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

Copyright © 1994 RISC-Linz.

Permission is granted to use this material and the accompanying programs within the terms of the GNU GPL.

1 Introduction

STURM is a package of C++ functions for algebraic computation. We have developed it based on our experience with SACLIB (see the *SACLIB 1.1 User's Guide*), a C variant of the well-known computer algebra library SAC-2. The goal of the STURM package is to provide the basic objects and methods for algebraic algorithms in a modern software-engineering framework. The library shall be embeddable into application programs that demand algebraic facilities, in particular into constraint solvers.

The STURM 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. Note that only the C++ version may actually 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 C++ Interface to 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

Sorry, no example program for the C version of the kernel available, yet.

3 The Overall Program Structure

The features provided by the STURM multi_processor kernel (but *not* the other features of the STURM library) may also be utilized by C programs which contain the line

```
#include "Sturm/C/parallel.h" /* multi-processor interface */
```

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. Program execution may be also terminated at any other point of the program (see Section 8.6 [Termination], page 24, see Section 8.7 [Abortion], page 25).

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 4). 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.

Program development is simplified by the shell command

```
psturmcc [ cc options ]
```

that compiles C source files and links them with the multi-processor kernel.

The script requires that the environment variable `PSturm` contains the path of the installation directory for the parallel version of the STURM library. As `psturmcc` 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 21). 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 26). 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 23). 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 22). If not given, a default value is used.

The program aborts with an error message on syntax errors when parsing these arguments. All other command line arguments are forwarded to `STURM_Main` (see Chapter 3 [Program Structure], page 3).

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.

Types

`parTask`
: A (pointer to a) task.

`parTaskList`
: `NULL` or a pointer to a structure with components `head` (the first task) and `tail` (`NULL` or a pointer to another such structure).

Constructors

list = `parCons(`*head*`,` *tail*`)`

`parTaskList` *list*
: The result of concatenating *head* with *tail*.

`parTask` *head*
: A task.

`parTaskList` *tail*
: A list of tasks (possibly `NULL`).

Description

1. Returns a *list* with the first element *head* and the rest list *tail*.

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*.

task = `parStart`*N*`(`*fun*`,` *arg1*`, ...,` *argN*`)`

task = `parStart(`*fun*`,` *size*`,` *arg1*`, ...)`

`parTask` *task*

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

N

A small non-negative integer constant denoting the number of arguments.

`void* (*`*fun*`)()`

A function whose result type is a pointer to a heap-allocated object.

`int` *size*

An upper bound for the byte size of the argument list `(`*arg1*`, ..., `*argN*`)` as aligned on the stack i.e. *size* `>` `(asizeof(`*arg1*`) + ... + asizeof(`*argN*`))` where `asizeof(`*arg*`)` is the byte size of *arg* as aligned on the stack.

`[any type]` *arg1*`, ...,` *argN*

The list of arguments passed to *fun*.

Description

1. `parStart`*N* is a macro wrapped around `parStart`.
2. `parStart` allocates the workspace for the new task (see Section 8.3 [Stack Size], page 23) 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 `parStart` returns. From this time *task* is ready for execution by any idle processor.
3. *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 10).
4. 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 22), the processor is interrupted and the task is inserted at the tail the ready queue for later continuation.
5. The task terminates when its function returns; the result of *task* is the value returned by *fun*.
6. 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 21) if there is no more reference to the result (and to the *task*).

Restrictions

1. `parStart`*N* is only defined for 0 `<=` *N* `<=` 8.
2. If *size* exceeds some internal bound, `parStart` aborts with an error message.
3. If there is no more memory for the workspace of a new task, `parStart` aborts with an error message (see Section 8.3 [Stack Size], page 23, see Section 5.5 [Virtual Tasks], page 8).

Warnings

1. If the result type of *fun* is not a pointer to some object allocated on the heap, the behavior of the program is undefined.
2. The function application is *not* type-checked by the compiler. If the types of the arguments do not match the formal parameter list of *fun*, the behavior of the program is undefined.

