.

The library is based on a list processing kernel with automatic garbage collection (applying the the mark and scan method). All *objects* in SACLIB are either fixed-size integers (called *atoms*) or references to sequences of objects (called *lists*). The user interface to this kernel is the function `COMP(`h`,`t`)` that allocates a heap cell and constructs a new list with head element h and tail list t.

PACLIB provides the functionality of the list processing kernel in a parallel environment: all tasks share a common heap space and there exists a `COMP` operation with the same interface as in SACLIB. Hence, any sequential SACLIB function can be further used (even by parallel tasks) without any change to the source code[1].

A detailed description of SACLIB can be found in [Collins *et al.*, 1992].

3 The uSystem

The μSYSTEM is a library of C functions that supports concurrency on shared memory multiprocessors and workstations running under the UNIX operating system. The central concept of the μSYSTEM is the *task* i.e. a program component that can be executed simultaneously with and independent of other tasks. The μSYSTEM kernel distributes tasks among a set of *virtual processors* for execution. The virtual processors are implemented as UNIX processes that are themselves scheduled by the operating system. On a multi-processor, these processes are executed by different hardware processors truly in parallel. Pictorially, we have the situation

$$\begin{array}{cl} \text{tasks} & \\ \downarrow & \mu\text{SYSTEM} \\ \text{virtual processors} & \\ \downarrow & \text{UNIX} \\ \text{hardware processors} & \end{array}$$

where the relation $\downarrow$ reads "is executed at" (and "μSYSTEM" respectively "UNIX" represents the associated scheduler).

The creation and and synchronization of UNIX processes requires a large runtime and memory overhead since each process has a separate execution environment that is protected from all other processes (*heavy-weight concurrency*). The UNIX facilities for parallel programming are therefore only suitable for rather *coarse-grained parallelism*.

On the other hand, a μSYSTEM task essentially consists just of a separate thread of control (defined by the contents of the stack, registers and program counter). The μSYSTEM kernel manages these tasks in a very efficient way (*light-weight concurrency*). All virtual processors share a common address space; hence communication is possible by global variables and synchronization can be performed by shared semaphores. The μSYSTEM facilities for parallel programming therefore allow the utilization of rather *fine-grained parallelism*.

[1]Provided that the function does not destructively update input arguments which could interfere with other tasks.

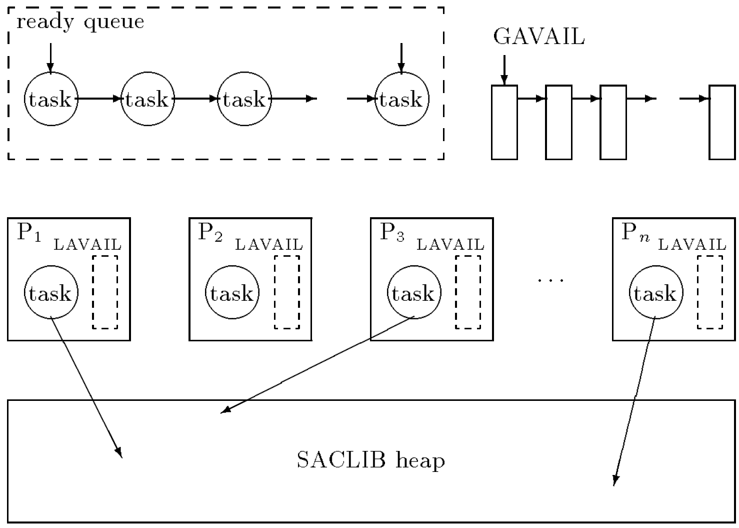

Figure 1: The PACLIB Structure

4 The Paclib Structure

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

1. **Shared Heap:** The SACLIB heap is an array of fixed-size list cells. All virtual processors (and therefore all PACLIB tasks) may access this heap simultaneously and without synchronization overhead.

2. **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.

3. **Ready Queue:** This queue contains all tasks that are *active* (i.e. ready for execution) but not *running* (there are usually more tasks than virtual processors). The μSYSTEM scheduler selects tasks for execution from the head of this queue; new tasks and tasks that are preemptively descheduled are inserted at the tail.

4. **Virtual Processors:** A set of virtual processors executes PACLIB tasks. Each processor has a local list `LAVAIL` of free heap cells that is used in `COMP` operations.

The PACLIB memory management scheme is described in the next section in more detail.

5 Heap Management

As already mentioned, PACLIB provides the same interface to the list processing subsystem as the sequential SACLIB. However, the internal implementation of the `COMP` operation in PACLIB differs considerably from that in SACLIB:

If a task calls `COMP`, a new list 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 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 different extremes that both have severe disadvantages:

1. **Global Available List Only:** In order to keep `GAVAIL` consistent, it must not be simultaneously updated by different processors (i.e. `GAVAIL` is protected by a semaphore). 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 although there are still plenty of free cells available (since these cells are attached to the local available lists of other processors).

