PACLIB User Manual*
(Version 1.17/2)

Hoon Hong Wolfgang Schreiner Andreas Neubacher Kurt Siegl
Hans-Wolfgang Loidl Tudor Jebelean Peter Zettler

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria

October 22, 1992

Abstract

This paper describes the user interface of PACLIB, a new library for parallel algebraic computation on shared memory computers. PACLIB has been developed as a professional tool for the simple design and efficient implementation of parallel algorithms in computer algebra and related areas. It provides concurrency, shared memory communication, non-determinism, speculative parallelism, streams and pipelining and a parallelized garbage collection. PACLIB has been implemented on a Sequent Symmetry multiprocessor and is portable to other shared memory machines and workstations.

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

1 Introduction

This paper describes the user interface of PACLIB, a new library for parallel algebraic computation on shared memory computers. PACLIB has been developed as a professional tool for the simple design and efficient implementation of parallel algorithms in computer algebra and related areas.

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

1. **Saclib** [Collins *et al.*, 1992] is a large library of algebraic algorithms written in the C language. These algorithms are based on a list-processing kernel that provides automatic garbage collection. Furthermore, the package supports exact arithmetic with arbitrary precision integers and rationals.

2. The **uSystem** [Buhr and Stroobosscher, 1990] is a package that provides light-weight concurrency on mono-processor workstations and shared memory multiprocessors running under the UNIX operating system. The μSYSTEM provides an efficient management of *tasks* i.e. light-weight parallel processes.

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

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

- **Shared Memory Communication:** All data can be *shared* among tasks, i.e. task arguments and task results are passed by reference. No copying of data structures is required.

- **Non-Determinism:** A task may simultaneously wait for the results of various other tasks. The further computation depends on which task delivers its result first.

- **Task Abortion:** If by the result of one task the result of another task is not required any more, the superfluous subtask may be explicitly aborted.

- **Streams and Task Pipelines:** Tasks may be connected by *streams* of data and work in an overlapping fashion (pipelining).

- **Parallel Garbage Collection:** The garbage collector is based on a *mark and scan* scheme where the marking of of used cells and the reclamation of free cells are both parallelized.

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

This report is intended as a readable but comprehensive reference manual for any user of PACLIB. It explains how the system can be *used*, not how it has been *implemented*. The internal structure and the implementation of PACLIB are described in [Schreiner, 1992]. We assume that the reader is familiar with the concepts of SACLIB; a description can be found in [Collins *et al.*, 1992].

2 A Sample Program

Before we describe the facilities of our system in more detail, a sample program shall demonstrate the main characteristics of a PACLIB program. In Figure 1 a parallel version of the school algorithm for the multiplication of two long integers a and b is presented.

The main idea of this program is to split b into subintegers of N digits each and to multiply a with each subinteger in parallel. `PIPROD` applies the PACLIB function `pacStart` to create n concurrent tasks executing the function `IPRODS` that multiplies a with b_0 and shifts the result by $i * N$ `BETA` digits[1]. The descriptor t of each task is stored in *tlist* for further reference.

After all tasks have been started, `PIPROD` iteratively waits for the results delivered by any two tasks in *tlist* and starts a new task that sums up the result integers by the function `ISUM`. Therefore, not only the multiplications but also the various addition operations are performed in parallel. Finally, there remains a single task left that returns the total result to the caller.

The main function `pacMain` reads two integers a and b, calls the function `PIPROD` and prints the result of the multiplication. The program includes the header file `paclib.h` that contains the prototypes of all PACLIB functions[2]. The program file `piprod.c` is compiled by the statement

```
pacc -o piprod piprod.c
```

that compiles the file, links it with the required libraries and creates the executable file `piprod`. The program can then be started by typing

```
piprod +p8 +t1
```

[1] The functions `IPRODS` and `ADVANCE` are actually not part of SACLIB.

[2] `paclib.h` itself includes the header file `saclib.h` that provides the definitions of all SACLIB functions.

```
#include <paclib.h>                    /* pac definitions */
#define N 20                           /* digits per task */

Word PIPROD(Word a, Word b) {          /* multiply a with b in parallel */

  Word l, n, i, b0, b1;
  pacTask t;
  pacTaskList tlist;

  l = LENGTH(b);                       /* assume b is less than a */
  n = l / N + (l % N != 0);            /* number of multiplying tasks */
  tlist = NIL;                         /* list of multiplying tasks */

  for (i = 0; i < n; i++) {            /* start multiplying task */
    ADVANCE(N, b, &b0, &b1);           /* get b[0..N-1] and b[N..]  */
    t = pacStart(IPRODS, 3, a, b0, i*N); /* start task a*b0*BETA**(i*N) */
    tlist = COMP(t, tlist);            /* remember new task in tlist */
    b = b1;                            /* rest of b */
  }

  for (i = 0; i < n-1; i++) {          /* start adding tasks */
    b0 = pacWaitListRm(&tlist);        /* get result and remove task */
    b1 = pacWaitListRm(&tlist);        /* get result and remove task */
    t = pacStart(ISUM, 2, b0, b1);     /* start adding task */
    tlist = COMP(t, tlist);            /* remember new task in tlist */
  }

  return(pacWaitListRm(&tlist));       /* return final result */

}

int pacMain(int argc, int *argv[]) {   /* pac main function */

  Word a, b;

  SWRITE("a = "); IREAD(a);
  SWRITE("b = "); IREAD(b);
  SWRITE("a*b = "); IWRITE(PIPROD(a, b));

}
```

Figure 1: Parallel Integer Multiplication

```
--------------------------------------------------------------------
PAC Parallel Algebraic Computation V1.13 (c) 1992 RISC-Linz
8 processors, 500000 cells, 50 free lists, 32000 bytes stack size.
--------------------------------------------------------------------
a = ...
b = ...
a*b = ...
--------------------------------------------------------------------
0 garbage collections, 0 cells and 0 arrays reclaimed.
Initialization time (TI):        2990 ms.
Garbage collection time (TG):       0 ms.
Actual execution time (TE):      2337 ms.
Total program time (TI+TG+TE):   5327 ms.
--------------------------------------------------------------------
```

Figure 2: PAC Program Output

where the command line argument `+p8` says that 8 processors shall be applied for the program and `+t1` triggers the generation of timing information. The output then looks similar to that presented in Figure 2.

