The Design of the SACLIB/PACLIB Kernels*

Hoon Hong and Andreas Neubacher and Wolfgang Schreiner

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

Abstract. This paper describes the design of the kernels 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 lightweight concurrency, non-determinism, 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 [3]: 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 of these two systems. 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 [11]. 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 [1]. Other people have worked on translating SAC-2 into languages such as LISP and MODULA-2 [10, 6]. 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 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 [5]. The kernel of PACLIB was designed in

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

order to fulfill the following requirements: upward compatibility to the SACLIB kernel, high-level parallel programming model, light-weight concurrency, nondeterminism and speculative parallelism, communication by heap references, and parallel garbage collection. For this purpose, we adapted the μSYSTEM package [2] that supports light-weight concurrency on shared-memory multiprocessors and workstations running under UNIX. The current kernel has been implemented on a 20 processor SEQUENT SYMMETRY computer.

2 The SACLIB Kernel

This section gives an overview of the kernel of SACLIB [1], the C version of the computer algebra library SAC-2 [3, 11].

2.1 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.

2.2 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.

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

2.3 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. Note

```
#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);
}
```

Fig. 1. A sample program.

that the standard `main` routine used in C is replaced by `sacMain`, which causes all the necessary initializations to be done before the main program is executed. 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 using `sacMain` it is also possible to call the initialization function explicitly at some later point in the program, as shown in Figure 3. Here

```
#include "saclib.h"

Word    F = NIL;

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

Fig. 2. Fragment of a sample program using global variables.

`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`. 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 an integer constant.
- *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.
- 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

```
#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(...);
}
```

Fig. 3. Embedding SACLIB in a subroutine.

additional memory for storing the type of an object. Its influence on garbage collection is made clear in the following sections.

2.4 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`). Figure 5 gives a graphical representation of the cell structure for a sample list, with arrows representing list handles.

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

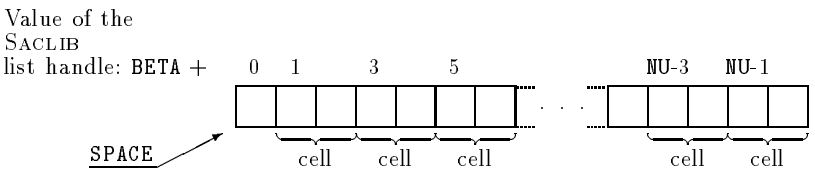


Fig. 4. The SPACE array.

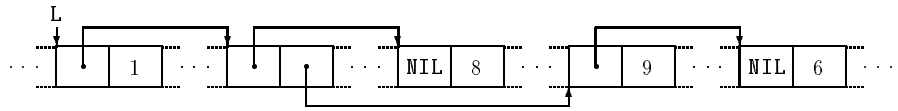


Fig. 5. The cell structure of the list L = (1, (9, 6), 8).

an atom or a list handle. Note that list handles can only be odd values between BETA and BETA + NU.

The Words of a cell addressed by a list handle L are SPACE[L − BETA] and SPACE[L − BETA + 1]. The use of the C programming language allows us to simplify these computations, by setting the C pointers SPACEB and SPACEB1 to the memory addresses of SPACE[−BETA] and SPACE[−BETA + 1], respectively. This is used by the functions FIRST(L), which returns SPACEB1[L], and RED(L), which returns SPACEB[L], thus avoiding time-consuming address computations.

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 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.
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.

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.

3 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 [14, 15].

3.1 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 [12].

3.2 Overall Design

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

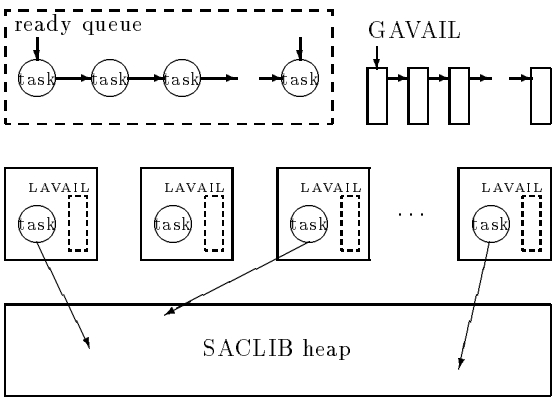


Fig. 6. PACLIB Kernel Design

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.
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.

3.3 Programming Interface

