*J. Symbolic Computation* (1997) **11**, 1–000

# The Design of the SACLIB/PACLIB Kernels†

Hoon Hong
Andreas Neubacher
Wolfgang Schreiner

{hong,neubacher,schreiner}@risc.uni-linz.ac.at
*Research Institute for Symbolic Computation (RISC-Linz)*
*Johannes Kepler University, Linz, Austria*

*(Received 8 April 1997)*

This paper describes the design of the runtime systems of two variants of the SAC-2 computer algebra library: Saclib and Paclib. Saclib is a C version of SAC-2, supporting automatic garbage collection and embeddability. Paclib is a parallel version of Saclib, supporting light-weight concurrency, non-determinism, virtual tasks, and parallel garbage collection.

## 1. Introduction

In this paper, we report an on-going work on developing two variants of the SAC-2 computer algebra library [Collins and Loos, 1982]: Saclib and Paclib, where Saclib is a C version of SAC-2, and Paclib is a parallel version of Saclib. In particular we concentrate on the design and implementation of the "kernels" (i.e. the runtime systems) of these two packages. We decided to report both systems in the same paper since they are closed related.

The SAC-2 computer algebra library is a collection of carefully designed algorithms for various operations over integers, rationals, finite fields, algebraic numbers, and polynomials with such coefficients. The algorithms in the library are coded in the programming language Aldes [Loos, 1976]. In order to produce an executable program using the library, one usually translates the Aldes code into FORTRAN and uses a FORTRAN compiler.

The first part of this paper describes the work on developing a C version of SAC-2, namely Saclib [Buchberger et al, 1992]. Other people have worked on translating SAC-2 into languages such as Lisp and Modula-2 [Langemyr, 1989] [Kredel, 1988]. We chose the C language for our work since it is most wide-spread, portable, and efficient. Portability was an important issue since we plan to distribute our work in the public domain

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

0747–7171/90/000000 + 00 $03.00/0 

so that any researchers in computer algebra or engineers can freely obtain a copy and use it. While designing the SACLIB kernel, we desired to meet various requirements. Among them, the most important ones were automatic garbage collection and embeddability. These requirements were met mainly by adopting a conservative garbage collection scheme where the system stack is used for identifying the potential roots of active list cells.

The second part of this paper describes the work on developing a parallel version of SACLIB, namely PACLIB [Hong et al, 1992]. The kernel of PACLIB was designed to meet the following requirements: upward compatibility to the SACLIB kernel, high-level parallel programming model, light-weight concurrency, non-determinism and speculative parallelism, communication by heap references, and parallel garbage collection.

For this purpose, we adapted the $\mu$SYSTEM package [Buhr and Stroobosscher, 1990] that supports light-weight concurrency on shared-memory multiprocessors and workstations running under UNIX. The system has been implemented on a 20 processor SEQUENT SYMMETRY shared memory multi-processor. The visualization environment PACVIS [Hong et al, 1994] allows to monitor the dynamic behavior of a PACLIB program.

## 2. Related Work

Our work on the further development and parallelization of SAC-2 was preceded by and owes much to Wolfgang Küchlin's PARSAC-2 system [Küchlin, 1990]. This package has introduced *S-threads* (symbolic threads) on top of the C-threads interface of the operating system MACH.

In addition to the functionality of C-threads, S-threads can allocate list cells from a local set of heap pages attached to the thread. If the set is exhausted, new pages are allocated from a global pool of memory pages. Memory is reclaimed by *preventive garbage collection* [Küchlin and Nevin, 1991]: a terminating S-thread copies its result into the page set of the parent thread and adds its own page set to the global pool. If a task exclusively owns the references to all cells it has allocated (either because it has not started any subtasks or because it has duplicated all argument structures into the page sets of its subtasks), it may also perform local garbage collection.

Küchlin and Ward have also implemented the concept of *virtual C-threads* [Küchlin and Ward, 1992]: A virtual thread may be "inlined" (i.e. executed by the parent thread) if it has not yet been executed at the point its result is required. Thus a restriction of MACH (that allows only a limited number of threads) is overcome, and, by reducing the overhead for thread creation and synchronization, parallel cut-off points in divide-and-conquer algorithms can be avoided. Idling processors or terminating threads do not themselves realize respectively execute virtual threads; instead an additional scheduler thread is awaked from time to time to realize new threads whenever the system load becomes too low.

NETWORK-SAC [Seitz, 1990] is a parallel version of SAC-2 for the coarse-grain parallelization of algebraic programs on computer networks. Every computer runs a single SAC-2 process (the *algorithm server*) that receives from a scheduler messages describing tasks to be executed.

Task arguments and results are converted into a character format, transferred over the network and then transformed back into the internal representation. Since each algorithm server uses a separate heap, no global garbage collector is required. The concept of *bags* is applied to express non-deterministic computations; futures (i.e. task references) are put into a bag, any access to the bag then returns the first available result.

Recently, also a parallel version of SAC-2 for the KSR-1 virtual shared memory computer has been presented [Kredel, 1994]. Similar to PARSAC-2 this system is based on the C-threads interface (provided by the machine's native operating system OSF/1). To improve the data locality of programs, every processor uses a separate heap that is garbage-collected independently of the others. Task arguments and results are copied into the heap of the corresponding processor.

## 3. The SACLIB Kernel

This section gives an overview of the kernel of SACLIB [Buchberger et al, 1992], the C version of the computer algebra library SAC-2 [Collins and Loos, 1982], [Loos, 1976].

### 3.1. Design Goals

The main points taken into consideration when SACLIB was designed were the following:

1 **Compatibility to** SAC-2**:** As we intended to build upon the work done for SAC-2 by using its wide range of algorithms, the kernel should be such that it would support the SAC-2 algorithms without making changes to the existing code (apart from syntactic translations) necessary.
2 **Efficiency:** The system (and especially the low-level kernel functions) should be computationally as efficient as possible.
3 **Portability:** The system should be easily portably among UNIX systems, thus making it available to a large number of users.
4 **Programming environment:** The system should be easy to use both for the algorithm designer and the applications programmer. This demand encompasses a simple interface to the system's functions and the availability of program development tools.
5 **Embeddability:** It should be possible to link SACLIB to application programs and to call the functions directly, thereby avoiding time-consuming communication protocols.
6 **Hidden garbage collection:** Developers should not be required to do special coding for bookkeeping tasks such as garbage collection.