The length of the lists in `GAVAIL` (which is the maximum length of the local available lists `LAVAIL`) can be defined by the user. Within certain limits, the overall efficiency of a program seems not to be very sensitive to the value of this parameter.

If a processor triggers garbage collection, it puts the currently executed task back into the ready queue, sets a global garbage collection flag and waits for all other processors to enter the garbage collection state, too. Each processor checks after a context switch (i.e. before a new task is picked for execution) whether the garbage collection flag is set and in this case acknowledges this signal. Depending on the remaining time slice of a task, this may take some hundred milliseconds.

After all processors have been synchronized, garbage collection takes place in two phases:

1. **Mark:** In this phase, all cells that can be (directly or indirectly) referenced by any task are marked. Since all user tasks are descheduled, all heap references are on the stacks of these tasks (on a context switch, the contents of the registers are pushed on the stack). Each virtual processor now picks a task descriptor, scans the associated stack and marks all heap cells that can be reached. If there are more tasks than virtual processors, the mark phase is performed in parallel by all processors.

2. **Scan:** In this phase, all unmarked cells are reclaimed to construct a new list `GAVAIL`. For this purpose, 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 last constructed free list becomes the new `LAVAIL` of each processor. The scan phase is always performed in parallel by all processors.

Besides the heap references on the task stacks, there may exist several references in global (static) variables declared by the SACLIB function `GCGLOBAL`. If there are no more stacks to be marked left, the first finished virtual processor marks the cells reachable from those global variables. The management and garbage collection of GCA handles (*garbage collected arrays* [Collins *et al.*, 1992]) proceeds in a similar way as for list handles.

6 Task Creation

The PACLIB function

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

creates a new task that executes *fun(args)* and can be referenced via the handle *task*[2]. *task* is a first-order SACLIB object, e.g. it can be stored in a SACLIB list. The result of this task may be retrieved (an arbitrary number of times) by any other task that knows the handle *task*.

What is the physical interpretation of *task*? One possibility is to implement this handle as a μSYSTEM *task descriptor* i.e. as a reference to the workspace defining the μSYSTEM task. We may reserve a location in this workspace (that contains besides other information the task stack) for the result of the task. Immediately before a task stops execution, it stores its result there that may be later retrieved by other tasks.

However, this approach has the disadvantage that the rather large workspace cannot be reclaimed after the corresponding task has terminated[3]. Moreover, the PACLIB model of parallelism is basically a functional one, i.e. tasks communicate only indirectly by their results[4]. For this reason, we decided to implement *task* as a *result descriptor* which is a (reference to a) SACLIB list with the following elements:

sem	task	value	wait

- `sem` is a pointer to a μSYSTEM semaphore that provides mutual exclusion on the result descriptor.

[2] The actual syntax of `pacStart` is a little bit different.

[3] The μSYSTEM function `uAbsorb` returns the result of a task and deallocates its workspace. However, this function may be called only *once* for each terminated task.

[4] An exception is the *stream* model of communication described in Section 9.

- `task` is a pointer to the μSYSTEM descriptor of the task that computes the result. If the task has already delivered its result, `task` is the constant `TERMINATED`.
- `value` contains, if `task` = `TERMINATED`, the result of the task. Otherwise, it contains the tail of a queue of tasks (i.e. μSYSTEM task descriptors) waiting for the result.
- `wait` contains, if `task` $\neq$ `TERMINATED`, the head of the queue of tasks waiting for the result.

As soon as a task has written its result into this descriptor, it may deallocate its workspace and terminate execution. The (large) *task descriptor* therefore occupies space only as long as the task is alive. After its termination, only the (small) *result descriptor* remains left. Moreover, since this descriptor is actually a SACLIB list, it is subject to garbage collection and is (together with the result value) reclaimed when no task has the handle *task* any more.

A PACLIB *task descriptor* is just a μSYSTEM task descriptor extended by the following fields:

sem	
prev	next
result	
value	
wait	
sub	
stack	stop

- `sem` is a pointer to a μSYSTEM semaphore that provides mutual exclusion on the task descriptor.
- `prev` and `next` connect all PACLIB task descriptors into a doubly linked list.
- `result` is a reference to the result descriptor of the corresponding task.
- `value` is a location that receives the result of another task if the current task is blocked in a call of `pacWait`.
- `wait` contains, if the task is blocked in a call of `pacWait`, a list of those result descriptors that the current task is waiting for.
- `sub` is a list of result descriptors of all subtasks created by the current task.
- `stack` is an integer that denotes the stack size to be allocated for all tasks created by the current task.
- `stop` is a flag that is set if another task wants to abort the current task.