3. If any task modifies global variables accessed by another task, the behavior of the program is undefined.
4. If any task modifies the contents of *arg1*, ..., *argN*, the behavior of the program is undefined.
5. 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 23).

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 = `parWaitOne`(*task*)

result = `parWaitList`(*taskptr*, *tasklist*)

result = `parWaitListRm`(*tasklistptr*)

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

`parTask` *task*
: A task.

`parTask*` *taskptr*
: The address of a `parTask` variable.

`parTaskList` *tasklist*
: A non-`NULL` list of tasks.

`parTaskList*` *tasklistptr*
: The address of a `parTaskList` variable.

Description

1. `parWaitOne` blocks the current task until *task* has terminated and then returns the *result* of this task.
2. `parWaitList` blocks the current task until any of the tasks in *tasklist* has terminated and then returns the *result* of this *task*. The identifier *task* is stored in the variable referenced by *taskptr*.
3. `parWaitListRm` 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 identifier *task* is destructively removed from the list referenced by *tasklistptr*.
4. The family of `parWait` functions is *non-destructive* i.e. several tasks may wait for the results of overlapping task sets without visible interference.

Warnings

1. `parWaitList` and `parWaitListRm` are *non-deterministic* constructs. They may return the result of *any* of the given tasks.

2. `parWaitList` and `parWaitListRm` 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 *parWait* must be explicitly casted to the return type of *fun*.

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.*

`parStopOne(`*task*`)`

`parStopList(`*tasklist*`)`

`parTask` *task*
: A task.

`parTaskList` *tasklist*
: A list of tasks (may be `NULL`).

Description

1. `parStopOne` aborts the execution of *task*. If *task* has already terminated, the statement has no effect.
2. `parStopList` aborts the execution of all tasks in *tasklist*. Every task in this list that has already terminated is not affected.

Warnings

1. The family of `parStop` 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 `parWait` 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.*

task `= parVirtual`*N*`(`*fun*`,` *arg1*`, ...,` *argN*`)`

task `= parVirtual(`*fun*`,` *size*`,` *arg1*`, ...)`

`parTask` *task*

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

N

A small non-negative integer constant denoting the number of arguments.

`void* (*`*fun*`)()`

A function whose result is a pointer to an object allocated on the heap.

`int` *size*

An upper bound for the byte size of the argument list `(`*arg1*`, ..., `*argN*`)` as allocated on the stack i.e. *size* `>` `(asizeof(`*arg1*`) + ... + asizeof(`*argN*`))` where `asizeof(`*arg*`)` is the byte size of *arg* as aligned on the stack.

`[any type]` *arg1*`, ...,` *argN*

The list of arguments passed to *fun*.

Description

1. `parVirtual`*N* is a macro wrapped around `parVirtual`.
2. `parVirtual` has the same interface and semantics as `parStart`. The main difference is the time when the new task is actually created: `parVirtual` 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.
3. 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 `parWaitOne` 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 `parWaitList` for several 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).

Warnings

1. The same restrictions apply to *fun* as in the `pacStart` family of functions (see Section 5.2 [Task Creation], page 5).

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 21).

6.2 Stream Types

`parStream`

A (pointer to a) stream of objects.

`parStreamList`

A pointer to a structure with components `head` (the first stream) and `tail` (a pointer to another such structure or `NULL`).

Notes

1. The operation `parCons` (see [Type Definitions], page 5) may be applied to lists of streams.

Warnings

1. `parStream` and `parStreamList` are just renamings of `parTask` and `parTaskList` since both have the same internal structure. While it is illegal to concatenate streams to task lists or tasks to stream lists, the type checker will not detect violations. The consequence is undefined program behavior.

6.3 Creating Streams

This section describes how an empty stream is created.

stream `= parOpen()`

`parStream` *stream*
: An empty stream.