The PACLIB timing statistics is written in binary form into a file `pac.dat`. The command

```
pacgraph 250 250 <pac.dat >graph.tex
```

visualizes these data by generating a LaTeX picture of size 250*250 points as displayed in Figure 3: each horizontal line represents one task and shows the time the task was really active. Similarly, the command

```
pacutil 200 200 <pac.dat >util.tex
```

generates from `pac.dat` a LaTeX picture that shows the utilization of the processors during the program run (see Figure 4).

3 Program Structure, Compilation and Execution

A *function* is a (piece of a) program that describes a sequence of statements to be executed by a *task*. Tasks are the units of concurrency. Each task has a separate state and a separate thread of control: it executes a function simultaneously to and

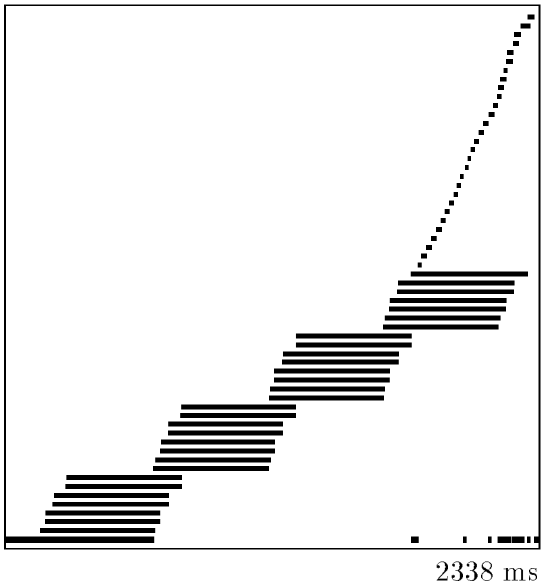

Figure 3: Program Visualization

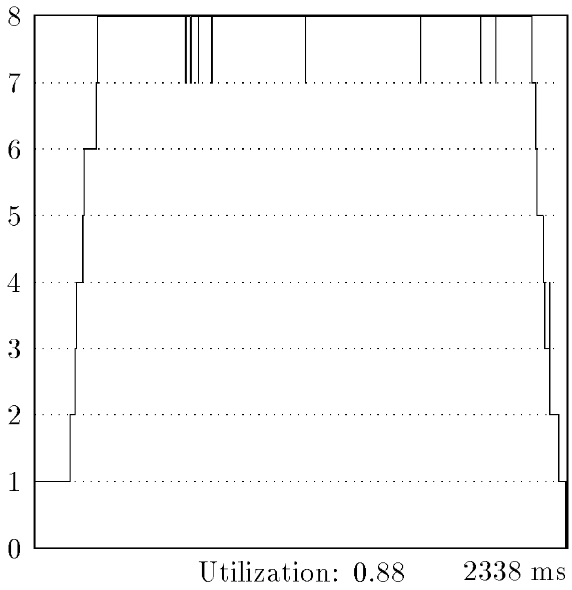

Figure 4: Program Utilization

independent of each other task. *Processors* are (physical) units that execute tasks. The number of tasks is independent of the number of processors; an automatic scheduler distributes the available processor time among all active tasks in a fair (and for the user invisible) way[3] .

Initially, there is only one task executing the function `pacMain`. This function (that is a replacement of the C function `main`[4]) must be provided by the user in a program file of the following form:

```
#include <paclib.h>
...
int pacMain(int argc, int *argv[]) {
  int code;
  ...
  return(code);
}
```

argc denotes the number of command line arguments stored in *argv* where all PACLIB (and all SACLIB) command line arguments have been already removed. The returned integer value *code* represents the result value of the main task and therefore of the whole program.

A PACLIB program file must be compiled and linked by one of the commands `pacc` or `paco` or `pacd` (that all take the usual `cc` command line arguments):

1. `pacc` compiles a program without optimization and links an unoptimized version of SACLIB to the program.
2. `paco` optimizes the program and links it with an optimized version of SACLIB. This command will generate the fastest version of a PACLIB program.
3. `pacd` generates a PACLIB program with debug information and links it with a debug version of SACLIB. No optimization is performed at all.

Each program can be used for compiling a program file (then only the degree of optimization of the compiled file is affected) as well as for linking (then the degree of optimization of SACLIBis affected).

For instance, the single statement

```
pacc -o main main.c aux.c
```

has the same effect as the statement sequence

[3]In PACLIB. we denote by processors actually *virtual* processors that are simulated by UNIX processes (see Section 10).

[4]`main` itself is declared in the PACLIB kernel.

```
pacc -c main.c
pacc -c aux.c
pacc -o main main.o aux.o
```

In addition to the usual SACLIB command line arguments (see [Collins *et al.*, 1992]), the following PACLIB arguments can be used:

1. `+p`*procnum* says that *procnum* processors shall be applied for the program (default 1).
2. `+s`*stacksize* says that a workspace of *stacksize* bytes shall be allocated for each PACLIB task (default 32000).
3. `+l`*length* determines the length of the local free list (list of available heap cells) for each processor (default 10000).
4. `+t1` triggers the generation of time information.