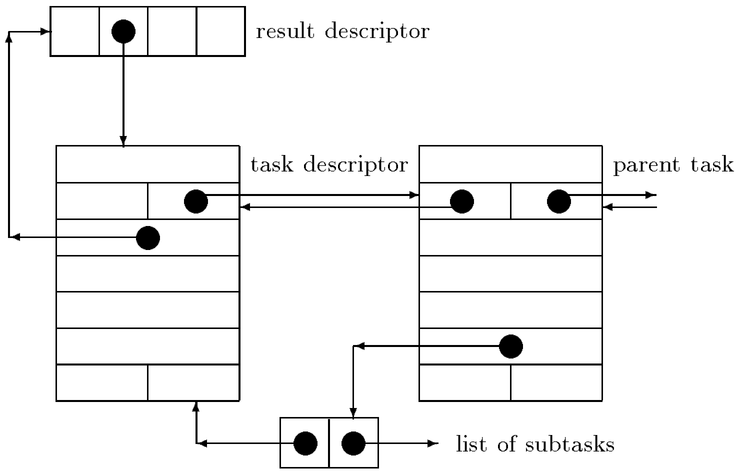

Figure 2: A new PACLIB task

When a new task is started, a result descriptor and a task descriptor are allocated. The size of the stack to be reserved in the task descriptor is determined by the `stack` field in the descriptor of the parent task (whose value can be changed by the function `pacStack`). The result descriptor is initialized with a new μSYSTEM semaphore, a pointer to the task descriptor and an empty task queue.

The fields of the task descriptor are initialized with the start address of the function to be executed, the corresponding arguments, a reference to the result descriptor and all the other values required to determine the task. The task descriptor is inserted into a doubly linked list that contains all tasks that are currently alive (which is required for the mark phase of the garbage collector). The task descriptor is finally put into the ready queue for execution. We then have the situation depicted in Figure 2.

Actually, the new task does not execute the function *fun* provided by the user but a function `pacShell` that receives the address of *fun* and the corresponding arguments. `pacShell` calls *fun* and gets the result of this function call. This result is stored in the `value` field of the result descriptor and all tasks blocked on this descriptor are awaked. The delivery of task results is described in the next section in more detail.

7 Task Results

A task may receive the result value of other tasks by the PACLIB function

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

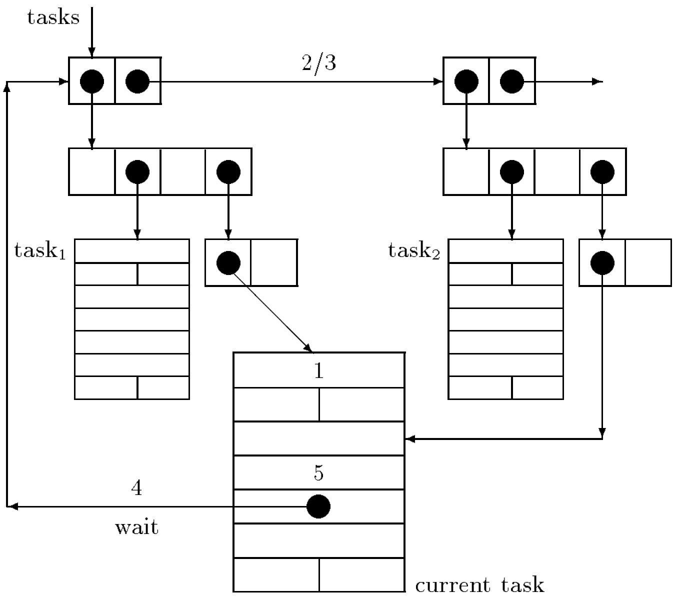

Figure 3: Waiting for a Result

`pacWait`[5] returns the result *value* of one of the tasks whose handles are in *tasks* and writes the handle of the corresponding task into the location *tptr*[6]. If all denoted tasks are still active, `pacWait` blocks until one task terminates and then delivers the result of this task.

This construct provides the basis of communication between PACLIB tasks. Its non-determinism gives additional flexibility to the user but also imposes an additional level of complexity on the implementation. The algorithm that implements `pacWait` works roughly as follows (the numbers in Figure 3 refer to the steps of the algorithm):

1. **Lock:** The semaphore associated to the descriptor t of the current task is locked. This prevents the *tasks* from delivering their result to t before `pacWait` has reestablished a consistent state.

2. **Scan:** All result descriptors in *tasks* are scanned and the result descriptor r of the current task is linked into the waiting queues of these descriptors.

3. **Found:** If a result descriptor in *tasks* already contains a result, r is removed from all waiting queues it has previously been linked into. t is unlocked and the result value is returned.

4. **Sleep:** If none of the results is already available, the result descriptor list *tasks* is stored in the `wait` field of t. t is unlocked and the current task is blocked.

[5]The actual syntax of `pacWait` is a little bit different.

[6]Remember that task handles are actually *result descriptors*.

