The C++ Interface to the STURM Multi-Processor Kernel

This is Edition 1.0, April 1994 for Version 1.0 of the STURM multi-processor kernel.

Hoon Hong, Andreas Neubacher, Wolfgang Schreiner, Volker Stahl

RISC-Linz Report Series No. 94-31

1 Introduction

The STURM library is a collection of packages written in C++ for solving algebraic constraints (equations, inequations).

The library is based on a runtime system (the *kernel*) that represents a portable interface to the underlying hardware and operating system. In particular the kernel simulates an infinite *heap* from which memory blocks of arbitrary size can be allocated. This is realized by a *garbage collection* mechanism which reclaims the memory occupied by no more used objects. The interface to the STURM kernel is documented in *The* STURM *Library Manual*.

In this manual we describe a variant of the STURM kernel designed for computers with multiple processors coupled by shared memory. This variant extends the interface of the standard kernel by a new type *task* and functions for task creation, communication, and synchronization. Tasks may be executed simultaneously and asynchronously by different processors. Every task may allocate memory from the heap that is shared among all processors. Any object stored in this heap may be accessed by any task.

The STURM multi-processor kernel is derived from the PACLIB kernel (see the *PACLIB User Manual*), a parallel variant of the SACLIB runtime system that is based on the uSystem task package (see *The uSystem: Light-Weight Concurrency under Unix*). The task management of the the STURM multi-processor kernel is therefore very similar to the PACLIB kernel (see *The Design of the PACLIB Kernel*). The memory management however is completely redesigned.

The STURM multi-processor kernel is implemented in C and provides a basic interface of C types and functions. A C++ shell is wrapped around this C interface. This manual documents the C++ variant, which is the one to be used in STURM programs. However, the C interface is of interest for people who want to use the multi-processor and memory management facilities of this kernel in plain C programs. For a description of the C interface, see section “Overview” in *The STURM Multi-Processor Kernel*.

The kernel is currently running on the Sequent Symmetry S81 shared memory multi-processor but should be portable to all other architectures supported by the uSystem.

2 A Sample Program

Below we give a sample C++ program which utilizes some features of the STURM multi-processor kernel:

```
#include <iostream.h>
#include "Sturm/Kernel.h"
#include "Sturm/Parallel.h"


double *
CrossProduct(double *A, double *B)
{
  int n = MEM_Number(double, A)
  double *M = MEM_NewInt(double, n*n);

  int i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      M[n*i+j] = A[i] * B[j];

  return M;
}


PAR_CreateStart2(double *, double *, double *)


int
STURM_Main(int argc, char *argv[])
{
  //
  // 1  Read inputs.
  //

  int m, n;

  cout << "Number of vectors = "; cin >> m;
  cout << "  Size of vectors = "; cin >> n;

  double **Vectors = MEM_New(double *, m);
  int i, j;

  for (i = 0; i < m; i++) {
    Vectors[i] = MEM_New(double, n);
    cout << "Vector " << i << '\n';
    for (j = 0; j < n; j++) {
      cout << "  Element " << j << " = "; cin >> Vectors[i][j];
    }
  }

  //
```

```
    // 2  Start CrossProduct computations in parallel.
    //

    Type_TaskList list = NULL;
    Type_Task t;

    for (i = 0; i < m-1; i++) {
      for (j = i+1; j < m; j++) {
        t = PAR_Start(CrossProduct, Vectors[i], Vectors[j]);
        list = PAR_Cons(t, list);
      }
    }

    //
    // 3  Wait for and write results.
    //

    while (list != NULL) {
      double *M = (double *)PAR_Wait(&list);
      cout << "\nCrossProduct:\n";
      for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
          cout << M[n*i+j] << " ";
        cout << '\n';
      }
    }
  }
```

This program

1. reads several floating point vectors from the standard input,
2. computes the cross products of all pairs of vectors in parallel by starting several tasks running the function `CrossProduct`, and
3. writes the resulting matrices to the standard output.

This program can be stored in a file called e.g.\ '`CrossProduct.cc`' and compiled with the command line

```
psturmc++ -O CrossProduct.cc -o CrossProduct
```

Please note that this will work only if the environment variables `Sturm` and `PSturm` contain the paths of the installation directories for the sequential and the parallel version of the STURM library, respectively. Furthermore, `psturmc++` is located in '`$PSturm/Bin`' so this directory should be on the shell's search path.