### 3.2. Overall Design

Aiming at the goals described above resulted in the following major design decisions:

1 **Programming Language:** By using C as the development language of SACLIB, several goals were already met to a high degree: C allows the production of highly efficient code and applications based on the standard C functions are highly portable. Furthermore, C is the basic language of the UNIX operating system, thus the powerful UNIX source code debugging and maintenance tools can be used.

2 **Conservative Garbage Collection:** Since SACLIB may be embedded in all kinds of applications, we must not make any assumptions about the context in which SACLIB functions are called, in particular not about the layout of program data. However, we have to identify all potential roots of active list cells for garbage collection. Since we do not want to burden the program (respectively the programmer) with extra code for the bookkeeping of heap references, we use a *conservative* garbage collection scheme where it is safe to mistake a memory item for a heap reference [Boehm and Weiser, 1988].

In the following section, we describe how the programmer can use the system.

### 3.3. Programming Interface

To the C programmer, SACLIB is simply another library of functions which are linked to a program and can be called (after proper initialization) from anywhere in the code.

Figure 1 shows the basic layout of a program using SACLIB functions. The user-supplied routine `sacMain` is called by a built-in C language `main` routine (not shown here) which carries out all the necessary initialization and cleanup. In this way, coding overhead for using SACLIB functions is minimized.

Furthermore, note that the programmer need not insert any statements regarding garbage collection, and neither are they added by some preprocessor. A minimal amount of bookkeeping is necessary for the handling of global variables. An example is given in Figure 2. See also the following section for details.

As an alternative to `sacMain` it is also possible to call the initialization function explicitly at some later point in the program, as shown in Figure 3. Here `some_subroutine` is a module of a larger application program. Calls to SACLIB's algebraic algorithms are embedded in this subroutine, so that the resources (esp. memory) used by SACLIB are only allocated when needed. Thereby, the use of SACLIB is completely encapsulated and has no impact on the design of the main program.

In SAC-2 and its programming language ALDES, the kernel's list processing and garbage collection facilities are accessed by the three primitive functions `COMP`, `FIRST`, and `RED`, which correspond to `cons`, `car`, and `cdr` in the LISP. Implementing C versions of these ALDES functions allowed us to convert all other algorithms of the SAC-2 library by a simple syntactic translation from ALDES to C. Below we describe the semantics of these SAC-2 / SACLIB list processing primitives.

The basic data type in SACLIB is called `Word`. This is similar to the type `int` from C, with some restrictions:

> *Atoms* (also called `BETA`-*digits* or simply *digits*) are all integer constants or variables of type `Word` of value greater than `-BETA` and less than `+BETA`, where `BETA` is a positive integer constant. For a machine with $n$-bit words, `BETA` should not be greater than $2^{n-3}$. This limit is set in order to avoid additive overflow and to provide bits for list processing and garbage collection. Usually `BETA` is set to be $2^{n-3}$.
>
> *Lists* are either empty, which is indicated by the constant value `NIL`, or they are constructed by the function `COMP(x,L)`, which puts the atom or list `x` at the head of the (possibly empty) list `L` and returns the resulting list.

```
#include "saclib.h"

int sacMain(int argc, char *argv[])
{
        Word    I,F;
Step1:  /* Input. */
        SWRITE("Please enter an integer: "); I = IREAD();
Step2:  /* Compute factorial. */
        F = IFACTL(I);
Step3:  /* Output. */
        IWRITE(F); SWRITE("is the factorial of "); IWRITE(I);
        return(0);
}
```

**Figure 1.** A sample program.

```
#include "saclib.h"

Word    F = NIL;

int sacMain(int argc, char *argv[])
{
Step0:  /* Declare global variables. */
        GCGLOBAL(&F);
Step1:  /* Input. */
        ...
}
```

**Figure 2.** Fragment of a sample program using global variables.

```
#include "saclib.h"

... some_subroutine(...)
{
        Word dummy;
        ...         /* Other variable declarations. */

Step1:  /* Initialize SACLIB. */
        BEGINSACLIB(&dummy);
Step2:  /* Use SACLIB. */
        ...
Step3:  /* Remove SACLIB. */
        ENDSACLIB(SAC_FREEMEM);
        return(...);
}
```

**Figure 3.** Embedding SACLIB in a subroutine.

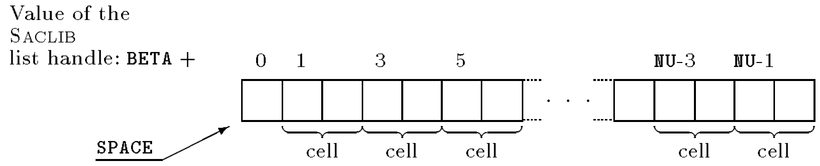



**Figure 4.** The `SPACE` array.

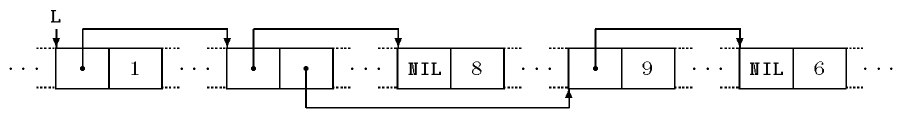



**Figure 5.** The cell structure of the list L = (1, (9, 6), 8).

> The two primitive list operations which are dual to `COMP` are `FIRST(L)`, which returns the first element of a (non-empty) list `L`, and `RED(L)`, which returns the reductum of a (non-empty) list `L`, i.e. the list `L` without its first element.

This scheme originally employed by SAC-2 allows the use of the standard C operators +, -, *, /, %, etc. for atoms provided that the results stay in the allowed range. Furthermore, the distinction between lists and atoms can be made based on the value of the variable, so it is not necessary to allocate and access additional memory for storing the type of an object. Its influence on garbage collection is made clear in the following sections.