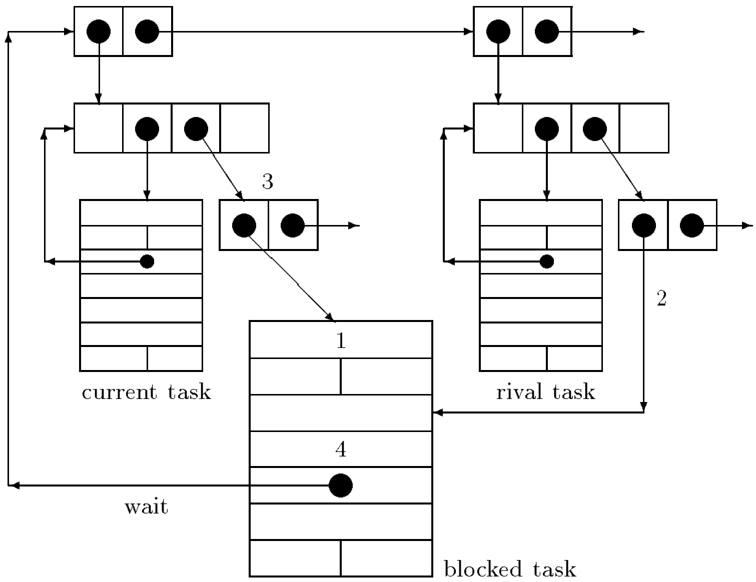

Figure 4: Delivering the Result

5. **Awake:** After the task has been awaked, the `value` field of t contains the result of one of the *tasks* it waited for. Likewisely, `wait` has been overwritten with the result descriptor of the task that returned the result.

After the `pacShell` has received the result of *fun*, it forwards the value to all tasks waiting on the result descriptor r (see Figure 4):

1. **Pick Task:** The semaphore associated to r is locked and the first task t waiting in the queue is determined. In order to prevent a deadlock situation, r is unlocked before t is locked and then locked again. Without this detour, a rival task delivering its result to t might already have locked t and itself try to lock r in order to remove t from the waiting queue of r.

2. **Check Task:** If t has disappeared from the waiting queue of r, the corresponding task has in the meantime received its result from a slightly faster rival task. Otherwise, the current task has "won" the race and may deliver its result to t.

3. **Remove Descriptor:** t is removed from the waiting queue of r and from the waiting queues of all other result descriptors that the task is waiting for (the `wait` field of t holds a list of these descriptors). This guarantees that only one result is delivered to t.

4. **Deliver Result:** The result value is written into the `value` field of t and r is written into the `wait` field of t. The corresponding task is then reawaked and can extract the desired information from t.

After the result has been delivered to all waiting tasks, it is written into the `value` field of r. The `task` field of r is set to `TERMINATED` and r is unlocked. The task then unlinks its descriptor from the global task list, releases its workspace and terminates.

8 Task Abortion

A task can abort other tasks by a call of the function

`pacStop(`*tasklist*`);`

`pacStop`[7] aborts each task whose result descriptor r is in *tasklist* (and all tasks created by these tasks). It first locks r and then determines from the `task` field of r whether the corresponding task is still alive. In this case, the `stop` flag in the task descriptor t is set and the `task` field in r is overwritten with the constant `STOPPED`. Afterwards, r is unlocked.

Before each context switch (and before a new task is started), the status of the `stop` flag of a task is checked. If the flag is set, the `stop` flags of all subtasks created by the stopped task are set, too (a list of all subtasks is contained in the `sub` field of t). The task then unlinks t from the task list, deallocates its workspace and stops execution.

9 Streams

The functional form of inter-task communication provided by task arguments and task results is sufficient for most applications. However, PACLIB also supports the concept of *streams* as an additional and very flexible tool that allows to transmit a sequence of values from one task to another task (or even to a set of tasks). A *stream* is an abstract data type that can have one of three states:

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

The usual application of streams (that are first-order PACLIB objects) is the following: an *empty* stream s_0 is created and passed as an argument to two new tasks. The one

[7] The actual syntax of `pacStop` is a little bit different.

task (the *producer*) changes the state of the stream from *empty* to $\langle v_0, s_1 \rangle$ where v_0 is a SACLIB object to be transmitted and s_1 is a new and *empty* stream. The other task (the *consumer*) receives the value v_0 and the stream s_1 that can be used for the transmission of another value. If the consumer tries to receive a value while s_0 is still in the *empty* state, the task is automatically blocked.

In this fashion, a sequence of streams $s_0, s_1, \ldots, s_n$ is generated that transmit the corresponding data objects $v_0, v_1, \ldots, v_{n-1}$ (actually, each stream s_i with $i < n$ transmits the pair $\langle v_i, s_{i+1} \rangle$). The producer finally sets the state of s_n to *closed*, which terminates the communication. It is important to note that *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 an arbitrary number of times. This is the main difference of the (functional) concept of streams to message passing models via pipelines or channels where a value is destructively removed after the transmission.

The implementation of streams is surprisingly simply and entirely based on already introduced concepts. A stream is nothing but a PACLIB *result descriptor* without an associated *task descriptor*:

1. `pacOpen()` (that is defined to create an *empty* stream) returns a result descriptor that does not yet contain a value. The `task` field of the descriptor is set to the constant `OPEN`.

2. `pacPut(`v`,`$sptr$`)` writes a cell `COMP(`v`,`s`)` (where s refers to a new stream) into the result descriptor at location *sptr* and releases all blocked tasks. The algorithm for this function is the same as for the function `pacShell` described in Section 7.