```
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 */
}
```

Fig. 7. Parallel Integer Multiplication

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.

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

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

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

returns the result *value* of one of the *tasks* and writes the handle of the delivering task into the location *tptr*. If all denoted tasks are still active, `pacWait` blocks

until one task terminates and then delivers the result of this task. There exist several variations of this function: `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 [5].

3.4 Task Management

The task management of the PACLIB kernel is built on top of the μSYSTEM [2]. This free software package consists of a library of C functions supporting light-weight concurrency on shared memory multiprocessors and UNIX workstations. The μSYSTEM 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 μSYSTEM task management is quite efficient and allows the utilization of rather *fine-grained parallelism*.

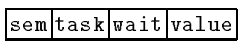
We applied and considerably extended the features of this package in order to implement the PACLIB parallel programming model which is higher-level and more general than that provided by the μSYSTEM. There were two major problems that had to be solved:

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

The implementation of `pacWait` required the development of a fairly complicated algorithm whose description is outside the scope of this overview paper. The details of this algorithm can be found in [12], its correctness has been formally verified in [13].

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 μSYSTEM) tasks are not first-order objects any more.

Therefore we decided to split the description of a task into two separate entities: a *task descriptor* and a *result descriptor*. *task* is then a reference to the result descriptor which is a SACLIB list with the following elements:



- `sem` is a pointer to a semaphore that provides mutual exclusion on the descriptor.
- `task` is a pointer to the workspace of the task that computes the result. 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.

On a call of `pacStart`, a result descriptor and a task descriptor are allocated and mutually linked. This new task executes a function `pacShell` that calls the denoted function with the corresponding arguments. When the function returns, `pacShell` stores the result value in the result descriptor and awakes all tasks blocked on this descriptor. Finally, `pacShell` deallocates its task descriptor and the task terminates.

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

3.5 Heap Management

The PACLIB kernel provides an operation `COMP` with the same interface and 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`. 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 [14] 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).

3.6 Performance Monitoring

For visualizing the dynamic aspects of a PACLIB program, the kernel optionally generates 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 programs,

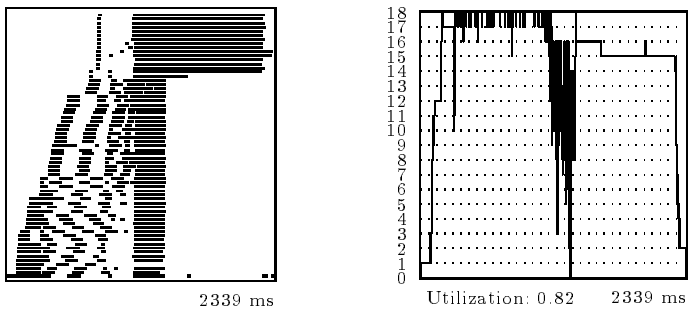


Fig.8. Performance Monitoring

`pacgraph` and `pacutil`, that generate from this information LaTeX pictures that visualize the behavior of the program and the utilization of the processors during the program run, respectively.

Figure 8 shows the output of these tools for some PACLIB program with 18 processors used. The left picture consists of a set of horizontal lines each of which represents one task and shows in the horizontal direction the times during which the task was scheduled for execution. The right picture shows how many processors were saturated with work during each moment of the program run. The output of these visualization tools has been proved to be a valuable help for the tuning of PACLIB programs.

3.7 Future Work

Currently, we are working on the integration of the following features into PACLIB:

1. **Task Priorities:** The user will be given the possibility of controlling the task scheduling mechanism be the introduction of different task priorities.
2. **Virtual Tasks:** `pacStart` will create "virtual" tasks that only turn into "real" tasks when processors are not saturated with work. The efficiency of task creation should increase and the space consumption of tasks should significantly drop.
3. **Visualization Environment:** An X11-based visualization environment is developed that will allow the programmer to learn in an interactive way many additional aspects of the dynamic behavior of a PACLIB program (e.g. memory consumption).
4. **Functional Language Compiler:** A compiler is developed that in a semi-automatic fashion generates parallel PACLIB code from high-level functional specifications.

A variety of parallel algebraic algorithms has already been implemented using the PACLIB kernel [16, 4].

3.8 Related Work

Finally, we would like to point out two projects that preceded and inspired our work:

- **Parsac-2** [7] has been the first parallel variant of SAC-2. The system is based on the *threads* concept of the operating system MACH and has by some of its features (e.g. light-weight concurrency and *virtual tasks* [9]) stood sponsor to the PACLIB kernel.
  The major differences between both systems is the heap management scheme: each PARSAC-2 thread owns a local set of heap pages from where it can allocate new cells and which may be extended from a global page set. Since free lists are attached to threads, the `COMP` operation needs one indirection to determine the head of the current free list (in the PACLIB kernel the head has the same location in the virtual address space of each processor).
  PARSAC-2 does not use a global garbage collection mechanism but it has introduced the concept of *preventive garbage collection* [8]: a terminating thread copies a SACLIB result structure into the page set of the parent thread (in contrast to PACLIB tasks that just return heap references) and frees its own page set.
  This heap management scheme is the reason for another major difference to the PACLIB kernel: PARSAC-2 threads are not first-class objects, since the result of a thread may be retrieved only once (by the parent task). Threads may be aborted but there is no PARSAC-2 equivalent to the PACLIB support of non-determinism.
- **Network-Sac** [17] is a parallel version of SAC-2 for computer networks and the coarse-grain parallelization of algebraic programs. Each 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 representation, transferred over the network and then transformed back into the internal representation. Since each algorithm server uses a separate heap with only local references, no global garbage collector is required.
  The concept of *multibags* is applied to express non-deterministic computations; futures (i.e. task references) are put into a multibag, any access to the multibag then returns the first available result. However, tasks are not really first-class objects (futures must not be passed as arguments to other tasks) and there is no possibility to abort tasks.

References

1. B. Buchberger, G. Collins, M. Encarnation, H. Hong, J. Johnson, W. Krandick, R. Loos, A. Mandache, A. Neubacher, and H. Vielhaber. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

2. P. A. Buhr and R. A. Stroobosscher. The μSystem: Providing Light-weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software — Practice and Experience*, 20(9):929–964, September 1990.
3. G. E. Collins and R. Loos. ALDES/SAC-2 now available. *ACM SIGSAM Bull.*, 1982.
4. H. Hong. Parallel Computation of Modular Multivariant Polynomial Resultants on Shared Memory Machine. In *Submitted to the 2nd International Conference of the Austrian Center for Parallel Computation*, Gmunden, Austria, October 4–6, 1993.
5. H. Hong, W. Schreiner, A. Neubacher, K. Siegl, H.-W. Loidl, T. Jebelean, and P. Zettler. PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992. Also: Technical Report 92-9, ACPC Technical Report Series, Austrian Center for Parallel Computation, July 1992.
6. H. Kredel. From SAC-2 to Modula-2. In *Proceedings of ISSAC*, pages 447–455. Springer, Berlin, 1988. Volume 358 of Lecture Notes in Computer Science.
7. W. Küchlin. The S-Threads Environment for Parallel Symbolic Computation. In *Computer Algebra and Parallelism, Second International Workshop*, pages 1–18, Ithaca, USA, May, 1990. Springer, Berlin. Volume 584 of Lecture Notes in Computer Science.
8. 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.
9. 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.
10. L. Langemyr. Converting SAC-2 Code to Lisp. In *Proceedings of EUROCAL 87*, pages 50–51. Springer-Verlag, 1989.
11. R. G. K. Loos. The Algorithm Description Language ALDES (Report). *ACM SIGSAM Bull.*, 10(1):15–39, 1976.
12. W. Schreiner. The Design of the PACLIB Kernel. Technical Report 92-33, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.
13. W. Schreiner. The Correctness of the PACLIB Kernel — A Case Study in Parallel Program Verification by Temporal Logic. Technical report, RISC-Linz, Johannes Kepler University, Linz, Austria, 1993. To appear.
14. W. Schreiner and H. Hong. A New Library for Parallel Algebraic Computation. In *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, Virginia, March 22-24, 1993. Also: Technical Report 92-73, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.
15. W. Schreiner and H. Hong. PACLIB — A Library for Parallel Algebraic Computation on Shared Memory Computers. In *Parallel Systems Fair at the Seventh International Parallel Processing Symposium*, Newport Beach, CA, April 13–16, 1993.
16. W. Schreiner and V. Stahl. The Exact Solution of Linear Equation Systems on a Shared Memory Multiprocessor. In *Submitted to the PARLE 93*, Munich, Germany, June 14–18, 1993.
17. S. Seitz. Algebraic Computation on a Local Net. In *Computer Algebra and Parallelism, Second International Workshop*, pages 19–32, Ithaca, USA, May, 1990. Springer, Berlin. Volume 584 of Lecture Notes in Computer Science.

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