### 3.4. Heap Management

In order to keep the SAC-2 semantics of list processing, its way of implementing lists was also used in SACLIB:

When SACLIB is initialized, the array `SPACE` containing `NU` + 1 `Word`s (with `NU` being a global variable defined in the kernel) is allocated. This array is used as the memory space for list processing.

Lists are built from *cells*, which are pairs of consecutive `Word`s the first of which is at an odd position in the `SPACE` array. *List handles* ("pointers" to lists) are defined to be `BETA` plus the index of the first cell of the list in the `SPACE` array. As mentioned above, the handle of the empty list is `NIL` (which is a constant equal to `BETA`). Figure 4 shows the structure of the `SPACE` array.

The first `Word` of each cell is used for storing the handle of the next cell in the list (i.e. the value returned by `RED`), while the second `Word` contains the data of the list element represented by the cell (i.e. the value returned by `FIRST`). Thus, we have `FIRST(L) = SPACE[L-BETA+1]` and `RED(L) = SPACE[L-BETA]`. Figure 5 gives a graphical representation of the cell structure for a sample list, with arrows representing list handles.

The C language allows us to improve the efficiency of `FIRST` and `RED`. We introduce two C pointers `SPACEB` and `SPACEB1` defined by `SPACEB` = `SPACE`−`BETA` and `SPACEB1` = `SPACE`−`BETA` + 1. Using these, we can set `FIRST(L)` = `SPACEB1[L]` and `RED(L)` = `SPACEB[L]`.

As already mentioned in the previous section, atoms are required to be integers $a$ with −`BETA` $< a <$ `BETA`. This allows the garbage collector and other functions operating on lists to decide whether a variable of type `Word` contains an atom or a list handle. Note that list handles can only be odd values between `BETA` and `BETA` + `NU`.

Most of the well-known garbage collection schemes require adding bookkeeping statements to the program code, e.g. for counting the number of references to a list cell or for declaring variables referencing list cells to the garbage collector. While such statements can be automatically inserted (by a separate pre-compiler) and need not increase the amount of work done by the programmer, they do increase the running time of the program, because they are executed even if there is a sufficient amount of memory available so that garbage collection is not necessary.

In SACLIB, time is spent for garbage collection only when its conservative garbage collector is invoked by `COMP` in the case of the kernel's internal list of available (i.e. unreferenced) cells being empty. A mark-and-sweep method is employed for identifying unused cells:

1 **Mark:** The processor registers, the system stack, and the global variables declared by a call to `GCGLOBAL` are searched for list handles. If a list handle is found, the cells of the list are traversed and marked. If the second `Word` of a cell contains a list handle, the corresponding cells are marked recursively. Marking of a cell is done by negating the value of its first word. Thus, the first word must contain only a list handle or `NIL`, while the second word can contain any word size data (such as single-precision floating point numbers).

2 **Sweep:** In the sweep step, the list of available cells is built: Unmarked cells in the `SPACE` array are linked to the list. If a cell is marked, it is only unmarked and not changed in any other way. Unmarking of a cell is done by negating (back) the value of its first word.

By this scheme it is guaranteed that all cells which are referenced (directly or indirectly) by any of the program's variables are not linked to the list of available cells. The problem here is that the stack in general does not only contain variables from SACLIB code, but also C pointers, floating point numbers, arbitrary integer values, etc. It might happen that some of these fall into the range of a legal list handle when interpreted as a `Word`. This may result in some cells, which should go into the list of available cells, being left unchanged. While this does not influence the correctness of the program, it does incur a trade off of memory needs against execution speed.

Extensive testing and heavy use of the system have shown that in general this does not impose a problem, i.e. the number of cells being kept allocated due to wrongly interpreted references is not significant compared to the amount of memory used.

## 4. The PACLIB Kernel

In this section, we describe the design of the PACLIB kernel which is a parallel variant of the SACLIB kernel for shared memory multiprocessors [Schreiner, Hong 1993].

### 4.1. Design Goals

In the design phase of the PACLIB kernel, we raised several demands the parallel computer algebra kernel should fulfill:

1 **Upward compatibility to the SACLIB kernel:** The PACLIB kernel should provide the same interface to the heap and the same efficiency of heap access as the SACLIB kernel. Each function of the SACLIB library should therefore run without change and without loss of speed when linked with the PACLIB kernel.
2 **High-level parallel programming model:** The system should be based on a high-level parallel programming model. It should be possible to spawn each function of the SACLIB library as a concurrent task and to retrieve the result of this function without any change of the interface. Tasks should be first-class objects.
3 **Light-weight concurrency:** Since many parallel algebraic algorithms are rather fine-grained, the runtime overhead for the creation of a new task should be very small and it should be possible to spawn thousands of tasks.
4 **Non-determinism and speculative parallelism:** Since the runtime of algebraic algorithms can often not be predicted, there should be a synchronization construct waiting for a whole set of tasks and returning the first available result. If the results of the other tasks are not needed any more, it should be possible to abort these tasks.
5 **Communication by heap references:** On a shared memory machine, task arguments and task results should be heap references only. Neither the creation of a new task nor its termination should require to copy whole SACLIB structures (which would increase the runtime and might duplicate originally shared substructures).
6 **Parallel garbage collection:** There should be a global garbage collector that on demand reclaims all of the no more referenced heap cells. The garbage collector should be parallelized in order to profit from the multi-processor architecture.

In the following subsections, we will describe how and to which extent we met these demands. A comprehensive description of the kernel design can be found in [Schreiner, 1992].

## 4.2. Overall Design

The task management of the PACLIB kernel is built on top of the $\mu$SYSTEM [Buhr and Stroobosscher, 1990]. This free software package consists of a library of C functions supporting light-weight concurrency on shared memory multiprocessors and UNIX workstations. The $\mu$SYSTEM kernel distributes *tasks* (light-weight processes) among *virtual processors* which are (heavy-weight) UNIX processes scheduled by the operating system. On a multi-processor, virtual processors are therefore executed by multiple hardware processors truly in parallel. The $\mu$SYSTEM task management is quite efficient and allows the utilization of rather *fine-grained parallelism*.

