Distributed Maple — User and Reference Manual (V 1.1.12)

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

July 6, 2001

Abstract

Distributed Maple is a system for writing parallel programs in the computer algebra system Maple. It allows to create concurrent tasks and have them executed by Maple kernels running on different machines of a network. The system consists of two components:

1. A Java class library which implements a general purpose communication and scheduling mechanism for distributed applications.
2. A binding that allows to access the Java scheduler from Maple and implements a parallel programming model on top.

The system is portable to most kinds of computer networks and can be freely downloaded from

`http://www.risc.uni-linz.ac.at/software/distmaple`

Contents

1 Introduction **4**

2 Examples **5**

2.1 A First Example 5

2.2 Distributed Factorization of Integers 5

2.3 Distributed Factorization (Divide and Conquer) 7

2.4 Shared Data Objects 9

2.5 Visualization 10

3 Programming Interface **12**

3.1 Session Setup 12

`dist[initialize]`: Initialize a Session 12

`dist[initializep]`: Initialize a Session in Parallel 14

`dist[terminate]`: Terminate a Session 14

3.2 Basic Model 15

`dist[all]`: Execute a Command 15

`dist[start]`: Create a Task 15

`dist[wait]`: Wait for a Task Result 15

`dist[select]`: Wait for Some Task Result 16

3.3 Shared Data 16

`dist[data]`: Create a Shared Object 16

`dist[put]`: Write Data into a Shared Object 16

`dist[get]`: Get Data from a Shared Object 16

`dist[clear]`: Clear a Shared Object 17

`dist[process]`: Execute a Task by a Separate Process 17

`dist[place]`: Execute a Task by a Placed Process 17

3.4 Miscellaneous 17

`dist[delete]`: Delete a Task 17

`dist[flush]`: Flush the Result Caches 18

`dist[reset]`: Reset the Session 18

`dist[abort]`: Abort the Session 18

`dist[logging]`: Switch Logging Status 19

3.5 Visualization 19

`dist[visualize]`: Create Visualization Window 19

`dist[trace]`: Create a Trace File 20

`dist[label]`: Create a Label 20

4 System Configuration **21**

5 Implementation **24**

5.1 Session Startup 24

5.2 Connection Servers 25

5.3 Session Operation 26

A The Scheduler Protocol **28**

A.1 Protocol Variables 29

A.2 Message Passing 30

A.3 The Protocol 31

1 Introduction

This report describes the use of a system for writing distributed Maple applications and sketches its implementation. Maple is a popular computer algebra system for which a large number of application packages have been written [3]. In particular, the Maple library CASA of algorithms for solving problems in algebraic geometry has been developed in our institute [5]. Our goal is to provide an environment that makes it easy to implement parallel algorithms in Maple and that can be easily installed in any environment in order to facilitate the distribution of these implementations. We intend to use this system in order to parallelize a significant number of CASA algorithms.

Among several approaches to parallel programming with Maple, the most recent ones were the following:

- Wang implemented a parallel algorithm for computing characteristic sets by running a number of Maple kernels on different machines of a local network [6]; data communication was realized by a simple mechanism, the reading and writing of shared files in a global network file system.
- On the basis of the parallel declarative language Strand88, Siegl developed ||Maple|| (read: "Parallel Maple") [4]. Strand served as the coordinating system for connecting multiple Maple kernels such that expressions could be passed from Strand to Maple for evaluation. A parallel algorithm was thus implemented on two layers as a parallel Strand program calling sequential Maple functions.
- Diaz and Kaltofen have developed an interface between Maple and their FoxBox system that provides parallel implementations of polynomial factorization algorithms [1]. Thus a Maple user can connect to FoxBox and access its functionality (which has been implemented in C++ using the messing passing library MPI).

Our own goal is to provide an environment where parallel programming is possible *within* Maple such that the mathematical programmer does not have to leave the familiar environment of the computer algebra system. Furthermore, the underlying technological basis should be "time-safe" and accessible such that the resulting code can be easily exchanged and used for a long time (Strand88 was only available in a few places and its vendor is now out of business).

We have tackled this goal by developing a simple but easily configurable communication and scheduling program that may be used to start and connect external application programs on different machines. This program is written in Java and can be executed on any machine running some implementation of the Java Virtual Machine. We have written a small Maple package that implements an interface to the scheduler and provides a high-level parallel programming model.