3. `pacClose(`s`)` writes a `NIL` reference into the result descriptor s and releases all blocked tasks. The algorithm is again the same as for the function `pacShell`.

4. `pacGet(`*sptr*`, `*vptr*`, `*rptr*`, `*streams* delivers the contents of a stream out of a set of *streams* in the same non-deterministic way as the function `pacWait` described in Section 7.

The definition of streams as result descriptors without task descriptors reflects the basic motivation for the introduction of this concept. Actually, the effect of a stream can be simulated by a sequence of tasks each of which returns a result cell composed of an element and of a reference to a new task that computes the next result cell[8]. However in order to improve the efficiency of this form of communication, we decided to introduce streams as an explicit concept that allows a single task to transmit a sequence of values to another task.

The price of this additional flexibility is the danger of inconsistencies introduced by programming errors. If a programmer changes by `pacPut` or `pacClose` the state of a stream that is *not* empty, an unpredictable behavior may occur[9]. In order to minimize

[8] This is the stream model of *dataflow languages*.

[9] This corresponds to the violation of the *single assignment rule* in dataflow languages.

this danger, the function `pacPut` updates the location of the input stream by the new rest stream i.e. the next `pacPut` operation can be performed on the same stream variable.

10 Performance Monitoring

If a PACLIB program is called with the command line option `+t1`, profiling information is generated during the program run. This information consists of fixed-size records (3 words each) that are written into a shared buffer which is flushed into the file `pac.dat`:

- **Startup:** The first record is of the form

main	*proc*	—

  where *main* is the μSYSTEM identifier of the task executing `pacMain` and *proc* is the number of processors used.

- **Task Generation:** If a task is generated, a record of the form

parent	*new*	-1

  is generated where *parent* is the μSYSTEM identifier of the task that created a task with the identifier *new*.

- **Time Mark:** If `pacMark` is called, a time mark of the form

mark	*time*	-2

  is generated where *mark* is an arbitrary integer and *time* denotes the current time (in ms).

- **Time Slice:** After each context switch, a record of the form

task	*from*	*to*

  is generated where *task* is a μSYSTEM task identifier and *from* and *to* are two non-negative integer values that denote the start and the end of the time slice (in ms) during which *task* was running.

From this information, the tools `pacgraph` and `pacutil` generate LaTeX pictures that visualize the behavior of the program and the utilization of the processors during the program run, respectively (see [Hong *et al.*, 1992] for details).

11 Benchmarks

The basic time overhead imposed by the PACLIB system is negligible. A sequential program runs as fast when compiled with PACLIB as with SACLIB. Moreover, even a sequential SACLIB program compiled with PACLIB profits from the parallelized garbage collector that is extremely efficient. For the parallel PACLIB constructs, we give in the following several key benchmarks that characterize the overall efficiency of the system.

All benchmarks were performed on a Sequent Symmetry with 128 MB of main memory and 20 processors i80386 running at 20 MHz clock frequency. In order to prevent system processes from interfering with the tests, we applied only up to 18 virtual processors (such that the corresponding UNIX processes were always scheduled for execution). The programs were designed such that no paging occurred during these tests. Similar benchmarks for the pure μSYSTEM can be found in [Buhr and Stroobosscher, 1990].

11.1 Task Creation

A function call takes on the Sequent Symmetry 8 μs; the creation of a PACLIB task takes in average 230 μs i.e. approximately 30 times longer. However, if many new tasks are allocated, the average task creation time may increase up to several ms. The reason for this is that the workspace of each task is allocated by a conventional `malloc` operation from the system heap; this may trigger the transfer of a virtual memory page from the disk into the physical main memory (which is very slow).

11.2 Task Communication

The amount of time required for the delivery of a task result (by `pacWait` or `pacGet`) essentially depends on two influence values:

- **Blocking:** is the receiver blocked (i.e. is the result demanded before it is available) or not?
- **Non-Determinism:** The more results a task waits simultaneously for, the more runtime overhead is imposed.

Furthermore, if a task is in the n-th position of a waiting queue of a result descriptor, the $n-1$ predecessor tasks receive the result value first. However, the runtime overhead for delivering a result to all tasks is just n times the runtime overhead for the delivery to a single task and therefore not considered in the following benchmark.