The PACLIB kernel consists of the following main components (depicted in Figure 6):

1 **Ready Queue:** This queue contains all tasks that are *active* but not *running*. The 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 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.

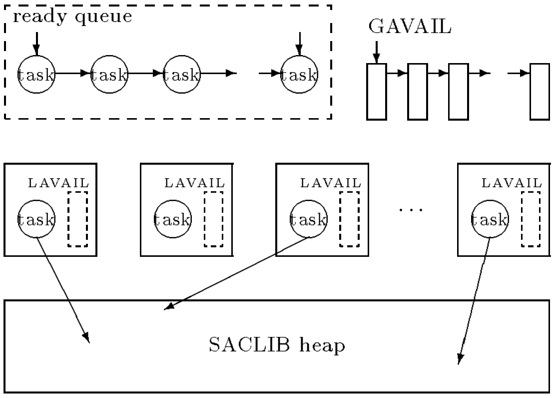



**Figure 6.** PACLIB Kernel Design

```
Word PIPROD(Word a, Word b) {           /* parallel integer product */
  Word l, n, i, j, b0, b1;
  Word t, tlist;
  l = LENGTH(b);                        /* length of b */
  n = l / N + (l % N != 0);             /* number of tasks */
  tlist = NIL;                          /* list of tasks */
  for (i = 0; i <= n; i ++) {           /* start tasks */
    ADVANCE(N, b, &b0, &b1);            /* split off subinteger b0 */
    j = i*N;                            /* shifting factor */
    t = pacStart(IPRODS, 3, a, b0, j); /* start task for (a*b0)<<j */
    tlist = COMP(t, tlist);             /* put task into list */
    b = b1;                             /* rest of b (without b0) */
  }
  for (i = 0; i < n-1; i++) {           /* collect results */
    b0 = pacWaitListRm(&tlist);         /* 1st result, remove task */
    b1 = pacWaitListRm(&tlist);         /* 2nd result, remove task */
    t = pacStart(ISUM, 2, b0, b1);      /* start task for b0+b1 */
    tlist = COMP(t, tlist);             /* put new task into list */
  }
  return(pacWaitOne(FIRST(tlist)));     /* result of last task */
}
```

**Figure 7.** Parallel Integer Multiplication

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 section describes the programming interface to the system in more detail.

### 4.3. Programming Interface

Most algebraic algorithms are based on purely mathematical functions that are entirely defined by their argument/result behavior. The basic 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.

$t$ = `pacStart(`$f$`, `$a_i$`)`

creates a new task that asynchronously executes $f(a_i)$. The task handle $t$ is a first-order object that can be passed to other functions/tasks and stored in any SACLIB data structure.

$v$ = `pacWait(&`$t$`, `$ts$`)`

returns the result $v$ of one of the denoted tasks $ts$ and returns the handle $t$ of the delivering task. If all denoted tasks are still active, `pacWait` blocks until a task terminates.

`pacWait` is a *non-deterministic* construct whose result in general depends on the system situation at runtime. Only when applied to a single task, its result is uniquely determined. The additional level of flexibility provided by non-determinism may help to considerably reduce the number of task blockings.

`pacWait` is also *non-destructive* i.e. it does not destroy the task whose result is delivered. As a consequence arbitrarily many tasks may asynchronously wait for the same set $ts$ without interference. This feature distinguishes the PACLIB kernel from most other thread packages except for the MultiLisp futures [Halstead, 1985].

There exist several variations of `pacWait`: `pacWaitListRm` takes as its single argument a SACLIB list of task handles and removes from the list the task whose result is delivered; `pacWaitOne` waits for the result of a single task.

Figure 7 illustrates the PACLIB model of parallelism by a parallel version of the school algorithm for the multiplication of two integers $a$ and $b$. `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.

`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.

There are several parallel algebraic algorithms (e.g. for *critical pair completion*) for which the purely functional model is not adequate. These algorithms can be better described in terms of tasks each of which iteratively 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* i.e. automatically synchronized lists that may be used for this purpose. For a comprehensive description of the PACLIB programming model, see [Hong et al, 1992].

### 4.4. Task Management

We applied and considerably extended the features of the $\mu$System in order to implement the Paclib parallel programming model which is higher-level and more general than the native one. There were two major problems that had to be solved:

1 How to represent the task handles returned by `pacStart` and
2 How to implement the non-deterministic functionality of `pacWait`.

For implementing task handles, there basically exist two possibilities: The first one is to view such a handle as a *task descriptor* i.e. as a reference to the workspace of the task. Before a task terminates, it stores its result in this workspace for the later retrieval by other tasks. Then either the workspace is not reclaimed as long as there are references to the task or the workspace is reclaimed when the result is received by another task. However, the first approach would waste too much space while in the second approach (implemented in the $\mu$System) tasks are not first-order objects any more.

Therefore we decided to split the description of a task into two separate entities:

1 The *result descriptor* is a small structure that serves as the rendezvous and synchronization point to other tasks.
2 The *task descriptor* is a small extension of the $\mu$System task descriptor that is placed on top of the task stack (32 KB by default).

A Paclib *task* is actually a reference to the result descriptor which is a Saclib list with the following elements:

| | | | |
|---|---|---|---|
| `sem` | `task` | `wait` | `value` |

`sem` is a pointer to a semaphore that provides mutual exclusion on the descriptor.
`task` is a pointer to the task descriptor of the computing task. If the task has already terminated, `task` is the constant `AVAILABLE`.
`wait` contains, if `task` $\neq$ `AVAILABLE`, the head of a queue of tasks blocked on an attempt to receive the result.
`value` contains, if `task` $\neq$ `AVAILABLE`, the tail of the `wait` queue and the result of the task, otherwise.

A *task descriptor* is a record that is placed on top of the stack of the task and contains (among others) the following fields:

<table>
<tr><td colspan="2">sem</td></tr>
<tr><td>prev</td><td>next</td></tr>
<tr><td colspan="2">result</td></tr>
<tr><td colspan="2">value</td></tr>
<tr><td colspan="2">wait</td></tr>
<tr><td colspan="2">sub</td></tr>
<tr><td>stack</td><td>stop</td></tr>
</table>

`sem` is a (pointer to a) semaphore that provides mutual exclusion on the task descriptor.