For instance, the program call

```
main +p16 +s8000 +l10000 +t1
```

specifies that 16 processors shall be used, that each task has a workspace of 8000 bytes, that each local free list consists of 10000 cells and that timing information is to be generated. If the timing facility is turned on, the PACLIB kernel generates after each task switch a statistics record that says during which time slice the task was executed. By applying the visualization tool

`pacgraph` *xdim ydim from to* `<pac.dat >`*file*

a LaTeX picture of size *xdim*ydim* points is generated and written into *file*. The picture consists of a set of lines each of which represents a task and shows in the horizontal direction when this task was active. Since a task may be preemptively descheduled, lines may have interruptions (i.e. only the actual execution times are displayed). The thickness of a line denotes its distance from the root of the task tree: the thickest line represents the task executing `pacMain`, somewhat thinner lines visualize the tasks directly created by the main task and so on.

from and *to* are optional arguments that denote *time marks* as generated by the function `pacMark` (see Section 9). The behavior of the program run is then visualized just in the time range that is bounded by these marks (and only those tasks are displayed that were active during this range). In a similar way, the command

`pacutil` *xdim ydim from to* `<pac.dat >`*file*

generates a LaTeX picture of the utilization of the processors during the program run.

4 Task Creation

A new task is created in the following way[5]:

task = `pacStart(`*fun*`,` n`,` arg_1`, ...,` arg_n`);`

`pacStart` spawns a new task that executes the function *fun* with n arguments $arg_1 \ldots arg_n$ and returns a result of type `Word`. *task* is an object of type `pacTask` that can be used for further reference to this task. `pacTask` is (as all PACLIB types) just a renaming of the SACLIB standard type `Word`; *task* can be therefore used at any position where a `Word` may occur. In particular it can be stored in a SACLIB list, passed to a function as argument and returned as a function result.

Please note that not the n but only the arg_i are passed as arguments to *fun*. Also note that *no* check performed whether *fun* has really n formal parameters; if *fun* actually expects $m > n$ arguments, the values of the remaining $m - n$ parameters are undefined (while if $m < n$ arguments are expected, all arg_i with $m < i \leq n$ arguments are ignored).

For instance, a new task that adds two long integers a and b is started as follows:

```
Word ISUM(Word, Word);
pacTask task;
Word a, b;
...
task = pacStart(ISUM, 2, a, b);
```

There is a second form of task creation

task = `pacStartList(`*fun*`,` *arglist*`);`

where the n arguments are packed into a single SACLIB list *arglist*, i.e. `FIRST(`*arglist*`)` denotes the first function argument, `FIRST(RED(`*arglist*`))` the second and so on. This form of task creation is especially useful if a function pointer *fptr* is passed to pacStartList where the number of arguments depends on the arity of the concrete function referenced by *fptr*, e.g.

[5] The exact prototypes of all PACLIB functions are given in Appendix B.

```
Word ISUM(Word, Word);
pacFunction fptr;
Word alist, a, b;
pacTask task;
...
fptr = ISUM;
alist = LIST2(a, b);
...
task = pacStartList(fptr, alist);
```

5 Communication and Synchronization

The result value of a task is an object of type `Word` that is delivered by the `return` statement of the top-level function of the task. E.g. if a new task is created by

```
Word ISUM(Word, Word);
Word a, b;
pacTask task;
...
task = pacStart(ISUM, 2, a, b);
```

then the result value of the task is the value returned by the function below:

```
Word ISUM(Word x, Word y) {
  Word z;
  ...
  return(z);
}
```

A task receives the result value of other tasks by the statement

value = `pacWait(`*taskptr*`,` n`,` $task_1$`,` $\ldots task_n$`);`

`pacWait` waits for a result to be delivered by one of the n tasks $task_1 \ldots task_n$ and returns this result as *value*. If all of these tasks are still active, the current task is blocked until the first task terminates and returns a result. *taskptr* is either `NULL` (in this case it is ignored) or it is the address of a variable that receives the identifier of the task whose result *value* is delivered. E.g.

```
Word ISUM(Word, Word);
Word a, b, c, d, r;
pacTask t1, t2, tr;

t1 = pacStart(ISUM, 2, a, b);
t2 = pacStart(ISUM, 2, c, d);
r = pacWait(&tr, 2, t1, t2);
```

waits for the first result to be delivered by the tasks t_1 or t_2. If a task needs to wait only for the result of a single other task, the more special statement

ident = `pacWaitOne(`*task*`);`

can be used.

A second form of task synchronization is provided by

value = `pacWaitList(`*taskptr*`, `*tasklist*`);`

where *tasklist* is a `pacTaskList`, i.e. a list of task identifiers (which must not be empty). `pacWaitList` returns the result value of one of the tasks whose identifiers are stored in *tasklist*. This statement is especially useful, if the number of tasks is not fixed.

Please note that after a call of `pacWaitList` the identifier of the terminated task has *not* been removed from *tasklist*. This can be done "manually" or by the function

value = `pacWaitListRm(`*tlptr*`);`

that destructively updates the `pacTaskList` at the location *tlptr* by discarding the identifier of the terminated task.