Our test program measured the time that a task spends in `pacWait` with respect to the number of task results m that this task is simultaneously waiting for. The delivering task is in this test always the last task in the task list. In one test, the receiving task finds the result of the last task already available and is therefore not blocked. In the

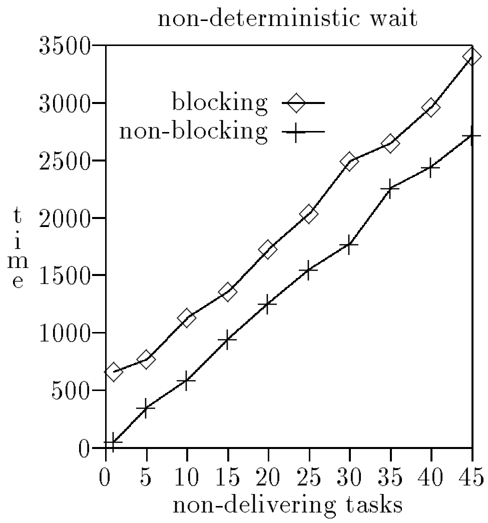

Figure 5: Non-Deterministic Wait

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. In both tests, the task descriptor has therefore to be removed from the waiting queues of $m-1$ result descriptors, once by the receiving task itself (in `pacWait`), once by the delivering task (in `pacShell`).

The results depicted in Figure 5 show that the time t required for the delivery of a task result is linear in the number of tasks m that the receiving task is simultaneously waiting for. Roughly, the formula $t = m * 60 + b$ μs holds where $b = 0$, if the receiving task is not blocked, and $b = 600$, if the current task is blocked and reawaked again[10]. In other words, a deterministic wait costs 60 μs, each further task to wait for additionally demands the same amount of time. This is pretty efficient but it also shows that non-determinism should be applied with care.

11.3 Task Granularity

In order to get some total information about the minimum granularity that makes sense for a PACLIB program, we ran the following benchmark test: a main task started 100 subtasks (that were calibrated to execute for a certain amount of time) and received the corresponding dummy results deterministically in sequence. The time from the

[10] Actually, these 600 μs also include 260 μs cost for two context switches, since the benchmark was run with a single virtual processor.

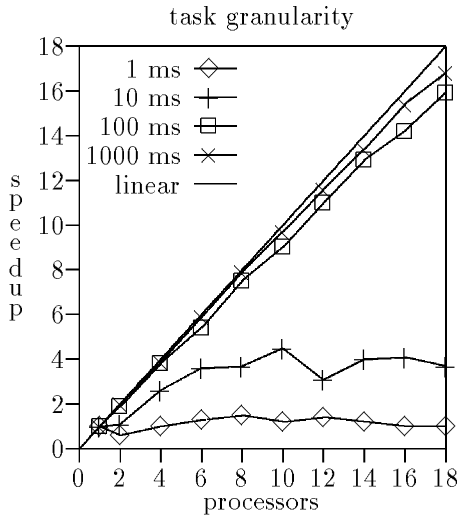

Figure 6: Task Granularity

start of the first task until the recept of the last result was measured. The program was run with 1 to 18 virtual processors and the speedup was computed (see Figure 6).

For tasks running only 1 ms, no speedup could be achieved at all; the granularity was definitely too low to outweigh the parallelization overhead. For a granularity of 10 ms, a maximum speedup of 4 turned out; hence the granularity was still below the "break-even point". However, for tasks running 100 ms, a very good speedup could be achieved (16 with 18 processors); a granularity of 1 s achieved almost linear speedup (17 with 18 processors).

As a result we can say that PACLIB tasks should be designed for a minimum runtime of 100 ms. Lower granularities do not pay off the overhead of task creation and synchronization; larger granularities improve the efficiency of a program not significantly any more.

11.4 Garbage Collection

As described in Section 5, both phases of the PACLIB garbage collector are parallelized: In the mark phase, the stacks of all tasks are investigated and the referenced cells marked in parallel; in the scan phase, each processor scans a different section of the heap for free heap cells. While the amount of parallelism in the mark phase therefore depends on the number of live tasks, the work in the scan phase is always evenly distributed among all processors.

We ran a set of benchmark tests that measured the garbage collection time of the PACLIB system by a number of dummy tasks that evenly allocated heap cells. By these tests, the influence of three different parameters on the garbage collection became visible (see Figure 7).

- **Heap Size:** the program was run with heaps of size 4, 8, 16 and 32 MB. The larger the heap is, the larger the efficiency of the scan phase becomes (where all virtual processors can work independently in parallel). However, the difference is not very significant even between the smallest and the largest heap (wit 50 active tasks and a load of a little bit more than 40%).
- **Heap Load:** the amount of cells that could not be reclaimed was calibrated to 0, 20, 40 and 60% of the total heap size[11] The more heap cells are in use, the more time the mark phase takes relatively to the total garbage collection time. For an almost empty heap, a speedup of almost 15 could be achieved; for a heap with more than 60% cells in use, the maximum speedup decreased to 12 (both cases with 50 tasks and an 8 MB heap).
- **Task Number:** the amount of tasks that allocated cells was tuned from 1 task to 25, 50, 100 and 200 tasks (with an 8 MB heap that was more than 40% loaded). The more tasks are active, the better the load balancing between virtual processors in the mark phase becomes. In case of a single task, only a speedup of 3 was possible (the mark phase done by a single processor consumed then roughly a third of the total garbage collection time). For all other cases, a speedup of approximately 13 could be achieved.