Description

1. The 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 `= parGetOne(`*valueptr*`,` *restptr*`,` *stream*`)`

done `= parGetList(`*valueptr*`,` *restptr*`,` *streamptr*`,` *slist*`)`

done `= parGetListRm(`*valueptr*`,` *slistptr*`)`

`int` *done* A boolean flag that signals whether an element could be read.

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

`parStream*` *restptr*
: The address of a `parStream` variable.

`parStream` *stream*
: A stream.

`parStream*` *streamptr*
: The address of a `parStream` variable.

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

`parStreamList*` *slistptr*
: The address of a `parStreamList` variable.

Description

1. `parGetOne` 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, `parGetOne` returns 0. Otherwise the function blocks until the state of the empty *stream* is changed by another task and then performs above actions.
2. `parGetList` 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, `parGetList` 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. `parGetListRm` 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. `parGetList` returns 0 if *slist* is `NULL`. Likewise `parGetListRm` returns 0 if the list referenced by *slistptr* is `NULL`.
3. `parGetOne` 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. `parGetList` and `parGetListRm` are *non-deterministic* constructs that need *not* be fair in any sense.
5. `parGetListRm` destructively updates the list referenced by *slistptr*. 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 13).

Notes

1. `parGetOne` 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 `= parPut(`*oldstream*`,` *value*`)`

`parPutRest(`*oldstream*`, `*value*`, `*reststream*`)`

`parStream` *newstream*
: An empty stream.

`parStream` *oldstream*
: An empty stream.

`void*` *value*
: A pointer to a heap-allocated object.

`parStream` *reststream*
: A stream.

Description

1. `parPut` sets the state of *oldstream* to *value*:*newstream* and returns *newstream*. *newstream* is an empty stream that is not in use.
2. `parPutRest` 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 13) 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 11).

6.6 Terminating Streams

This section describes how an empty stream is closed.

`parCloseOne(`*stream*`)`

`parCloseList(`*slist*`)`

`parStream` *stream*
: An empty stream.

`parStreamList` *slist*
: A list of empty streams.

Description

1. `parCloseOne` puts *stream* into the state closed.
2. `parCloseList` 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 13) or close a stream.

7 Memory Management

This section describes the interface to the memory management that simulates an infinite heap from wich memory blocks of arbitrary size can be allocated.

7.1 Header Files

Every program that needs access to the memory management facilities of the kernel must include the following header file.

`#include "sturm/C/memory.h"`
: Defines the C interface of the memory management facilities.

7.2 Allocation

This section describes how memory blocks may be allocated from the heap.

pointer `= memAlloc(`*size*`)`

`void*` *pointer*
: A pointer to a heap-allocated block of at least *size* bytes or `NULL` if the allocation failed.

`size_t` *size*
: The number of bytes to be allocated.

Description

1. The function returns a *pointer* to a heap-allocated memory block of at least *size* bytes that is not in use.
2. If *size* <= 0, a pointer to a memory block of zero size is returned.
3. *pointer* is aligned such that an object of *any* data type can be stored at the beginning of the block.
4. *pointer* must be explicitly casted to the required pointer type.

Warnings

1. The user must check for the success of the allocation i.e. whether *pointer* `== NULL`.
2. Any attempt to read or write beyond the bounds of the allocated block results in undefined program behavior.
3. `memSize(`*pointer*`)` does *not* necessarily return *size* (see Section 7.4 [Size Information], page 16).
4. Pointers to heap-allocated objects must *not* be stored in global variables without provision (see Section 7.5 [Global Variables], page 17).

7.3 Integer Blocks

This section describes an alternative way of allocating memory for objects that do *not* contain pointers to heap-allocated objects. This may make garbage collection more efficient (see [Garbage Collection], page 21).

pointer = `memAllocInt`(*size*)

`void*` *pointer*
: A pointer to a heap-allocated block of at least *size* bytes or `NULL` if the allocation failed.