If compilation was successful the command line

```
./CrossProduct --proc 3 --heap 4000000
```

starts the program with 3 processors and a heap of 4 MB size and produce an output similar to the following:

```
--------------------------------------------------------------------
STURM Multi-Processor Kernel $Revision: 1.1 $ (c) 1994 RISC-Linz
3 processors, 3999744 bytes heap in 12 clusters, 16000 bytes stack.
--------------------------------------------------------------------
Number of vectors = 3
  Size of vectors = 2
Vector 0
  Element 0 = 1
  Element 1 = 2
Vector 1
  Element 0 = 10
  Element 1 = 20
Vector 2
  Element 0 = 4
  Element 1 = 8

CrossProduct:
4 8
8 16

CrossProduct:
40 80
80 160

CrossProduct:
10 20
20 40
--------------------------------------------------------------------
0 garbage collections, 0 bytes reclaimed.
Initialization time (TI):          421 ms.
Garbage collection time (TG):        0 ms.
Actual execution time (TE):       9862 ms.
Total program time (TI+TG+TE):   10283 ms.
--------------------------------------------------------------------
```

3 The Overall Program Structure

Every C++ program that utilizes the features of the STURM multi-processor kernel must contain the line

```
#include "Sturm/Parallel.h" // multi-processor interface
```

in addition to any header files required by certain packages of the STURM library (such as the basic 'Sturm/Kernel.h'). Furthermore the programmer must provide a function with the prototype

```
int STURM_Main(int argc, char *argv[])
```

This function serves as the program's entry point; when it returns, the program terminates and the return value is forwarded to the program environment.

The arguments `argc` and `argv` denote the program's command line arguments *after* having parsed and removed the arguments used for initializing the kernel (see Chapter 4 [Command Line Options], page 6). Consequently, the user must *not* use the syntax of these arguments for the command line arguments of her application.

The usual entry function `main()` must *not* be provided by the user since it is implemented in the kernel to perform several necessary initializations. Currently it is *not* possible to initialize the multi-processor kernel manually.

To simplify program development, a shell command is provided:

`psturmc++` [`c++` options]

takes the same command line arguments as `c++`. It automatically defines the search paths for the STURM header files (for compiling source files) and for the STURM libraries (for linking object files). This command automatically links the multi-processor kernel with object files.

The script requires that the environment variables `Sturm` and `PSturm` contain the paths of the installation directories for the sequential and the parallel version of STURM, respectively. As `psturmc++` is located in '`$PSturm/Bin`' this directory should be on your shell's search path.

4 Command Line Options

The multi-processor kernel can be initialized with the following command line options

`--cluster` *number*
: splits the heap into *number* clusters for distributed memory allocation (see [Garbage Collection], page 19). If *number* is smaller than the number of processors or the cluster size gets too small, the program is aborted with an error message. If not given, a default value is used.

`--heap` *size*
: allocates a heap with *size* bytes. The actual heap size may be actually somewhat smaller due to necessary alignments. If the heap is too small for program execution or too large for the machine, the program is aborted with an error message. If not given, a default value is used.

`--help`
: prints a summary of the command line options and terminates the program. If not given, the program is executed.

`--proc` *number*
: lets the program utilize *number* processors for execution. If not given, a single processor is used.

`--profile` *flag*
: generates profiling output. If *flag*=1, a profiling file '`sturm.dat`' is created for use with the shell commands `pacgraph` and `pacutil`. If *flag*=2, a profiling file '`sturm.dat2`' is created for use with the interactive visualization environment PACVIS (see Chapter 9 [Visualization], page 22). Other values cause the program to abort with an error message. If not given, no profiling output is generated.

`--silent`
: disables the generation of diagnostic output by the multi-processor kernel. If not given, the kernel generates output at the start of the program, at garbage collections, and at the end of the program.

`--stack` *size*
: sets the default stack size of every task to *size* bytes. This value may be overridden at runtime (see Section 8.3 [Stack Size], page 21). If not given, a default value is used.

`--time` *time*
: sets the time slice for preemptive task scheduling to *time* milli-seconds. *time*=0 disables time-slicing (non-preemptive scheduling). This value may be overridden at runtime (see Section 8.2 [Time Slice], page 20). If not given, a default value is used.

The program aborts with an error message on syntax errors when parsing these arguments. Note that additional STURM specific command line arguments (which are used both for the sequential and the parallel version) are described in section "Command Line Arguments" in *The* STURM *Library Manual*.