From these figures, it turns out that garbage collection in PACLIB is rather efficient. Provided that there are sufficient tasks to saturate all processors in the mark phase with work, the speedup scales linear up to 10 processors and achieves a maximum of almost 15 with 18 processors. In absolute numbers, each processor reclaims 1–1.5 MB of heap memory per second (depending on the heap load) which is identical to the sequential SACLIB garbage collection rate on this machine.

12 Comparison with Other Work

Finally, we want to mention other projects aimed at the development of parallel systems for algebraic computation and relate this work to ours.

12.1 Parsac-2

PARSAC-2 [Kuechlin, 1990] is a parallel implementation of SAC-2 (a former name for SACLIB) based on the operating system MACH. Similar to PACLIB, this system

[11] Actually, a basic system load of roughly 5% has to be added to these values.

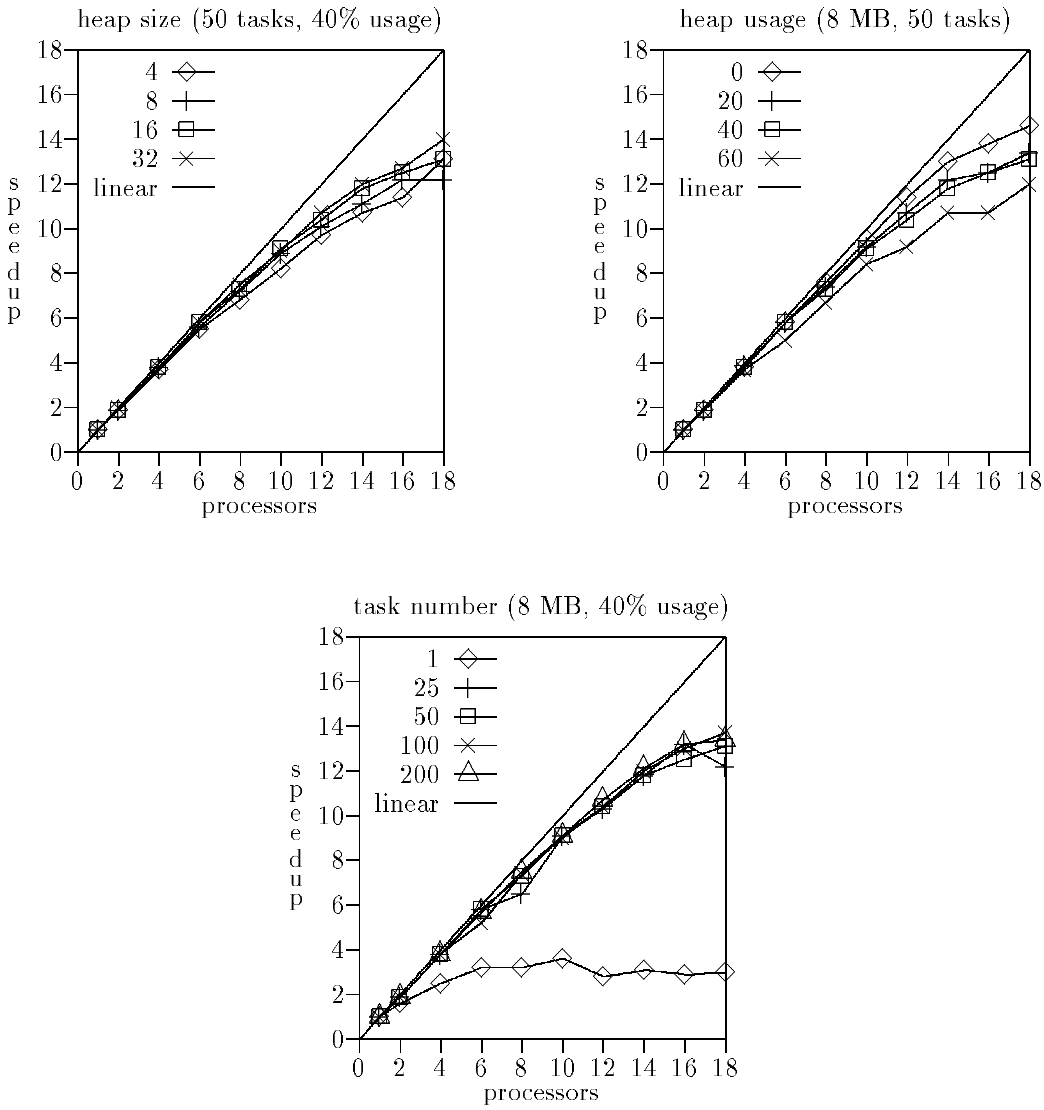

Figure 7: Garbage Collection

provides concurrency based on *threads* (light-weight processes) and shared memory communication through a common heap. The main difference between PARSAC-2 and PACLIB is the management of heap memory:

Each PARSAC-2 thread owns a set of contiguous heap pages that contain all data structures created by this thread. The `COMP` operation allocates a new heap cell from a local page and, if necessary, extends the page set of a thread by a new page from a global free list. There is no explicit garbage collection but heap pages are reclaimed on the termination of a thread only. The data structure that denotes the result of the thread is copied on a new page that becomes part of the page set of the parent thread. Afterwards, all heap pages of the terminated thread are reclaimed into the global free list.