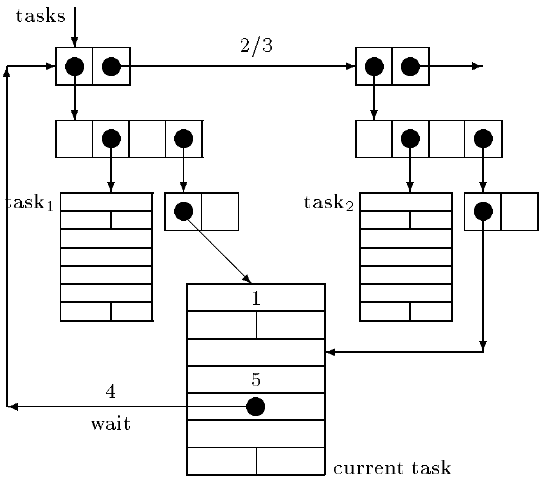



**Figure 8.** Waiting for a Result

`prev` and `next` connect all existing task descriptors into a doubly linked list.

`result` is a reference to the result descriptor of the 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.

`t = pacStart(`$f$`,`$a_i$`)` allocates a task descriptor and stores the function pointer $f$ and the arguments $a_i$ at the begin of the stack. A result descriptor is allocated and mutually linked with the task descriptor; a reference to this result descriptor is returned as the task handle $t$. The new task executes a function `pacShell` that calls $f(a_i)$. When $f$ has returned its result, `pacShell` stores the value in the result descriptor and awakes all tasks blocked on this descriptor. Finally `pacShell` erases the task link in the result descriptor, deallocates the task descriptor and terminates the task.

The large task descriptor therefore occupies space only as long as the task is executing. After termination only the small result descriptor remains active from where subsequent `pacWait` calls will return the result without blocking. Since this descriptor is a Saclib object, it is subject to the general garbage collection mechanism and together with the result value reclaimed when no task has the handle $t$ any more. Hence, also task results do not occupy memory longer than necessary.

The non-deterministic nature of `pacWait` actually requires a complex result delivery protocol that runs roughly as follows (the numbers in Figure 8 refer to the steps of the algorithm where the current task waits for a list of tasks containing task$_1$ and task$_2$):

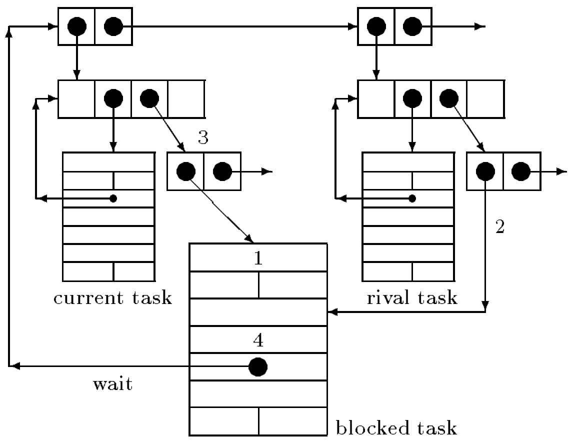



**Figure 9.** Delivering the Result

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 **Awake:** After the task has been awaked, the `value` field of $t$ contains the result of one of the *tasks* it waited for. Likewise, the `wait` field now contains 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 9 where the current task delivers a result to the blocked task):

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 awaked 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 `AVAILABLE` and $r$ is unlocked. The task then unlinks the task descriptor from the global task list, releases its workspace and terminates.

The details of this algorithm for the delivery of result tasks can be found in [Schreiner, 1992], its correctness has been formally verified in [Schreiner, 1993].

### 4.5. Heap Management

The Paclib kernel provides an operation `COMP` with the same interface and practically the same efficiency as the Saclib kernel. However, the organization of free lists in the Paclib kernel is much more sophisticated: There is a free list `LAVAIL` attached to each (virtual) processor and there is a global list of free lists `GAVAIL`. When a task executes `COMP`, the new cell is allocated from the `LAVAIL` of that processor that currently executes this task. If `LAVAIL` runs out of free cells, the processor picks the first list in `GAVAIL` as its new `LAVAIL` (this is the only overhead compared with the Saclib `COMP`). If `GAVAIL` is also empty, a global garbage collection is triggered. This two-level memory management scheme of Paclib is a compromise between two extremes:

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 Unix signal the execution of the other (virtual) processors. 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. Several experiments [Schreiner and Hong, 1993] show that the efficiency of the garbage collector is actually very high (a speedup of more than 14 could be achieved with 18 processors compared to the sequential Saclib collector).

### 4.6. Virtual Tasks

In Paclib parallelism may be utilized at a rather low-level yielding a large number of tasks with small grain-sizes. However we (as other researchers before us) soon faced an obvious discrepancy. Parallel program development is considerably simplified by the ability to create a large number of light-weight tasks, but each task requires a significant amount of system resources (in particular memory) and is subject to some performance penalty.

Since the number of physical processors is actually very limited (20 on our system), the large number of light-weight processes is not appropriately rewarded by reducing the computation time at the same scale. In practice, during the fine-tuning of a parallel program the number of tasks is therefore reduced at least to the same order of magnitude (say some 100) as the number of processors.

The techniques of *virtual threads* [Küchlin and Ward, 1992] or *lazy task creation* [Mohr et al, 1990] were developed to overcome this discrepancy. Virtual threads are basically just descriptions of threads that may be created at a low cost but are themselves not yet executable. The idea is that most of these virtual threads will never become real at their own (thus avoiding the overhead) but their descriptions will be executed by other real threads that are already in existence. Consequently, the number of real threads will be reduced and their grain size will be increased.

In the following subsection we describe the concept of virtual tasks as implemented in the PACLIB kernel. This implementation was inspired by the virtual S-thread package of [Küchlin and Ward, 1992] but implements a more elegant and more efficient task realization scheme by idling processors and terminating tasks (see the description below and compare with Section 2).