5 Functional Parallelism

In this section we describe the basic mechanisms for task creation, communication, synchronization and abortion. Most parallel programs can be comfortably expressed using this interface.

5.1 Type Definitions

We define several new types used in the following sections.

Type_Task Pointer type

This defines an object representing a task.

Type_TaskList Pointer type

This defines an object representing a list of tasks.

The following constant in general will be used as a starting object for iteratively constructing a list of tasks:

`Type_TaskList` **NULL** Constant

A constant representing the empty list.

For objects of type `Type_TaskList` the following operations are defined:

list = **PAR_Cons** (*head*, *tail*) Function

Append a task to a list of tasks.

head = **PAR_Head** (*list*) Function

Return the first element of a list of tasks.

tail = **PAR_Tail** (*list*) Function

Return a list of tasks without its first element.

`Type_TaskList` *list*

A list of tasks.

`Type_Task` *head*

A task – the first element of a list.

`Type_TaskList` *tail*

A list of tasks.

Description

1. `PAR_Cons` returns a *list* of tasks with the first element set to *head* and the rest list *tail*. The *tail* may be `NULL`.
2. `PAR_Head` returns the first element of the non-`NULL` list of tasks *list*.
3. `PAR_Tail` returns the sublist *tail* of the non-`NULL` list of tasks *list* starting from the second element of *list*.

5.2 Task Creation

In this section we describe how new tasks are started. For details on the task management, see *The Design of the PACLIB Kernel*.

The goal is to have a given function with given arguments be executed in parallel to the current task. This is handled by an overloaded function `PAR_Start`. This function is not defined in the library but must be created by calling the following macro somewhere outside of any function definition, before the corresponding `PAR_Start` is called:

PAR_CreateStart*N* (*ret*, *type1*, ..., *typeN*) Macro

This macro (which is defined for $N = 0, \ldots, 8$) takes the return type *ret* and the types of the arguments *type1*, ..., *typeN* of a function to be executed in parallel and generates an instance of the overloaded inline function `PAR_Start` which can be used for executing functions matching the given prototype in parallel (see Chapter 2 [Example], page 2 for a concrete example):

task = **PAR_Start** (*fun*, *arg1*, ..., *argN*) Function

Start a parallel task.

`Type_Task` *task*
: A a task computing *fun*(*arg1*, ..., *argN*).

ret (**fun*)(*type1*, ..., *typeN*)
: A pointer to the function to be executed in parallel.

type1 arg1, ..., *typeN argN*
: The arguments passed to *fun*.

Description

1. `PAR_Start` allocates the workspace for the new task (see Section 8.3 [Stack Size], page 21) and initializes it with the function pointer *fun* and the argument list *arg1*, ..., *argN*. The descriptor of the task is put into the global ready queue and `PAR_Start` returns. From this time *task* is ready for execution by any idle processor.
2. *task* is a reference to a heap-allocated object and may be stored in other heap-allocated objects, passed as an argument to functions and tasks, and transmitted by streams (see Chapter 6 [Stream Parallelism], page 13).
3. Tasks are preemptively executed in a first-come, first-serve order. An idle processor removes the first task from the head of the ready queue and starts its execution. If the task has not terminated after some fixed amount of time (see Section 8.2 [Time Slice], page 20), the processor is interrupted and the task is inserted at the tail the ready queue for later continuation.

4. The task terminates when its function returns; the result of *task* is the value returned by *fun* (see Section 5.3 [Task Synchronization], page 9).
5. When a task terminates, its workspace is immediately deallocated. The space occupied by the result of a task is reclaimed during garbage collection (see [Garbage Collection], page 19) if there is no more reference to the result (and to the *task*).

Restrictions

1. The return type *ret* must be `void` or a pointer to an object allocated on the heap. Otherwise, the behaviour of the program is undefined.
2. If there is no more memory for the workspace of a new task, `PAR_Start` aborts with an error message (see Section 8.3 [Stack Size], page 21, see Section 5.5 [Virtual Tasks], page 11).

Warnings

1. If any task modifies global variables accessed by another task, the behavior of the program is undefined.
2. If any task modifies the contents of *arg1*, ..., *argN*, the behavior of the program is undefined.
3. The *task* executes with a stack of finite size. If the stack overflows, the behavior of the program is undefined (see Section 8.3 [Stack Size], page 21).