`size_t` *size*
: The number of bytes to be allocated.

Description

1. The function behaves like `memAlloc` (see Section 7.2 [Allocation], page 15) but asserts that the memory block will *not* contain any pointers.
2. Garbage collection may become more efficient if `memAllocInt` is used instead of `memAlloc` (see [Garbage Collection], page 21).

Warnings

1. The same warnings apply as for `memAlloc` (see Section 7.2 [Allocation], page 15).
2. Any attempt to store pointers to heap-allocated objects in memory blocks allocated by this function result in undefined program behavior during and after garbage collection (see [Garbage Collection], page 21).

7.4 Size Information

This section describes how to request the size of a heap-allocated memory block.

size = `memSize`(*pointer*)

`size_t` *size*
: The number of bytes allocated on the heap for the memory block starting at *pointer*.

`void*` *pointer*
: A pointer to a heap-allocated memory block.

Description

1. This function returns the number of bytes allocated on the heap for the memory block starting at *pointer*.

Warnings

1. If *pointer* is `NULL` or *not* the result of a previous `memAlloc`, `memAllocInt` or `memRealloc` call or if the corresponding block was after allocation deallocated by `memRealloc` or `memFree`, the program behavior is undefined.
2. *size* not necessarily equals the size used in the allocation call. Only the inequality
   size `<=` `memSize(memAlloc(`*size*`))`
   holds (likewise for the other allocation functions).

7.5 Global Variables

Basically pointers to heap-allocated objects must *not* be stored in global variables. The following function allows to relax this restriction.

`memRegister(`*pointer*`,` *size*`)`

`void*` *pointer*
: A pointer to a memory region.

`size_t` *size*
: The size of the memory region in bytes.

Description

1. This function declares that the memory region starting at *pointer* and ranging over *size* bytes may contain pointers to heap-allocated objects.

Warnings

1. Any pointer to a heap-allocated object that is not stored in the local variable of a function (i.e. on the stack) or in another heap-allocated object results in undefined program behavior unless the location of the pointer is declared by this function.
2. Memory regions may be only registered at the very beginning of the program *before* any other function of the STURM library is called.
3. The contents of registered regions must be defined *before* any task is started that may refer to this region. Otherwise race conditions may occur and the behavior of the program is undefined.

Notes

1. This function may be used for registering statically allocated variables (storage class `extern` or `static` in C).
2. The necessity for this feature is most probably the consequence of a bad program design. You should not use it.

7.6 Deallocation

This section describes how to explicitly free the space of a heap-allocated object without relying on garbage collection. This feature is potentially dangerous and should be used with utmost care.

`memFree(`*pointer*`)`

`void*` *pointer*
: A pointer to a heap-allocated block of memory.

Description

1. This function asserts that the heap-allocated block starting at *pointer* is not used any more.
2. It may be the case that the memory block is returned to the free heap space immediately without waiting for the next garbage collection.

Warnings

1. If *pointer* is `NULL` or *not* the result of a previous `memAlloc`, `memAllocInt` or `memRealloc` call or if the corresponding block was already deallocated by `memFree` or `memRealloc`, the program behavior is undefined.
2. Any further access to the block starting at *pointer* results in undefined program behavior. Do *not* use this function if *pointer* might be in use somewhere else.

Implementation

1. A deallocated memory block is returned to the free block list only if the block has been recently allocated on the same processor where the current task is running. Furthermore, it is only attempted to merge the block with the current free block.
2. Complete reclamation of memory and merging of free blocks takes place only in garbage collection (see [Garbage Collection], page 21).

Notes

1. This function is dangerous and does not guarantee any implementation-independent advantage. Use it only if the efficiency gains may be really significant.
2. The assignment *pointer* `= NULL` has almost the same effect and is always safe. If the variable *pointer* was the only reference to the corresponding memory block, the block will be reclaimed at the next garbage collection.