6 Task Termination

A task terminates and deallocates its workspace as soon as it has returned its result (even if no other task waits for this result yet). However, there are cases when we want to abort the execution of a task since its result is not required any more and we do not want the task to consume processor time and memory resources. Tasks can be terminated by statements of the form

`pacStop(`n`, `$task1_1$`, `$\ldots task_n$`);`

that stops the execution of the n tasks $task_1 \ldots task_n$. If one of these tasks has already terminated and delivered its result, it is not affected by `pacStop` any more. If a task is aborted, all subtasks created by this task are aborted, too.

```
Word FORALL(Word l, Word (*p)()) {     /* (forall x in l)(p(x)) */

  pacTask t;
  pacTaskList tlist;

  tlist = NIL;                         /* no tasks yet */
  while (!ISNIL(l)) {                  /* traverse whole list */
    t = pacStart(p, 1, FIRST(l));      /* start new task p(x) */
    tlist = COMP(t, tlist);            /* remember task in tlist */
    l = RED(l);                        /* rest of list */
  }

  while (!ISNIL(tlist))                /* traverse all tasks */
    if (!pacWaitListRm(&tlist)) {      /* task returned FALSE */
      pacStopList(tlist);              /* stop remaining tasks */
      return(0);                       /* return FALSE */
    }

  return(1);                           /* all tasks returned TRUE */

}
```

Figure 5: Speculative Parallelism

Any task can terminate any other task (provided that it knows its task identifier). However, if a task t_1 waits on the result of a task t_2 that is stopped by another task t_3, t_1 will be blocked forever. `pacStop` should be therefore not used if several tasks know the identifier of the affected task.

If only one task is to be terminated, the more special form

`pacStopOne(`*task*`);`

can be used.

Another form of task termination is provided by the function

`pacStopList(`*tasklist*`);`

that stops all tasks whose identifiers are stored in *tasklist*.

An application of explicit task abortion is *speculative parallelism*. Consider the function `FORALL` depicted in Figure 5. It tests whether a predicate p is true for all elements on a list l by starting concurrent tasks for the test operation of each element of l. If all of these tasks return 1 (i.e. true), `FORALL` returns 1 itself. However, if only one of the tasks returns 0 (i.e. false), the result of `FORALL` is 0 and the remaining tasks can be aborted.

7 Streams and Task Pipelines

Besides task arguments and the task results, PACLIB supports an additional and very flexible form of task communication by *streams*: a stream is a buffered communication channel by which a task can transmit a sequence of values to another task (or even to a set of other tasks). A stream can have one of three states:

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

A stream that is not *closed* is called *open*.

The usual application of streams (that are first-order PACLIB objects) is the following: an *empty* stream s_0 is created and passed as an argument to two new tasks. The one task (the *producer*) changes the state of the stream from *empty* to $\langle v_0, s_1 \rangle$ where v_0 is a SACLIB object to be transmitted and s_1 is a new and *empty* stream. The other task (the *consumer*) receives the value v_0 and the stream s_1 that can be used for the transmission of another value. If the consumer tries to receive a value while s_0 is still in the *empty* state, the task is automatically blocked.

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

A new `pacStream` is created by

```
stream = pacOpen();
```

that delivers the identifier of an *empty* stream. Streams may be *closed* by

```
pacClose(n, s1, ..., sn);
```

that sets the state of n *empty* streams $s_1 \ldots s_n$ to *closed*. For closing a single stream, the simplified version

```
pacCloseOne(stream);
```

may be used where *stream* refers to an *empty* stream. The function

```
pacCloseList(slist);
```

closes all *streams* in the `pacStreamList` *slist* (that must be all *empty*). A task may put an element into a stream by

```
pacPut(sptr, value);
```

sptr must be the location of an *empty* stream, *value* may be any object of type `Word`. `pacPut` sets the state of the stream to $\langle$*value, new*$\rangle$ where *new* is a new and *empty* stream. Furthermore, the location *sptr* is set to *new*, i.e. any further application of *sptr* will actually refer to *new*.

The following program shows how a task creates a stream and fills it with values:

```
pacStream s;
...
s = pacOpen();
for (i = 0; i <100; i++)
  pacPut(&s, f(i));
pacClose(s);
```

The following function can be used to connect two streams:

```
pacPutRest(stream, value, new);
```

stream must be an *empty* stream; *value* may be any object of type `Word`. `pacPutRest` sets the state of *stream* to $\langle$*value, new*$\rangle$ where *new* may be any previously created stream (that is not necessarily *empty* any more).

Elements may be retrieved from a stream by

```
done = pacGet(sptr, vptr, rptr, n, s1, ..., sn);
```

`pacGet` investigates the n streams $s_1 \ldots s_n$. If all streams are *closed*, the function returns 0 and the values of the locations *sptr*, *vptr* and *rptr* are undefined. If some streams are *open* but all of them are *empty*, `pacGet` blocks the current task until the state of some stream has changed.

```
Word producer(Word n, pacStream s) {  /* generate stream s of n objects */
  Word i;
  for (i = 0; i < n; i++)             /* s refers to empty stream */
    pacPut(&s, F(i));                 /* generate next element */
  pacCloseOne(s);                     /* close stream */
  return(1);                          /* terminate task */
}

Word consumer(pacStream s) {          /* consume stream s */
  Word v;
  while (pacGetOne(&v, &s))           /* get next element v of s*/
    G(v);                             /* process v */
  return(1);                          /* stream closed, terminate */
}

Word pacMain(int argc, char *argv[]) {/* producer/consumer structure */
  pacStream s;
  pacTask p, c;
  s = pacOpen();                      /* open shared stream s */
  p = pacStart(producer, 2, 100, s);  /* start producer with s */
  c = pacStart(consumer, 1, s);       /* start consumer with s */
  return(pacWaitOne(c));              /* wait for consumer */
}
```

Figure 6: Producer/Consumer Parallelism

If a stream s_i has the state $\langle$*first, rest*$\rangle$, the identifier s_i is written into the location *sptr*, the element *first* is written into the location *vptr* and the identifier of the stream *rest* is written into the location *rptr*. `pacGet` then returns 1. If one of *sptr*, *vptr* or *rptr* is `NULL`, the corresponding location is ignored.