5.3 Task Synchronization

In this section we describe how tasks can retrieve the results of other tasks. For details on the task management, see *The Design of the PACLIB Kernel.*

result = **PAR_Wait** (*task*) Function

Wait for the termination of a single task.

result = **PAR_Wait** (*taskptr*, *tasklist*) Function

Wait for the termination of any task in a list of tasks.

result = **PAR_Wait** (*tasklistptr*) Procedure

Wait for the termination of any task in a list of tasks and remove it from the list.

`void*` *result*

The return value of *task* (a pointer to a heap-allocated object).

`Type_Task` *task*

A task.

`Type_Task*` *taskptr*

A pointer to a task variable.

`Type_TaskList` *tasklist*

A non-`NULL` list of tasks.

`Type_TaskList*` *tasklistptr*

A pointer to a task list variable.

Description

1. The first version of `PAR_Wait` blocks the current task until *task* has terminated and then returns the *result* of this task.
2. The second version of `PAR_Wait` blocks the current task until any of the tasks in *tasklist* has terminated and then returns the *result* of this task. The task whose result was returned is stored in the variable referenced by *taskptr*.
3. The third version of `PAR_Wait` blocks the current task until any of the tasks in the task list referenced by *tasklistptr* has terminated and then returns the *result* of this task. The task whose result was returned is destructively removed from the list referenced by *tasklistptr*.
4. All versions of `PAR_Wait` are *non-destructive* i.e. several tasks may wait for the results of overlapping task sets without visible interference.

Warnings

1. The versions waiting on lists of tasks are *non-deterministic* constructs. They may return the result of *any* of the given tasks.
2. The versions waiting on lists of tasks are *not* necessarily *fair* in any sense; in particular they need not return the result of that task that terminated first.
3. If *taskptr* or *tasklistptr* is `NULL`, the behavior of the program is undefined.
4. The *result* of `PAR_Wait` must be explicitly cast to the return type of the function which the finished task executed.

5.4 Task Abortion

In this section we describe how tasks can prematurely abort the execution of other tasks. For details on the task management, see *The Design of the PACLIB Kernel*.

`void` **PAR_Stop** (*task*) Procedure

Abort the execution of a task.

`void` **PAR_Stop** (*tasklist*) Procedure

Abort the execution of all tasks in a list.

`Type_Task` *task*

A task.

`Type_TaskList` *tasklist*

A list of tasks (may be `NULL`).

Description

1. The first version of the `Stop` function aborts the execution of *task*. If *task* has already terminated, the statement has no effect.
2. The second version of the `Stop` function aborts the execution of all tasks in *tasklist*. Any task in this list which has already terminated is not affected.

Warnings

1. The family of `Stop` functions does *not* wait until the denoted tasks have been aborted. Task abortion is only triggered but may actually take place later.
2. No task may call any of the `Wait` functions on an aborted task. Otherwise the waiting task may be blocked forever which may lead to a deadlock. Consequently only the aborting task should usually have access to the reference of the aborted task.

5.5 Virtual Tasks

In this section we describe how tasks can be created in a lazy fashion. For more information on the management of virtual tasks, see *Virtual Tasks for the PACLIB Kernel*.

The scheme for starting virtual tasks is essentially identical to the scheme for starting ordinary tasks (see Section 5.2 [Task Creation], page 8): There is a macro which must be used for creating instances of the function `PAR_Virtual` which handles the parallel execution of functions.

PAR_CreateVirtual*N* (*ret*, *type1*, ..., *typeN*) Macro

This macro (which is defined for $N = 0, \ldots, 8$) takes the return type *ret* and the types of the arguments *type1*, ..., *typeN* of a function to be executed in parallel and generates an instance of the overloaded inline function `PAR_Virtual` which can be used for executing functions matching the given prototype in parallel:

task = **PAR_Virtual** (*fun*, *arg1*, ..., *argN*) Function

Start a parallel task.

`Type_Task` *task*
: A pointer to a task computing *fun*(*arg1*, ..., *argN*).

ret (**fun*)(*type1*, ..., *typeN*)
: A pointer to the function to be executed in parallel.

type1 arg1, ..., *typeN argN*
: The arguments passed to *fun*.