7.7 Resizing

This section describes how to allocate a memory block that has partially the contents of another block that will be no more used. This feature is potentially dangerous and should be used with utmost care.

newptr `= memRealloc(`*oldptr*`,` *newsize*`)`

`void*` *newptr*
: A pointer to a heap-allocated block of at least *size* bytes or `NULL` if the allocation failed.

`void*` *oldptr*
: A pointer to a heap-allocated block.

`size_t` *newsize*
: The number of bytes to be allocated.

Description

1. If *newptr* is `NULL`, the block starting at *oldptr* is unchanged and may be further used.
2. If *newptr* is non-`NULL` the first *size* bytes of the block starting at *newptr* will have the same contents as the first *size* bytes of the block starting at *oldptr* where

   size = `min(`*newsize*`, memSize(`*oldptr*`))`

   Furthermore this function asserts that the block starting at *oldptr* is not used any more.
3. If *newsize* `<=` 0, a pointer to a memory block of zero size is returned.
4. It may be the case that the space of the old block is immediately reused for the new block.

Warnings

1. If *oldptr* is `NULL` or *not* the result of a previous `memAlloc`, `memAllocInt` or `memRealloc` call or if the corresponding block was deallocated by `memFree` or `memRealloc`, the program behavior is undefined.
2. The user must always check for the success of the allocation i.e. whether *newptr* = `NULL`.
3. If *newptr* is non-`NULL`, any further attempt to access the block starting at *oldptr* results in undefined program behavior. Do *not* use this function if *pointer* might be in use somewhere else.
4. *newptr* needs not equal *oldptr*, even if *newsize* `<=` `memSize(`*oldptr*`)`.
5. Any attempt to write beyond the bounds of the allocated block results in undefined program behavior.
6. If the block starting at *oldptr* was allocated by `memAllocInt`, any attempt to store pointers in the block starting at *newptr* results in undefined program behavior during and after the next garbage collection.
7. `memSize(`*newptr*`)` does *not* necessarily return *newsize* (see Section 7.4 [Size Information], page 16).

Implementation

1. When a memory block is enlarged, a new block is allocated, the contents are transferred, and the old block is deallocated using `memFree` (see Section 7.6 [Deallocation], page 17).
2. When a memory block is reduced in size, the unused rest is deallocated using `memFree` (see Section 7.6 [Deallocation], page 17) which returns it into the free list but does not merge it with free neighbor blocks until garbage collection (see [Garbage Collection], page 21).

Notes

1. This function is dangerous and does not guarantee any implementation-independent advantage. Only use it if the efficiency gains may be really significant.
2. Almost the same effect can be achieved in a safe way by allocating a new block of memory, transferring the contents of the old block and performing the assignment *oldptr* `=` `NULL`. The old block will be reclaimed at the next garbage collection.

7.8 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 4).

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 (see Section 7.5 [Global Variables], page 17) 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 (see Section 7.6 [Deallocation], page 17, see Section 7.7 [Resizing], page 18), 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

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

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

`parMark(`*mark*`)`

`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 26).

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 26).

8.2 Time Slice

`parTimeSlice(`*time*`)`

`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 4).

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

`parStackSize(`*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 4).

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 5, see Section 5.5 [Virtual Tasks], page 8).
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

`parPutchar(`*char*`)`

char `= parGetchar()`

`parUngetchar(`*char*`)`

`parPrintf(`*format*`, ...)`

`parScanf(`*format*`, ...)`

`int` *char*
: A character code (or `EOF`=-1 in `parGetChar`).

`const CHAR*` *format*
: A literal format string.

Description

1. `parPutChar` appends one character to the standard output stream.
2. `parGetChar` reads one character (and removes it) from the standard input stream.

3. `parUngetChar` inserts one character at the beginning of the input stream. This function may be called at most once before the next `parGetChar` is called.
4. `parPrintf` behaves like the C function `printf` writing formatted data to the standard output stream.
5. `parPrintf` behaves like the C function `scanf` reading formatted data from the standard input stream.