If one is only interested in the state of a single stream, a simplified form of the function may be used:

done = `pacGetOne(`*vptr*`,` *sptr*`);`

`pacGetOne` returns 1 if the stream s at the location *sptr* is of the form $\langle$*first, rest*$\rangle$ and delivers the head element *first* into the location *vptr* (provided that *vptr* is not `NULL`). Furthermore, s is destructively updated by *rest*. If s is *closed*, `pacGetOne` returns 0 and the value of the location *vptr* is undefined. If s is *empty*, the current task is blocked until the state of the stream has changed.

```
int worker(pacStream ws, pacStream rs) {
  Word w, r;
  pacPut(&rs, COMP(ws, NIL));         /* request on result stream */
  while (pacGetOne(&w, &ws)) {        /* get work w from ws */
    r = WORK(w);                      /* compute result r */
    pacPut(&rs, COMP(ws, r));         /* return result and ws */
  }
  pacCloseOne(rs);                    /* close result stream */
  return(1);                          /* terminate task */
}

Word pacMain(int argc, char *argv[]) {
  pacStream ws, rs;
  pacStreamList rstreams, wstreams;
  Word wlist, rlist, result;
  wlist = GETWORK();                  /* get work list */
  wstreams = NIL;                     /* list of work streams */
  rstreams = NIL;                     /* list of result streams */
  for (i = 0; i < WORKERS; i++) {     /* start worker tasks */
    ws = pacOpen(); rs = pacOpen();   /* work and result stream */
    wstreams = COMP(ws, wstreams);    /* remember work stream */
    rstreams = COMP(rs, rstreams);    /* remember result stream */
    pacStart(worker, 2, ws, rs);      /* start worker task */
  }
  rlist = NIL;                        /* list of results */
  while (wlist) {                     /* still work left */
    pacGetListRm(&rw, &rstreams);     /* get worker result */
    ws = FIRST(rw); r = RED(rw);      /* work stream and result */
    if (!ISNIL(r)) rlist = COMP(r, rlist); /* store result */
    pacPut(&ws, FIRST(wlist));        /* deliver next piece of work */
    wlist = RED(wlist);               /* rest of task list */
  }
  for (i = 0; i < WORKERS; i++) {     /* no work, get final results */
    pacGetListRm(&rw, &rlist);        /* get worker result */
    ws = FIRST(rw); r = RED(rw);      /* work stream and result */
    if (!ISNIL(r)) rlist = COMP(r, rlist); /* store result */
    pacCloseOne(ws);                  /* close work stream */
  }
  result = COMBINE(rlist);            /* combine results */
}
```

Figure 7: Manager/Worker Parallelism

By streams, a *producer/consumer* form of parallelism can be implemented as depicted in Figure 6: the main task creates an empty stream and passes it to two tasks `producer` and `consumer`. `producer` computes new elements, puts them into the stream and finally closes the stream. The `consumer` iteratively reads elements from the stream and processes them. When all elements are processed, the task terminates.

done = `pacGetList(`*sptr*, *vptr*, *rptr*, *slist*`);`

is an equivalent to `pacGet` except that all streams are contained in the `pacStreamList` *slist*. Please note that neither *closed* streams are removed from *slist* nor the streams in *slist* are destructively updated. For this purpose, PACLIB provides the function

done = `pacGetListRm(`*vptr*, *slptr*`);`

that behaves similar to `pacGetList` except that *slptr* is the location of a stream list *slist*. If a stream *s* in *slist* becomes ⟨*first, rest*⟩, *s* is destructively updated by *rest*. Moreover, if some stream becomes *closed* it is destructively removed from *slist*.

By streams, also *manager/worker* schemes can be implemented as depicted in Figure 7: the main task starts a number of `worker` tasks that are connected to the manager by a work stream *ws* and a result stream *rs*. The manager splits the total work into small pieces and delivers a piece to each idle worker. As soon as a worker has delivered its result through *rs*, it receives new work by *ws*. The workers in Figure 7 actually return in *rs* cells that contain besides the result value the corresponding *ws* (such that the manager immediately knows where to send a new task to).

The use of streams gives the PACLIB programmer a larger flexibility in choosing the pattern of communication. However, it may also help to write more complicated programs that are harder to understand and harder to verify. For this reason, we strongly recommend to apply this concept with care and only in selected cases. For most algorithms, the functional form of task communication presented in the previous section is sufficient and yields to better structured parallel programs.

8 Buffer Interfaces

Streams are sufficient to express all patterns of communication between tasks. However, it requires some experience to formulate the following forms of communication in the pure stream model:

1. Several tasks are reading from the same stream; if one task reads a value from the stream, this value shall be removed such that no other task can receive it.
2. Several tasks simultaneously write values into the same stream; these values shall be non-deterministically merged.

```
Word breader(pacBuffer rb) {
  Word v;
  while (pacRead(&v, rb)) IWRITE(v);
}

Word sreader(pacStream s) {
  Word v;
  while (pacGetOne(&v, s)) IWRITE(v);
}

void parent(void) {
  pacStream s;
  pacBuffer rb;
  pacTask t1, t2, t3;
  Word i;
  s = pacOpen();
  rb = pacReadBuffer(s);
  t1 = pacStart(breader, 1, rb);
  t2 = pacStart(breader, 1, rb);
  t3 = pacStart(sreader, 1, s);
  for (i = 1; i < N; i++) pacPut(&s, i);
  pacClose(s);
}
```

Figure 8: Multiple Readers

3. The unprocessed part of the stream shall have bounded length; if the writer tasks are significantly faster than the readers, the writers shall be blocked.