Virtual Paclib tasks are created by a function call

$t$ = `pacVirtual(`$f$`, `$a_i$`)`

with the same interface and basic semantics as a `pacStart` call. The difference between both calls is the activation time of the task $t$: a call of `pacStart` immediately creates a full task descriptor and links it into the ready queue of tasks to be selected for execution.

`pacVirtual` however only allocates a virtual task descriptor as depicted in Figure 10. This descriptor consists of the Paclib extension of the $\mu$System task descriptor and space for the function pointer $f$ and arguments $a_i$. Virtual task descriptors have fixed size (about 80 bytes) and are obtained by locked access from a central pool (that is dynamically extended on demand). The descriptor is linked into a "virtual" scheduling queue that is distinct from the ready queue of "real" tasks.

The new result descriptor contains a link to the virtual task descriptor; the handle $t$ returned by `pacVirtual` may participate in all operations like the handles of result descriptors that are returned by `pacStart` and thus connected to real tasks.

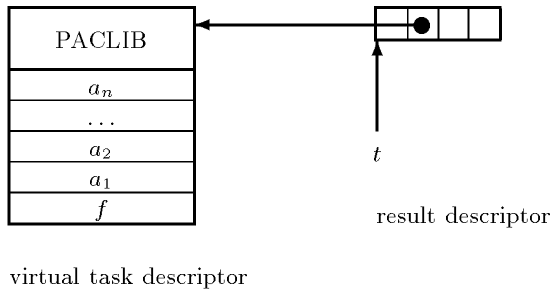



**Figure 10.** A Virtual PACLIB Task

A virtual task is "realized" (i.e. transformed into a real task) on the following occasions:

1 **Idling:** The scheduler of a virtual processor finds the global queue of executable tasks empty. In this case it picks a descriptor from the virtual task queue and allocates a full-sized task descriptor with a stack. The contents of the virtual descriptor are transferred into the task descriptor and the link of the corresponding result descriptor is reset to this descriptor. The virtual descriptor is freed and the new task is activated.
2 **Termination:** A task is going to terminate after it has delivered its result. In this case the task tries to grip a descriptor from the virtual task queue. If such a virtual task is found, the task overwrites its own descriptor with the retrieved information, resets the link in the corresponding result descriptor, frees the virtual task descriptor and restarts its execution with the computation of the denoted function.
3 **Deterministic Wait:** A task calls `pacWait` for a single task that is still virtual. In this case, the task grips the virtual task, computes the denoted function itself, delivers its result into the corresponding result descriptor (and to all tasks waiting there) and continues execution.
4 **Non-deterministic Wait:** A task calls `pacWait` for several tasks some of which are still virtual. In this case, the task realizes all virtual tasks before it gets blocked. Otherwise, the semantics of `pacWait` would be effectively changed, since some of the virtual tasks might never become active (if other tasks do not terminate).

Case 1 is handled by the scheduler code executed in every virtual processor. Case 2 is managed by the `pacShell` function that every PACLIB task executes. Cases 3 and 4 are covered by the `pacWait` function. Consequently no change in the user code is necessary, i.e. the management details of virtual tasks are entirely hidden from the user. The extra runtime overhead in `pacShell` and `pacWait` consists of a simple test operation.

We will now discuss some of the consequences of our scheduling scheme for virtual tasks:

1 A real task is only created (unless explicitly by `pacStart`) when a processor would idle (or the semantics of the program would be changed) otherwise. The application

of virtual tasks may therefore essentially reduce (a) the total number of real tasks ever created and (b) the maximum number of real tasks that exist at any moment.

2 Virtual tasks have lower priority than real tasks. While the available processor time is fairly scheduled among all real tasks on a preemptive basis, a virtual task becomes only active when some real task or some processor has no other choice (in particular if there is nothing else to do).

3 A task is only terminated when there is no more virtual task to be executed, hence the workspace of a task is reused as long as possible. This may essentially improve the data locality of a program by the reduction of its overall memory requirements (yielding a higher cache hit ratio and less disk paging).

4 In a deterministic wait, a task is only blocked when the task connected to the result descriptor is currently active. Otherwise, the corresponding task either has already delivered the result or is still virtual and gets "inlined" into the current task.

5 In a non-deterministic wait, none of the participating tasks is virtual. If just one task were virtual, this would effectively change the behavior of the program since this task might never (or at least too late) become active.

Since a task may be blocked only on a `pacWait`, the last two arguments show that virtual tasks do not change the termination semantics of a Paclib program: a program with virtual tasks terminates if and only if the corresponding program with real tasks terminates.

## 4.7. Timings

We present the results of several benchmarks to illustrate the efficiency of the run-time system. The timings were performed on a lightly loaded Sequent Symmetry multiprocessor with 128 MB of physical memory and 20 processors i386 running at 16 Mhz.

### 4.7.1. Task Creation and Synchronization

The following table displays the time overhead of the Paclib operations `pacStart` and `pacWait` once for real tasks and once for their virtual counterparts (using `pacVirtual` instead). The figures in parentheses are normalized with respect to a function call with 4 arguments (= 1) which takes about 5 $\mu$s on our Sequent Symmetry.

| | Real task | Virtual task |
|---|---|---|
| `pacStart` | 260 $\mu$s (52) | 85 $\mu$s (17) |
| `pacWait` | 340 $\mu$s (68) | 85 $\mu$s (17) |

In case of the `pacWait` benchmark, a single task argument was used (deterministic wait). As shown in [Schreiner and Hong, 1993], every additional task argument (non-deterministic wait) costs another 60 $\mu$s.

These figures illustrate that the application of virtual tasks considerably reduces the creation overhead and also the synchronization overhead. However it is obvious that this overhead is still an order of magnitude larger than the overhead for a simple function call.