Warnings

1. All input/output *must* be implemented by these functions, otherwise the behavior of the program is undefined. The standard C functions must *not* be used.

Note

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

8.5 Clock

time = `parClock()`

time = `parClockLong()`

`int` *time*
: The current time in milli-seconds.

Description

1. Both functions return the current time in milli-seconds.
2. `parClock` has a resolution of 1 milli-second. Since the function is actually based on a clock with a resolution of 1 micro-second, the function has a turn-around time of approximately half an hour, which is too small for many applications.
3. `parClockLong` has a resolution of 10 milli-seconds only. However, its turn-around time is practically infinite (i.e. long enough for most applications).

Warnings

1. When using `parClock`, the user should check for clock turn-around.

8.6 Termination

`parExit(`*code*`)`

`int` *code*

The program return code.

Description

1. This function allows any task to terminate the program delivering the program return value *code*.
2. The function never returns.

8.7 Abortion

`parAbort(`*format*`, ...)`

`const char*` *format*
A literal format string.

Description

1. This function allows any task to abort the program with the denoted error message.
2. The function never returns.

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 4) 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 22).

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

A

argc ... 3
argv ... 3

C

cc ... 3

M

main ... 3
memAlloc ... 15
memAllocInt ... 16
memFree ... 17
memRealloc ... 18
memRegister ... 17
memSize ... 16

P

pacgraph ... 4, 26
pacutil ... 4, 26
PAR_Cons ... 5
parAbort ... 25
parClock ... 24
parClockLong ... 24
parCloseList ... 13
parCloseOne ... 13
parCons ... 5, 10
parExit ... 24
parGetchar ... 23
parGetList ... 11
parGetOne ... 11
parMark ... 22
parOpen ... 11
parPrintf ... 23
parPut ... 13
parPutchar ... 23
parPutRest ... 13
parScanf ... 23
parStackSize ... 23
parStart ... 5
parStartN ... 5
parStopList ... 8
parStopOne ... 8
parStream ... 10
parStreamList ... 10
parTask ... 5
parTaskList ... 5
parTimeSlice ... 22
parUngetchar ... 23
parVirtual ... 8
parVirtualN ... 8
parWaitList ... 7
parWaitListRm ... 7
parWaitOne ... 7
psturmcc ... 3
pv ... 4, 26

S

sturm.dat ... 4, 26
sturm.dat2 ... 4, 26
STURM_Main ... 3

Index

A

Aborting tasks ... 8
Aborting the program ... 25
Allocating memory ... 15
Allocating memory for integer blocks ... 16
Arguments to the program ... 3, 4

B

Benchmarking ... 24

C

C interface ... 1
C++ interface ... 1
Character streams ... 23
Clock ... 24
Closing streams ... 13
Cluster number ... 4
Command line options ... 3, 4
Communication between tasks ... 7
Communication by streams ... 11
Compiling C programs ... 3
Consumer tasks ... 10
Creating a stream ... 11
Creating a task ... 5
Creating a virtual task ... 8

D

Deallocating memory ... 17
Delayed task creation ... 8
Diagnostic output ... 4

E

Entry point of program ... 3
Example program ... 2
Exiting the program ... 24

F

Freeing memory ... 17
Functional parallelism ... 5

G

Garbage collection ... 20
Getting values from streams ... 11
Global variables ... 17

H

Hardware ... 1
Header files ... 3, 15
Heap allocation ... 15
Heap clusters ... 4, 20
Heap organization ... 20
Heap size ... 4
Help ... 4

I

Include files ... 3, 15
Input/output by the program ... 23
Integer blocks ... 16
Interactive visualization ... 26
Interface to memory management ... 15
Interface to task management ... 3