Description

1. `PAR_Virtual` has the same interface and semantics as `PAR_Start`. The main difference is the time when the new task is actually created: `PAR_Virtual` does *not* allocate the (large) workspace of the task and link it into the ready queue. Instead it creates a (small) virtual task descriptor, stores the information on *fun* and *arg1*, ..., *argN*, and links this descriptor into a *virtual task queue*. The descriptor stays there until it is transformed into a real task.

2. A virtual task is *realized* (i.e. transformed into a real task) only on the following occasions:
   a. *Idling* An idle processor finds the ready queue empty. It picks a descriptor from the virtual task queue, allocates a task descriptor containing the task workspace, and starts execution of the task.
   b. *Termination* A real task is going to terminate. It grips a descriptor from the virtual task queue, overwrites its own workspace with the new task information, and restarts its execution.
   c. *Inlining* A task calls `PAR_Wait(`*task*`)` on a *task* that is still virtual. In this case, the task grips the virtual task descriptor, computes the denoted function itself, delivers the result and continues execution.
   d. *Non-determinism* A task calls `PAR_Wait` waiting on a list of tasks some of which are still virtual. In this case, the task realizes all virtual tasks before it gets blocked.

   The last item is necessary to maintain the semantics of the program, since otherwise some of the virtual tasks might never become active (if there are non-terminating tasks).

Restrictions and Warnings

1. The same restrictions on `PAR_Virtual` are the same as those on `PAR_Start` (see Section 5.2 [Task Creation], page 8).

Notes

1. The main motivation for the application of virtual tasks is to save memory in those cases where the program creates many more tasks (say some 1000s) than processors are actually available. With virtual tasks the program may still run very efficiently, while with real tasks the program may become extremely slow (due to virtual memory paging) or even run out of memory.

6 Stream Parallelism

This section describes an alternative model for communication between tasks. While in the functional model a task has exactly one result that becomes available when the task terminates, in the stream model a task may transmit during its execution sequences of values to other tasks.

6.1 Streams

A *stream* is a buffered communication channel by which a task can transmit a sequence of values to other tasks. A stream can have one of three states:

1. *empty*: no element is *currently* available in the stream (but there may be one in the future).
2. *value:rest*: the stream consists of an element *value* and another stream *rest*.
3. *closed*: no element is and will ever become available in the stream.

Initially a stream is empty; an empty stream may be transformed into the states *value*:*rest* or closed.

The usual application of streams is the following: an empty stream *s0* is created and passed as an argument to a set of tasks.

- One task (the *producer*) appends an element *v0* to *s0* changing its state to *v0:s1* where *s1* is a new and empty stream.
- The other tasks (the *consumers*) receive *v0* and the rest stream *v1*. If *s0* is still empty, the consumer is blocked.

In this fashion, a sequence of streams *s0*, *s1*, ..., *s(N)* is generated transmitting the values *v0*, *v1*, ..., *v(N-1)*. The producer finally sets the state of *s(N)* to closed.

Please note that various tasks may have the same stream handle *s(I)* and simultaneously receive the associated value *v(I)* without visible interference. Streams are not destructively freed but their space is collected by the general garbage collection mechanism (see [Garbage Collection], page 19).

6.2 Stream Types

We define several new types used in the following sections.

Type_Stream Pointer type

This defines an object representing a stream.

Type_StreamList Pointer type

This defines an object representing a list of streams.

The following constant in general will be used as a starting object for iteratively constructing a list of streams:

`Type_StreamList` **NULL** Constant

A constant representing the empty list.

For objects of type `Type_StreamList` functions `PAR_Cons`, `PAR_Head`, and `PAR_Tail` are defined which behave similar to the corresponding functions defined for objects of type `Type_TaskList` (see Section 5.1 [Type Definitions], page 7).

Warnings

1. `Type_Stream` and `Type_StreamList` are just renamings of `Type_Task` and `Type_TaskList` since both have the same internal structure. While it is illegal to concatenate streams to task lists or tasks to stream lists, the compiler will not detect such violations. The consequence will be undefined program behavior.

6.3 Creating Streams

This section describes how an empty stream is created.

stream = **PAR_Open** (`void`) Function

Create a new stream.

`Type_Stream` *stream*

An empty stream.

Description

1. This function returns an empty stream that is not in use.
2. *stream* is a reference to a heap-allocated object and may be stored in other heap-allocated objects, passed as an argument to functions and tasks, and transmitted by streams.