The PARSAC-2 scheme therefore reclaims the total heap space of a thread on its termination (*preemptive garbage-collection*) but it does not reclaim the no more used heap space of a thread during its execution. This may cause space leaks in threads that run for a long time and allocate many data structures. Furthermore, since the result structure of a thread must be duplicated, sharing information goes lost and identical copies of data structures waste heap space.

On the contrary, in PACLIB a true garbage collector is able to reclaim any unused cell at any point of the computation. Moreover, data structures are between tasks always passed by *reference* without any duplication overhead. There is also no PARSAC-2 equivalent to the PACLIB support of non-determinism and the streams model of communication.

12.2 Linda-Maple

LINDA-MAPLE [Char, 1990] is a parallel implementation of the computer algebra system MAPLE with the parallel programming model LINDA. The central communication medium in this model is the *tuple space*, a shared bag of data objects by which the different processes may communicate. In LINDA-MAPLE these may be user-defined processes or MAPLE processes that perform algebraic computations.

Each MAPLE process has a separate heap space with separate garbage-collection. Communication between user-defined processes and MAPLE processes is performed via the input/output interface: a new MAPLE process receives the textual description of a function call as its input argument; the result structure is converted into a text string that is placed into the tuple space.

LINDA-MAPLE has been implemented on a shared-memory multiprocessor but is in principle portable to distributed memory architectures or workstation networks. The inter-process communication scheme of this system is also the most important difference to PACLIB. Since the parallel MAPLE processes do not share a common heap, all data structures are converted into their printable form, transferred from one process to another and then converted into the internal representation again. In PACLIB, this detour is not required.

12.3 ||MAPLE||

||MAPLE|| [Siegl, 1991] is a parallel implementation of MAPLE with the parallel programming language STRAND. STRAND is a declarative language of the guarded horn clause type (i.e. with PROLOG-like syntax) that is available on many parallel computer architectures. It has an interface to the C language that allows to incorporate sequential functions and programs (such as MAPLE).

A ||MAPLE|| program is essentially a STRAND program where the body of a logical clause may contain a MAPLE call. The communication between STRAND and MAPLE is performed by the text interface of MAPLE. The result of a MAPLE call is converted into printable form and packed into a STRAND string that can be forwarded to other STRAND processes where it is transformed into the internal representation again.

||MAPLE|| is running on all parallel architectures supported by the STRAND environment, e.g. transputers, hypercubes and shared memory computers. Again, the main difference of PACLIB to this system concerns the form of inter-process communication: MAPLE data structures have to be encoded into a neutral representation, transferred from one process to another and then reconverted again. In PACLIB, data structures are always shared among tasks and only references are transmitted.

13 Conclusions

The main goal in the design of PACLIB was the development of a professional tool that most effectively utilizes the resources of a shared memory computer and allows researchers in computer algebra to implement their algorithms in a natural way. We think that this goal has been achieved. Our further work will now concentrate on the parallelization of a good deal of algebraic methods in PACLIB and on the refinement of the system according to the experiences we will gain in the future.

14 Acknowledgements

PACLIB was developed by the parallel computation group at RISC-Linz on an initiative of its leader Hoon Hong. The detailed design and the implementation of PACLIB were both performed by the author of this paper. Andreas Neubacher provided valuable support as the maintainer of SACLIB. Kurt Siegl, Hand-Wolfgang Loidl, Tudor Jebelean, Peter Zettler and Klaus Kusche helped in many discussions to clarify the main concepts. This work was sponsored by the Austrian Science Foundation (FWF) in the frame of the research project S5302-PHY "Parallel Symbolic Computation".

References

[Buhr and Stroobosscher, 1990]
Peter A. Buhr and Richard 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.

[Char, 1990]
Bruce 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.

[Collins *et al.*, 1992]
George Collins, Hoon Hong, Mark Encarnation, Jeremy Johnson, Werner Krandick, and Andreas Neubacher.
A SACLIB Primer.
Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

[Hong *et al.*, 1992]
Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, Kurt Siegl, Hans-Wolfgang Loidl, Tudor Jebelean, and Peter Zettler.
PACLIB User Manual.
Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992.

[Kuechlin, 1990]
Wolfgang Kuechlin.
A Parallel SAC-2 Based on Threads.
Technical report, Computer and Information Sciences, Ohio State University, Columbus, Ohio, April 1990.

[Roch, 1988]
Jean-Louis Roch.
PAC — Towards a Parallel Computer Algebra Co-Processor.
In *Computer Algebra and Parallelism*, pages 33–50, Workshop at the TIM3 Laboratory, University of Grenoble, France, June 1988. Academic Press, London.

[Siegl, 1991]
Kurt Siegl.
Programming Techniques for ||MAPLE||.
Technical Report 92-10, RISC-Linz, Johannes Kepler University, Linz, Austria, 1991.