The first two forms allow to transmit and to receive sequences of values in an indeterminate order, respectively. The third form of communication deals with the dynamic relationship between sending and receiving tasks.

In order to facilitate the application of these communication patterns, we have built a *buffer interface* on top of the streams model. This interface allows to apply a style of communication that is different from that provided by `pacPut` and `pacGet`.

The function

rbuf = `pacReadBuffer(`*stream*`)`

creates a read buffer interface *rbuf* that is connected to *stream* and can be used by different tasks for the concurrent reading of values from the stream. Vice versa, the function

```
Word breader(pacBuffer rb) {
  Word v;
  while (pacRead(&v, rb)) IWRITE(v);
}

Word bwriter(pacBuffer wb) {
  Word i, v;
  for (i = 1; i < 100; i++) pacWrite(wb, v);
}

void parent(void) {
  pacStream s;
  pacBuffer rb, wb;
  pacTask t1, t2, t3, t4;
  s = pacOpen();
  pacBoundedBuffer(s, 10, &rb, &wb);
  t1 = pacStart(breader, 1, rb); t2 = pacStart(breader, 1, rb);
  t3 = pacStart(bwriter, 1, wb); t4 = pacStart(bwriter, 1, wb);
}
```

Figure 9: Bounded Buffer

wbuf = `pacWriteBuffer(`*stream*`);`

creates a write buffer interface *wbuf* for *stream* (which must be *empty*) that can be used by different tasks for the concurrent writing of values into *stream*. In this case, it is not any more allowed to apply `pacPut` on *stream*.

`pacBoundedBuffer(`*stream*`,` *n*`,` *rptr*`,` *wptr*`);`

creates a pair of a read buffer interface *rbuf* and a write buffer interface *wbuf* that are connected to *stream*. The buffer references are written into the locations *rptr* and *wptr*, respectively. It is not any more allowed to apply `pacPut` on *stream* .

wbuf and *rbuf* are now related in the following way: if the difference of the number of write operations on *wbuf* to the number of read operations on *rbuf* equals n, any task will be blocked on an attempt to write a value into *wbuf*. The blocked task will be released only when a read operation on *rbuf* takes place.

There are two access functions for the buffer interface:

```
done = pacRead(vptr, rbuf);
```

reads a value from the stream connected to the read buffer *rbuf*, writes this value into the location *vptr* and removes the value from *rbuf*. Different tasks may use `pacRead` concurrently on the same read buffer; any value retrieved by one task will not be retrieved by any other. `pacRead` returns 0 if and only if the stream connected to *rbuf* is *closed*.

```
pacWrite(wbuf, value)
```

writes *value* into the stream connected to *wbuf* and releases all tasks blocked on that stream. `pacWrite` can be used by different tasks concurrently on the same write buffer *wbuf*, the values will be non-deterministically merged on the connected stream.

Figure 8 demonstrates the use of the read buffer interface. A parent task creates a stream *s* and a read buffer interface *rb* for *s* that is forwarded to two reader tasks *t1* and *t2*. The parent task now writes a sequence of values into *s* (since there is only one writer, it is not necessary to create a write buffer interface for *s* and use `pacWrite`). The values sent by the parent task are received by *t1* and *t2* via *rb*. However, each transmitted value will be received either by *t1* or by *t2* but not by both. On the contrary, the task *t3* will receive each value sent by the parent task since it directly listens on the stream *s*.

The use of multiple readers and writers with a bounded buffer is demonstrated in Figure 9. The main task allocates a bounded buffer interface for the stream *s* and forwards the read buffer *rb* to two reader tasks and the write interface *wb* to two writer tasks. Both writers now concurrently send values through *wb* that are merged in a non-deterministic way on the stream *s*. Both readers receive the transmitted values through the read interface *rb* but each value sent by one of the writers will be received by one of the readers only!

The use of the buffer interface gives a larger flexibility than the pure stream model. However, the comments at the end of the previous section also apply to this concept. Streams and buffers often lead to more complicated programs that are not necessarily more efficient than programs that restrict themselves to the purely functional communication model by task arguments and task results.

9 Miscellaneous Functions

9.1 Workspace Size

For each new task, a separate workspace has to be allocated that in particular holds the stack of the task. Each task has a separate default value for the size of workspace allocation used by `pacStart` and `pacStartList`. Initially, this value is the same for all tasks as specified by the command line argument `+s` (see Section 3). This default value may be changed by the command

```
pacStackSize(size);
```

which says that all tasks to be created by the current task will receive a workspace of *size* bytes. Moreover, these tasks will inherit the new value from the current task, i.e. their subtasks will also receive workspaces of this size.

9.2 Workspace Integrity

If a task executes a deeply recursive function, a stack overflow may occur. In this case, the program will crash or show some unpredictable behavior. The integrity of the current task stack can be checked by the function

```
pacVerify();
```

that aborts the program with an error message if a stack overflow has occurred for the current task. This check should be performed immediately after the declaration of the local variables of a function i.e. before any statement is executed, e.g.

```
int f(int x) {
  int l1, l2;
  pacVerify();
  ...
}
```

9.3 Program Termination

A PACLIB program terminates as soon as the task executing `pacMain` has returned its result value. However, any task may terminate the program by the statement

```
pacExit(code);
```

All tasks are then stopped and the integer value *code* is returned as the result of the program.