6.4 Reading from Streams

This section describes how to get an object from a stream.

done = **PAR_Get** (*valueptr*, *restptr*, *stream*) Procedure

Get a value from a single stream.

done = **PAR_Get** (*valueptr*, *restptr*, *streamptr*, *slist*) Procedure

Get a value from any stream in a list of streams.

done = **PAR_Get** (*valueptr*, *slistptr*) Procedure

Get a value from any stream in a list of streams and update the list.

`Type_Bool` *done*
: A flag that signals whether an element could be read.

`void*` *valueptr*
: The address of a pointer variable.

`Type_Stream*` *restptr*
: A pointer to a stream variable.

`Type_Stream` *stream*
: A stream.

`Type_Stream*` *streamptr*
: A pointer to a stream variable.

`Type_StreamList` *slist*
: A list of streams.

`Type_StreamList*` *slistptr*
: A pointer to a stream list variable.

Description

1. The first versionof `PAR_Get` returns 1 if *stream* is in the state *value*:*rest*. The function then has written *value* into the location *valueptr* and *rest* into the location *restptr*. If *stream* is closed, `PAR_Get` returns 0. Otherwise the function blocks until the state of the empty *stream* is changed by another task and then performs above actions.
2. The second version of `PAR_Get` returns 1 if some *stream* in *slist* is in the state *value*:*rest*. The function then has written *value* into the location *valueptr*, *rest* into the location *restptr* and *stream* into the location *streamptr*. If all streams in *slist* are closed, `PAR_Get` returns 0. Otherwise the function blocks until the state of some empty stream in *slist* is changed by another task and then performs above actions.
3. The third version of `PAR_Get` returns 1 if some *stream* in the stream list referenced by *slistptr* is in the state *value*:*rest*. The function then has written *value* into the location *valueptr*. The function has also destructively replaced *stream* by *rest* in the list referenced by *slistptr*. If all streams in this list are closed, `parGetList` returns 0. Otherwise the function blocks until the state of some empty stream in the list referenced by *slistptr* is changed by another task and then performs above actions.

Warnings

1. The program behavior is undefined if any of *valueptr*, *restptr*, *streamptr* or *slistptr* is `NULL`.
2. The versions getting a values from a list of streams return 0 if *slist* is `NULL` or if the list referenced by *slistptr* is `NULL`.
3. The version getting a value from a single stream returns always the same value when called with the same *stream*. To get the next value, the stream stored in the variable referenced by *restptr* must be used.
4. The versions getting a values from a list of streams are *non-deterministic* constructs that need *not* be fair in any sense.

5. The version taking an argument *slistptr* destructively updates the list referenced by this argument. Do not call this function if the list may be in use anywhere else.
6. The value transmitted by a stream is *not* type-checked by the compiler. The user is responsible to provide as *valueptr* the address of a variable that matches the type of the transmitted value which must be a pointer to heap-allocated object (see Section 6.5 [Writing to Streams], page 16).

Notes

1. The version getting a value from a single stream is implemented such that *restptr* may be safely the address of the *stream* variable: *first* the *stream* variable is used and *then* the *rest* stream is written into the location *restptr*.

6.5 Writing to Streams

This section describes how to put an object into an empty stream.

newstream = **PAR_Put** (*oldstream*, *value*) Procedure

Put a value into a stream.

`void` **PAR_Put** (*oldstream*, *value*, *reststream*) Procedure

Put a value and a rest stream into a stream.

`Type_Stream` *newstream*

An empty stream.

`Type_Stream` *oldstream*

An empty stream.

`void*` *value*

A pointer to a heap-allocated object.

`Type_Stream` *reststream*

A stream.

Description

1. The first version of `PAR_Put` sets the state of *oldstream* to *value*:*newstream* and returns *newstream*. *newstream* is an empty stream which is not in use.
2. The second version of `PAR_Put` sets the state of *oldstream* to *value*:*reststream*.

Warnings

1. If *oldstream* is not empty, the behavior of the program is undefined. Consequently exactly one task may write to or close (see Section 6.6 [Terminating Streams], page 17) a stream.
2. If *value* is not a pointer to a heap-allocated object, the program behavior is undefined.

3. The functions are *not* type-checked by the compiler. When retrieving the value from a stream, the user is responsible for correct typing (see Section 6.4 [Reading from Streams], page 14).

6.6 Terminating Streams

This section describes how an empty stream is closed.

`void` **PAR_Close** (*stream*) Procedure
Terminate a stream.