K

Kernel ... 1
Killing tasks ... 8

L

Lazy tasks ... 8
Linking C programs ... 3
List operations ... 5, 10

M

Main program ... 3
Mark and sweep ... 20
Marking program runs ... 22
Memory blocks, allocation ... 15
Memory blocks, allocation for integers ... 16
Memory blocks, deallocation ... 17
Memory blocks, resizing ... 18
Memory blocks, size of ... 16
Memory for tasks ... 9
Memory management ... 20
Memory management interface ... 15
Messages by the kernel ... 4
Monitoring the program ... 4, 26
Multi-processors ... 1

O

Opening a stream ... 11
Operating system ... 1
Options to the program ... 3, 4
Output by the kernel ... 4
Output by the program ... 23

P

PACLIB ... 1
PACVIS ... 4, 26
Performance analysis ... 4, 26
Processor number ... 4
Producer tasks ... 10
Profiling the program ... 4, 26

Program abortion ... 25
Program arguments ... 3, 4
Program development ... 3
Program entry point ... 3
Program example ... 2
Program options ... 4
Program structure ... 3
Program termination ... 24
Program visualization ... 26

R

Reading from streams ... 11
Ready queue ... 6, 9
Registering global variables ... 17
Resizing memory blocks ... 18
Runtime marks ... 22
Runtime system ... 1

S

SAC-2 ... 1
SACLIB ... 1
Sample program ... 2
Scheduling tasks ... 4, 6, 9, 22
Search paths ... 3
Sequent Symmetry ... 1
Shared memory ... 1
Shell scripts ... 3
Silent program execution ... 4
Size of memory blocks ... 16
Stack size ... 4, 23
Starting a task ... 5
Stopping tasks ... 8
Stopping the program ... 24, 25
Streams, creation ... 11
Streams, overview ... 10
Streams, reading from ... 11
Streams, terminating ... 13
Streams, type definitions ... 10
Streams, writing to ... 13
Sturm ... 1
Synchronizing tasks ... 7

T

Task abortion ... 8
Task blocking ... 7
Task communication ... 7
Task communication by streams ... 11
Task creation, delayed ... 8
Task creation, immediate ... 5
Task stacksize ... 9
Task synchronization ... 7
Terminal input/output ... 23
Terminating streams ... 13
Terminating tasks ... 8
Terminating the program ... 24, 25
Time slice ... 4, 22
Timing ... 24
Trace files ... 4, 26
Type definitions ... 5, 10

U

uSystem ... 1

V

Verbose program execution ... 4
Virtual tasks ... 8
Visualizing the program ... 4, 26

W

Waiting for tasks ... 7
Workspace size ... 4, 23
Writing to streams ... 13

X

X11 ... 26

Table of Contents

1 Introduction 1

2 A Sample Program 2

3 The Overall Program Structure 3

4 Command Line Options 4

5 Functional Parallelism 5

5.1 Type Definitions 5
Types 5
Constructors 5
5.2 Task Creation 5
5.3 Task Synchronization 7
5.4 Task Abortion 8
5.5 Virtual Tasks 8

6 Stream Parallelism 10

6.1 Streams 10
6.2 Stream Types 10
6.3 Creating Streams 11
6.4 Reading from Streams 11
6.5 Writing to Streams 12
6.6 Terminating Streams 13

7 Memory Management 15

7.1 Header Files 15
7.2 Allocation 15
7.3 Integer Blocks 16
7.4 Size Information 16
7.5 Global Variables 17
7.6 Deallocation 17
7.7 Resizing 18
7.8 Garbage Collection 20
Logical Organization 20
Explicit Deallocation 20
Physical Layout 21

8 Miscellaneous Features 22

8.1 Runtime Marks 22
8.2 Time Slice 22
8.3 Stack Size 23
8.4 Input/Output 23
8.5 Clock 24
8.6 Termination 24
8.7 Abortion 25

9 Visualization 26

References **27**

List of Functions, Types and Commands **28**

Index **29**