In case of an error, a program can be aborted by

```
pacAbort(format, ...)
```

where *format* is a format string in the style of the C function `printf`. The program is then aborted and the corresponding error message is printed, e.g.

```
Word m, n, r;
...
if (n == 0) pacAbort("Zero Division (%d, %d)", m, n);
r = m/n;
```

9.4 Time Marks

The function

`pacMark(`*number*`);`

associates the integer mark *number* with the particular moment when `pacMark` was called. Marks may be used to identify different time sections of a program run for the visualization tool `pacgraph` (see Section 3). Time marks must be generated in increasing (but not necessarily consecutive) order.

10 Parallel Program Development

Some hints on the pragmatics of parallel program development in PACLIB shall conclude this paper.

10.1 Processes and Processors

There exists an additional level between PACLIB tasks and physical processors: that of *virtual processors*. The PACLIB command line argument `+p` actually specifies how many virtual processors are to be applied for executing PACLIB tasks. Each virtual processor is implemented by a UNIX process that is scheduled by the operating system to a physical processor, i.e. we have a hierarchy

PACLIB task	
↓	PACLIB (μSYSTEM)
virtual processors	
↓	UNIX
physical processors	

If there are not more virtual processors than physical processors, the difference does not matter: the UNIX operating system will attach each virtual processor to an idle physical processor. However, if there are also other processes running, these processes will compete with the PACLIB virtual processors for computing time i.e. the virtual processors will not be active all the time. One should also consider that even on an unused machine always several processes (demons) are active. Consequently, one should always apply strictly less virtual PACLIB processors than physical processors are available.

By the shell command `ps`, a list of all UNIX processes is displayed[6] including all virtual PACLIB processors. For n virtual processors, there exist $n + 1$ UNIX processes (one process is always used for input/output). On most machines, there are also visual tools by which the processor load can be watched[7]

10.2 Number and Granularity of Tasks

How many tasks can be created in a PACLIB program? The answer is difficult and depends on several factors:

1. **Granularity:** the creation of a new task takes roughly 50 times longer than a simple function call. This is actually not very much and will not decrease the efficiency of the program provided that the task has a sufficiently large grainsize (say if the task runs at least 100 ms). In general, new tasks should be created only if the task creation overhead is significantly outweighed by the benefits of parallelism.

2. **Workspace:** each active task requires a separate workspace that essentially contains the stack of the task. The default size of this workspace is 32 KB[8]. Hence if 100 tasks are active at the same time, 3.2 MB of main memory are used; for 1000 tasks most machines will run out of memory.

3. **Processors:** The speedup of an application is determined by the number of available processors and their utilization. In general, 2–5 tasks per virtual processor should be generated in order to guarantee that no processor is idle at any time. More tasks will typically (due to task creation overhead) increase the runtime of the program.

As a rule of thumb, for the RISC-Linz 20 processor Sequent Symmetry with 128 MB of memory it may make sense to have up to a few hundred tasks active at a time. If a significantly larger number of tasks is required, one should try to combine several tasks into one or consider a *manager-worker* scheme as discussed in Section 7.

10.3 Debugging

Debugging of concurrent PACLIB programs is quite difficult, since debuggers like `dbx` get confused by the μSYSTEM task management. Some hints may help to simplify the debugging process:

---

[6] On the RISC-Linz Sequent Symmetry, use the command `ps -ef`.

[7] `/etc/monitor` and `xload` on the RISC-Linz Sequent Symmetry.

[8] Section 9 describes how to change this default size.

- **Read the Source:** The usual editing/running/debugging cycle can be significantly shortened, if the program source is printed on paper and read carefully two or three times. By this method, usually nine errors out of ten can be found without any additional help.
- **Check the Interfaces:** Many PACLIB functions allow a variable number of arguments. This prevents the compiler from checking the consistency of function applications with function prototypes. Since all PACLIB functions start with the prefix `pac`, you need to consider lines with this string only. In particular, you should check that in functions with a variable number of arguments the number of arguments n has not been omitted.
- **Debug the Sequential Parts:** From our experience, the PACLIB primitives are so simply to use that most programming errors actually occur in the sequential parts of the program. Try to establish the correctness of these parts (by test outputs or test runs in the sequential SACLIB) before you debug the parallel parts.
- **Localize Errors:** If the program fails with an error message like "segmentation violation", try to localize the point of failure. Try to reproduce the error by compiling the program with `pacd` and applying a single processor. If a `core` file is generated, call `dbx` and type `w` to find out in which function the failure occurred. If this does not deliver sensible results, introduce break points in your program (infinite loops that cannot be passed) in order to localize the error.
- **Test Outputs:** Include test outputs that show the values of characteristic variables at certain points of the program (the arguments of a task, the results delivered by tasks, ...). Compare the results with the expected ones and try to understand what is going on in your program. Also apply the timing facility of PACLIB to learn about the dynamical behavior of the program.

If all this does not help, there still is the possibility of an error in the PACLIB kernel itself. Contact the PACLIB maintainer for further steps.

10.4 Timings

The PACLIB version of the SACLIB `CLOCK` (that returns the current time in ms) has a turnaround time of roughly half an hour only. The reason for this is that the PACLIB `CLOCK` is based on a low level system clock with a resolution of 1 μs and a turnaround time of 2^{31} μs. It may therefore happen that the difference from the end time of a program to its begin time becomes negative. Assuming that the program runs less than half an hour, the timing can be corrected by adding $\frac{2^{31}}{1000}$ ms to the negative time value.