`void` **PAR_Close** (*slist*) Procedure
Terminate a stream.

`Type_Stream` *stream*
An empty stream.

`Type_StreamList` *slist*
A list of empty streams.

Description

1. The first version of `PAR_CLOSE` puts *stream* into the state closed.
2. The second version of `PAR_Close` puts all streams in *slist* into the state closed.

Warnings

1. If *stream* or some stream in *slist* is not empty, the program behavior is undefined. Consequently only one task may write to (see Section 6.5 [Writing to Streams], page 16) or close a stream.

7 Memory Management

Memory allocation on the heap is done using the functions documented in section "Memory Management" in *The* STURM *Library Manual*.

The only difference between the memory management of the sequential kernel and that of the multi-processor kernel is the garbage collection mechanism.

7.1 Garbage Collection

This section describes how heap memory is organized and reclaimed by garbage collection. The user basically needs not know about this.

Logical Organization

Logically, the heap is organized as a set of *clusters*. Initially every processor has one cluster attached; all remaining clusters reside in a global *free pool* that is protected by a lock variable from concurrent access. The number of clusters can be controlled by a command line option (see Chapter 4 [Command Line Options], page 6).

If a task issues a memory allocation request, the executing processor tries to fulfill this request from the attached cluster. All free memory blocks in this cluster are doubly linked in a cycle. The processor traverses this cycle to find a free block of appropriate size. If such a block exists, the processor cuts off a part from the end of this block and returns this part to the caller.

If the attached cluster is not able to fulfill the request, the processor puts the cluster into the *garbage pool* and picks a new cluster from the free pool. If the free pool is empty, the processor triggers garbage collection interrupting all other processors.

The garbage collector is based on a *mark and sweep* scheme:

1. *mark phase*: the processors concurrently scan the stacks of all tasks to find potential references to heap-allocated objects. These objects are marked; all objects potentially referenced by marked objects are recursively marked as well. All memory regions registered to contain pointers are in the same fashion scanned for potential references.
2. *sweep phase*: the processors concurrently scan every cluster in the garbage pool. The mark flags in marked blocks are erased, the unmarked blocks are merged and cyclically linked into the free list of the cluster. One cluster is attached to every processor, all remaining clusters are linked into the free pool.

To simplify the mark phase, every cluster is organized as a sequence of *cells*. Furthermore every cluster has a *bitmap* in which one bit is reserved for every cell. Memory blocks are always allocated at cell boundaries, the bit that corresponds to the initial cell of the block is set. During the mark phase, these bits are used to detect the beginnings of allocated blocks and to set the corresponding mark flags.

Explicit Deallocation

If a memory block is explicitly deallocated the processor checks whether the deallocated block is located in the cluster attached to the processor. If not, the deallocation request is ignored.

Otherwise the processor tries to merge the deallocated block with the first free block of the cluster. This is possible, if the freed block was created in the last allocation request executed on this processor from this free block. If this is not the case, the block is linked on its own into the free cycle of the cluster.

In any case, complete reclamation and merging of no more used blocks takes place at the next garbage collection phase.

Physical Layout

The physical layout of the heap is somewhat different from the logical organization. There are three contiguous memory areas:

1. The *header area* contains a (fixed size) record for every heap cluster.
2. The *bitmap area* contains the sequence of bits for all cells in all heap clusters.
3. The *cell area* contains the sequence of cells for all heap clusters.

Memory blocks are allocated in the cell area. By this layout a simple comparison with the bounds of the cell area is able to distinguish potential pointers from sure non-pointers.

Currently a cell consists of 3 words a 4 bytes. The first word of every memory block contains management information; the address of the second word is actually returned to the user. In total, on a request of n bytes

$$\mathit{floor}((n+15)/12) \text{ cells}$$

are allocated in the cell area where each cell occupies 96 bits + 1 bit in the bitmap area.

8 Miscellaneous Features

In this section we describe several other features useful for parallel STURM programs.

8.1 Runtime Marks

`void` **PAR_Mark** (*mark*) Procedure

Set a mark for visualization.

`int` *mark* An integer mark.

Description

1. The function sets a runtime *mark* for later visualization of the program behavior (see Chapter 9 [Visualization], page 22).

Warnings

1. Marks must be set over the time in increasing (but not necessarily consecutive) order; otherwise the visualization tools may abort with an error message (see Chapter 9 [Visualization], page 22).