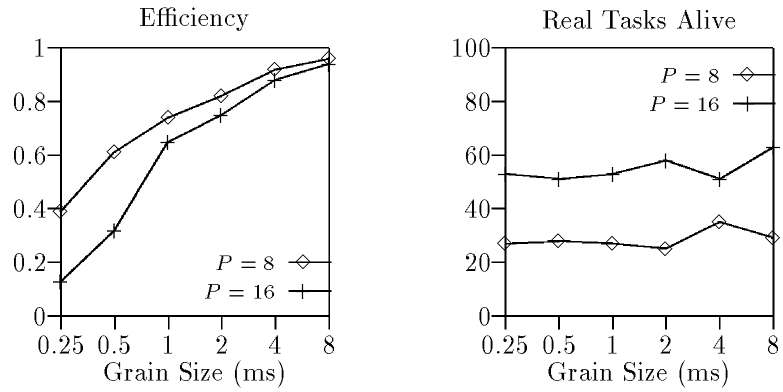
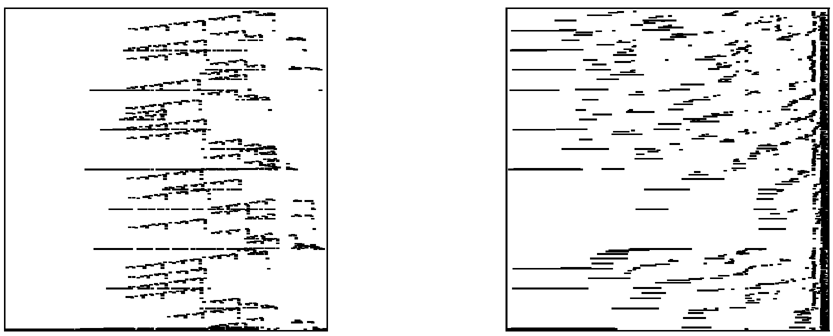



**Figure 11.** Minimum Granularity



Virtual Tasks 246 ms

Real Tasks 10219 ms

**Figure 12.** Profiling Data

#### 4.7.2. Minimum Granularity and Maximum Task Number

We use an artificial benchmark program to measure the minimum grain size that virtual PACLIB tasks permit. This program has a recursive divide-and-conquer structure where one recursive call is spawned as a virtual task and the task spins in the base case for a period of $t$ ms. The program is called with $N = 12$ i.e. 4048 virtual tasks are created. Figure 11 displays the results of this benchmark executed with $P$ processors:

1 The efficiency of the parallel program (compared to a corresponding sequential divide-and-conquer program) is acceptable for a grain size of $t \geq 1$ ms (which corresponds to 200 function calls). This still represents a significant constraint.

2 During the program run at most 30 respectively 60 real tasks were in existence at a time independently of the grain size $t$. Consequently the total memory consumption of the program was limited to about 1.5 MB (with a 32 KB stack per task).

We note that with the application of virtual tasks the number of real tasks is constant. This behavior (explained in detail in [Schreiner, 1994]) is a consequence of the changed scheduling order of virtual tasks (depth-first) in contrast to real tasks (breadth-first) when executing a divide-and-conquer algorithm. If the time required for the realization of a virtual task were the same as for the creation of a virtual task, only $P$ real tasks would execute at all. However, since the realization time is much longer, the generated task tree is unbalanced, which causes tasks to get blocked and additional virtual tasks to be realized to keep processors busy.

We also benchmarked a corresponding program with real tasks and a reduced stack size of 4 KB. This limited the maximum memory requirements of the program to 16 MB but still caused some disk paging. Consequently the parallel program was in all cases 50–100 times *slower* than the sequential program.

Figure 12 visualizes profiling data generated for both program variants ($n = 10$, $t = 1$ ms) where the horizontal axis denotes the runtime. In the left diagram the lines represent virtual tasks where the long ones (actually sequences of short segments) exhibit real tasks that execute many virtual tasks. In the right diagram, lines represent real tasks; the long ones show where tasks were slowed down by disk paging (note the different time scale). This drastically shows how the memory consumption of a program also influences the runtime.

### 4.7.3. Garbage Collection

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 two parameters (see Figure 13):

**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.

## 4.8. Performance Monitoring

For visualizing the dynamic aspects of a Paclib application, the kernel generates on demand profiling information during a program run. This information consists of trace records that are written on each context switch of a task into a buffer which is constantly flushed into a file. We have developed two batch programs, `pacgraph` and `pacutil`, that

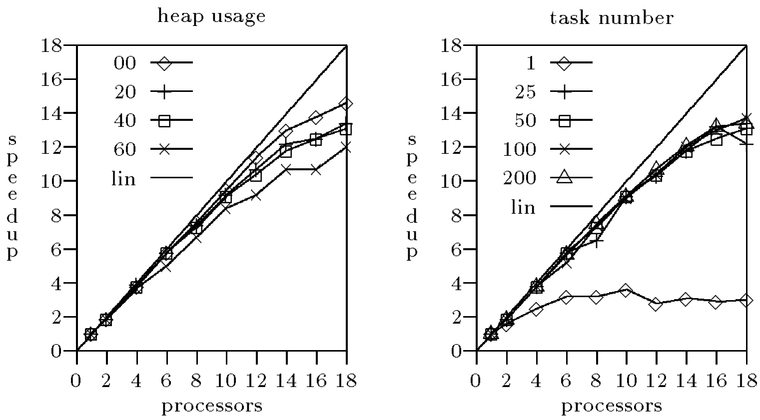



**Figure 13.** Garbage Collection

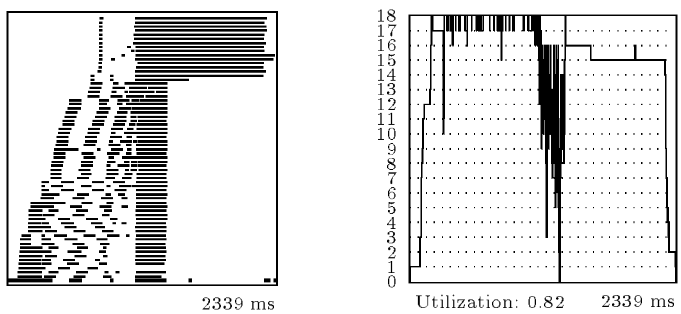



**Figure 14.** Performance Monitoring

generate from this information LaTeX pictures that visualize the execution of tasks and the utilization of processors, respectively.