Alternatively, PACLIB provides a long time clock `LCLOCK` that has a practically infinite turnaround time. However, `LCLOCK` has only a resolution of 1 s, i.e. the ms returned by this function are always multiples of 1000. Depending on whether high accuracy or a long turnaround time is desired, either `CLOCK` or `LCLOCK` should be applied.

11 Acknowledgements

PACLIB was developed by the parallel computation group at RISC-Linz on an initiative of its leader Hoon Hong. The detailed design and the implementation of PACLIB were both performed by the author of this paper. Andreas Neubacher provided valuable support as the maintainer of SACLIB. This work was sponsored by the Austrian Science Foundation (FWF) in the frame of the research project S5302-PHY "Parallel Symbolic Computation".

A SACLIB Differences

A sequential SACLIB program should also run in PACLIB except the following modifications:

1. **Old Style:** PACLIB does not support the old style SACLIB functions. It is a pure implementation of the new SACLIB as described in [Collins *et al.*, 1992].
2. **Compilation:** A PACLIB program must be compiled and linked by the commands `pacc`, `paco` or `pacd`. The necessary SACLIB functions are automatically linked to the program; it is illegal to link SACLIB functions explicitly to a PACLIB program.
3. **Main Function:** The main function `sacMain` is replaced by `pacMain`. This function *must* be used, a manual initialization of the program is not possible.
4. **Command Line:** The SACLIB command line option `+T` is not supported. PACLIB provides the additional command line options `+p`, `+l`, `+s` and `+t`.
5. **Variables:** The global variables `AVAIL`, `GCAAVAIL`, `BACSTACK`, `EACSTACK`, `NPTR1` and `GCAPTR1` have become obsolete.
6. **Initialization:** The SACLIB functions `ARGSACLIB`, `BEGINSACLIB`, `ENDSACLIB` and `INFOSACLIB` are not available to the user. An explicit program initialization is not possible.
7. **Garbage Collection:** The SACLIB functions `GC`, `GCSI` and `MARK` are not available to the user. Garbage Collection cannot be explicitly triggered.
8. **Input and Output** is allowed by the official SACLIB functions (`IREAD`, `IWRITE`, ...) only. In particular, it is illegal to apply the standard C input/output functions.
9. **Memory Management** is allowed by the official SACLIB functions (`GCAMALLOC`, `GCAFREE`) only. In particular, it is illegal to apply the standard C functions `malloc` and `free`.

10. **Global Variables** are shared by all tasks. The declaration of global variables by `GCGLOBAL` must take place *only* in `pacMain` *before* any task has been started. The value of each global variable *must* be defined in `pacMain` and it must *not* be changed any more after a task has been started.

11. **Timings** The PACLIB version of the timing function `CLOCK` has a turnaround time of roughly 30 minutes. If longer time periods are to be measured, the function `LCLOCK` should be used that has a practically infinite turnaround time but is only precise up to whole seconds.

B PAC Function Prototypes

This appendix gives the exact prototypes of all PACLIB functions:

```
/* type definitions */
typedef Word pacObject;
typedef pacObject pacObjectList;
typedef pacObject pacTask;
typedef pacObject pacTaskList;
typedef pacObject (*pacFunction)();
typedef pacObject pacStream;
typedef pacObject pacStreamList;
typedef pacObject pacBuffer;


/* main function */
int pacMain(int argc, char *argv[]);


/* start tasks */
pacTask pacStart(pacFunction f, int n, /* pacObject arg1, arg2, */ ...);
pacTask pacStartList(pacFunction f, pacObjectList arglist);


/* wait for task results */
pacObject pacWait(pacTask *taskptr, int n, /* pacTask task1, task2, */ ...);
pacObject pacWaitOne(pacTask task);
pacObject pacWaitList(pacTask *taskptr, pacTaskList tasklist);
pacObject pacWaitListRm(pacTaskList *tlptr);


/* stop tasks */
void pacStop(int n, /* pacTask task1, task2, */ ...);
void pacStopOne(pacTask task);
void pacStopList(pacTaskList tasklist);
```

```
/* open a stream */
pacStream pacOpen(void);


/* close stream(s) */
void pacClose(int n, /* pacStream s1, s2, */ ...);
void pacCloseOne(pacStream stream);
void pacCloseList(pacStreamList slist);


/* put an element into a stream */
void pacPut(pacStream *sptr, pacObject value);
void pacPutRest(pacStream stream, pacObject value, pacStream new);


/* get an element from a stream */
int pacGet(pacStream s, pacObject *v, pacStream *r,
      int n, /* s1, s2 */ ...);
int pacGetOne(pacObject *v, pacStream *s);
int pacGetList(pacStream s, pacObject *v, pacStream *r,
      pacStreamList slist);
int pacGetListRm(pacObject *v, pacStreamList *slist);


/* create buffer interface for a stream */
pacBuffer pacReadBuffer(pacStream s);
pacBuffer pacWriteBuffer(pacStream s);
void pacBoundedBuffer(pacStream s, int n, pacBuffer *rb, pacBuffer *wb);


/* read from and write to a buffer */
int pacRead(pacObject *v, pacBuffer rb);
void pacWrite(pacBuffer wb, pacObject v);


/* generate a time mark */
void pacMark(int mark);
```

References

[Buhr and Stroobosscher, 1990]
Peter A. Buhr and Richard A. Stroobosscher.
The μSystem: Providing Light-weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX.

Software — Practice and Experience, 20(9):929–964, September 1990.

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

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