8.2 Time Slice

`void` **PAR_TimeSlice** (*time*) Procedure

Set the default time slice.

`int` *time* The default time slice in milli-seconds.

Description

1. The function sets the time slice for preemptive task scheduling to *time* milli-seconds. *time*=0 disables time-slicing (non-preemptive scheduling). The initial time slice may be set by a command line option (see Chapter 4 [Command Line Options], page 6).

Warning

1. Disabling time-slicing may change the semantics of programs and cause program deadlocks, since non-terminating tasks then occupy processors forever.

8.3 Stack Size

`void` **PAR_StackSize** (*size*) Procedure

Set the default stack size.

`int` *size* The default byte size for the workspace of new tasks.

Description

1. The function causes all tasks created by the current task to get a workspace (stack) of *size* bytes. This value will be also inherited by all tasks created by the current task. The initial default workspace size may be set by a command line option (see Chapter 4 [Command Line Options], page 6).

Warnings

1. If too many tasks are started (respectively their stacks are too large), the system may run out of memory when creating new tasks. The program then aborts with an error message (see Section 5.2 [Task Creation], page 8, see Section 5.5 [Virtual Tasks], page 11).
2. The default stack size should be large enough for most applications. However, deeply nested recursions may cause stack overflows and consequently undefined program behavior.

8.4 Input/Output

The C++ interface for input/output is based on the usual stream model using the standard C++ operators.

`>>` Operator on istream

Read an object *obj* from an input stream *cin*: *cin* `>>` *obj*

`<<` Operator on ostream

Write an object *obj* to an output stream *cout*: *cout* `<<` *obj*

These are overloaded for most data types of the STURM library.

Note

1. File input/output is still to be defined.

9 Visualization

The behavior of a program linked with the multi-processor STURM kernel may be visualized *after* the program run. If the program has been started with the corresponding command line options (see Chapter 4 [Command Line Options], page 6) the kernel writes profiling information into a file '`sturm.dat`' (respectively '`sturm.dat2`').

The PACLIB shell scripts

`pacgraph` *xdim ydim [mark1 [mark2]]* `< 'sturm.dat' >` *graphfile*

and

`pacutil` *xdim ydim [mark1 [mark2]]* `< 'sturm.dat' >` *utilfile*

can be used to visualize the information of the first type by creating LaTeX pictures of size *xdim**ydim* points. If *mark1* is given, the program only visualizes the part of the program run starting at the runtime mark *mark1*. If also *mark2*>*mark1* is given, the program only visualizes the part of the program run starting at *mark1* and ending at *mark2* (see Section 8.1 [Runtime Marks], page 20).

1. *graphfile* consists of a sequence of (possibly interrupted) horizontal lines. Each line represents a task; the line segments represent the time slices when the task was executed.
2. *utilfile* represents a graph that shows how many tasks were executed (i.e. how many processors were utilized) at any moment.

The profiling information of the second type can be investigated in an interactive way by the X11-based visualization environment PACVIS:

```
pv 'sturm.dat2' &
```

pops up one or more windows in which several runtime aspects of the program may be investigated in various levels of detail.

For more details on PACVIS, see *Performance Analysis of Parallel Programs: The PACVIS Visualization Tool.*

References

Bruno Buchberger, George Collins, Hoon Hong et. al.
SACLIB 1.1 User's Guide. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992.

Hoon Hong, Wolfgang Schreiner, Andreas Neubacher, et al.
PACLIB User Manual. Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria, May 1992.

Hoon Hong, Wolfgang Schreiner, Tibor Fadgyas.
Performance Analysis of Parallel Programs: The PACVIS Visualization Tool. Technical Report 94-07, RISC-Linz, Johannes Kepler University, Linz, Austria, January 1994.

Hoon Hong, Andreas Neubacher, Volker Stahl.
The STURM *Library Manual - A C++ Library for Symbolic Computation* Technical Report, RISC-Linz, Johannes Kepler University, Linz, Austria, 1994.

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

Wolfgang Schreiner.
Virtual Tasks for the PACLIB Kernel. Technical Report, RISC-Linz, Johannes Kepler University, Linz, Austria, January 1994.

Peter A. Buhr, Richard A. Stroobosscher.
The uSystem: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running Unix. Software - Practice and Experience, 20(9), pages 929-964, September 1990.

List of Functions, Types and Commands

Index

S

T

U

V

W

X

Table of Contents