Figure 14 shows the output of these tools for a sample PACLIB program with 18 processors in use. The left picture consists of a set of horizontal lines each of which represents a particular task and shows in the horizontal direction the moments during which the task was scheduled for execution. The right picture shows how many processors were saturated with work at any time.

For interactive analysis we have also developed an X11 based visualization tool: PACVIS [Hong et al, 1994] allows the programmer to monitor various aspects of a particular program run in various levels of detail: these aspects include events corresponding to individual tasks (like task creation and synchronization) as well as information about the overall system state (heap consumption or the number of tasks executing, ready, and blocked).

The output of these tools has proved to be a valuable help for the development and in particular for the fine-tuning of PACLIB applications.

## 5. Current Work

PACLIB is a system that is under continuous evolution. It has shown to be useful for a number of applications [Stahl, 1992], [Hong, 1993], [Hong and Loidl, 1994] but there are a variety of directions in which we are going to further develop it. First of all we will extend the kernel to run on networks of computer workstations and/or shared memory multi-processors. Then a compiler is being written for generating SACLIB/PACLIB code from (para-)functional specifications. Last but not least a complete re-implementation of the system in C++ is under way that uses arrays (instead of lists) as the basic data structure and utilizes modern software engineering concepts.

We would like to thank the anonymous referees for their constructive suggestions for clarifying and improving the presentation of the paper.

## References

B. Buchberger, G. Collins, M. Encarnación, 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, June 1992.

H.-J. Boehm, M. Weiser. Garbage Collection in an Uncooperative Environment. *Software — Practice and Experience*, 18(9):807–820, September 1998.

P. A. Buhr and R. A. Stroobosscher. The $\mu$System: Providing Light-weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software — Practice and Experience*, 20(9):929–964, September 1990.

G. E. Collins and R. Loos. ALDES/SAC-2 Now Available. *ACM SIGSAM Bull.*, 1982.

R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

H. Hong. Parallel Stability Analysis of Difference Algorithms. In *5th Scientific Workshop of the Austrian Center for Parallel Computation*, Wilhelminenberg Castle, Vienna, Austria, March 19–20, 1993.

H. Hong and H.-W. Loidl. Parallel Computation of Modular Multivariate Polynomial Resultants on Shared Memory Machine. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 — VAPP VI Third Joint Conference on Parallel and Vector Processing*, volume 854 of Lecture Notes in Computer Sciences, pages 325–336, Linz, Austria, September 26–28, 1994. Springer, Berlin.

H. Hong, W. Schreiner, and T. Fadgyas. Performance Analysis of Parallel Programs: The PACVIS Visualization Tool. Technical Report 94-07, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, January 1994.

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.

H. Kredel. From SAC-2 to Modula-2. In P. Gianni, editor, *Symbolic and Algebraic Computation, International Symposium ISSAC '88*, volume 358 of Lecture Notes in Computer Science, pages 447–455, Rome, Italy, July 4–8, 1988. Springer, Berlin.

H. Kredel. Computeralgebra on a KSR1 Parallel Computer. In *4th Supercomputing Day "Application of Supercomputers"*, Johannes Kepler University, Linz, February 3, 1994.

W. Küchlin. PARSAC-2: A Parallel SAC-2 Based on Threads. In S. Sakata, editor, *Eighth International Symposium on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC-8)*, volume 508 of *Lecture Notes in Computer Science*, pages 206 –217, Tokyo, Japan, August 1990. Springer, Berlin.

W. Küchlin. The S-Threads Environment for Parallel Symbolic Computation. In R. E. Zippel, editor, *Second International Workshop on Computer Algebra and Parallelism*, volume 584 of *Lecture Notes in Computer Science*, pages 1 – 18, Ithaca, USA, May 1990. Springer, Berlin.

W. Küchlin and N. Nevin. On Multi-Threaded List-Processing and Garbage Collection. In *3rd IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, December, 1991. IEEE Press.

W. Küchlin and J. Ward. Experiments with Virtual C Threads. In *4th IEEE Symposium on Parallel and Distributed Processing*, Arlington, TX, December, 1992. IEEE Press.

W. W. Küchlin and J. A. Ward. Experiments with Virtual Threads. Technical report, CIS Department, Ohio State University, Columbus, OH, September 1992.

L. Langemyr. Converting SAC-2 Code to Lisp. In J. H. Davenport, editor, *EUROCAL '87, European Conference on Computer Algebra*, volume 378 of Lecture Notes in Computer Science, pages 50–51, Leipzig, GDR, June 2–5, 1987. Springer, Berlin.

R. G. K. Loos. The algorithm description language ALDES (Report). *ACM SIGSAM Bull.*, 10(1):15–39, 1976.

E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *1990 ACM Symposium on Lisp and Functional Programming*, pages 185–197, Nice, France, June 27–29, 1990. ACM Press.

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

W. Schreiner. The Correctness of the PACLIB Kernel — A Case Study in Parallel Program Verification by Temporal Logic. Technical Report 93-13, RISC-Linz, Johannes Kepler University, Linz, Austria, March 1993.

W. Schreiner. Virtual Tasks for the PACLIB Kernel. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 — VAPP VI Third Joint Conference on Parallel and Vector Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 533–544, Linz, Austria, September 6–8, 1994. Springer, Berlin.

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.

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.

W. Schreiner and H. Hong. The Design of the PACLIB Kernel for Parallel Algebraic Computation. In J. Volkert, editor, *Parallel Computation — Second International ACPC Conference*, volume 734 of *Lecture Notes in Computer Science*, pages 204–218, Gmunden, Austria, October 4–6, 1993. Springer, Berlin.

S. Seitz. Algebraic Computation on a Local Net. In R. E. Zippel, editor, *Computer Algebra and Parallelism, Second International Workshop on Parallel Algebraic Computation*, volume 584 of *Lecture Notes in Computer Science*, pages 19–31, Ithaca, USA, May, 1990. Springer, Berlin.

V. Stahl. Solving a System of Linear Equations with Modular Arithmetic on a MIMD Computer. Technical Report 92-62, RISC-Linz, Johannes Kepler University, Linz, Austria, October 1992.