Both the Java scheduler (compiled Java classes) and the Maple interface (Maple source code) are freely available under the GNU Library Public License at

`http://www.risc.uni-linz.ac.at/software/distmaple`

2 Examples

2.1 A First Example

The screenshot below demonstrates the use of the system by a simple example:

```
deneb!1> maple
    |\^/|     Maple V Release 4 (Johannes Kepler University Linz)
._|\|   |/|_. Copyright (c) 1981-1996 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read `dist.maple`;
Distributed Maple V1.0 (c) 1998 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([[deneb,solaris], [iris,irix]]);
connecting deneb...
connecting iris...
                                     okay

> t1 := dist[start](int, x^n, x);
                                   t1 := 0

> t2 := dist[start](int, x^n, n);
                                   t2 := 1

> dist[wait](t1) + dist[wait](t2);
                                (n + 1)     n
                               x           x
                               -------- + -----
                                n + 1     ln(x)

> dist[terminate]();
                                     okay

> quit;
```

In this session, we first load the package *dist.maple* which prints a copyright message. Then we call the procedure `dist[initialize]` with an argument list that tells the system to start a distributed session connecting (in addition to the current Maple kernel) two Maple kernels running on machine `deneb` of type `solaris` and on machine `iris` of type `irix`, respectively. After the connection has been established and the distributed session has been initialized, two calls of `dist[start]` create two tasks computing `int(x^n, x)` and `int(x^n, n)` respectively. The two `dist[wait]` tasks let the current execution block until the corresponding task results are available and then return these results. Finally the distributed session is closed by a call of `dist[terminate]`.

2.2 Distributed Factorization of Integers

The following program demonstrates how to compute the factorizations of a list of integers using the features of Distributed Maple. We may use this program in the context of a session as follows:

```
deneb!1> maple
    |\^/|     Maple V Release 4 (Johannes Kepler University Linz)
._|\|   |/|_. Copyright (c) 1981-1996 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read `dist.maple`;
Distributed Maple V1.0 (c) 1998 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([[deneb,solaris], [iris,irix]]):
connecting deneb...
connecting iris...
> main(20,2^90):
> dist[terminate]():
> quit:
```

The program itself is shown below:

```
# ------------------------------------------------------------------------
#
# distributed Maple
# distributed integer factorizations
#
# ------------------------------------------------------------------------

# ------------------------------------------------------------------------
# return a list of `length` elements each of which is a pair [i, f] where
# `i` <= `size` is a random integer and `f` is the factorization of `i`
# ------------------------------------------------------------------------
main := proc(length, size)
  local
    numbers, results, ti;

  # let machines load factorization library
  dist[all](`readlib(ifactors);`);

  # generate list of numbers
  numbers := rands(length, size);

  # compute result
  results := dfactors(numbers);

  # return list of argument/result pairs
  RETURN ([seq([numbers[i], results[i]], i = 1..length)]);
end:

# ------------------------------------------------------------------------
# return list of facorizations of `numbers` list
# ------------------------------------------------------------------------
dfactors := proc(numbers)
  local
    length, tasks, results, i, t, r;

  # length of list
  length := nops(numbers);

  # generate list of tasks
  tasks := [];
  for i from 1 to length do
    t := dist[start](ifactors, numbers[i]);
```

```
    tasks := [op(tasks), t];
  od;

  # wait for result values
  results := [];
  for i from 1 to length do
    r := dist[wait](tasks[i]);
    results := [op(results), r];
  od;

  # return results
  RETURN(results);
end:

# ------------------------------------------------------------------------
# return a list of `length` random numbers less than `size`
# ------------------------------------------------------------------------
rands := proc(length, size)
  local
    r, i, n, numbers;

  # random number generator
  r := rand(size);

  # generate list
  numbers := [];
  for i from 1 to length do
    n := r();
    numbers := [op(numbers), n];
  od;

  # return list
  RETURN (numbers);
end:
```

The `main` function of this program executes the procedure `dist[all]` which evaluates its argument string (denoting a command) on each Maple kernel connected to the session; thus on each kernel the Maple library *ifactors* is loaded. The core function `dfactors` creates a list of tasks each of which computes `ifactors(numbers[`i`])` for some i. The program then traverses the list waiting for each result in turn and constructs a corresponding list of results.

Above example demonstrates the simple strategy of *data parallelism* where each element of a central input data structure is processed in parallel and the task results are joined to form the desired output structure. Above example only uses *one-level parallelism* where each task computes a sequential function; however, as shown in the following section, a task may itself create other tasks.

2.3 Distributed Factorization (Divide and Conquer)

The following program solves the same problem as that one shown in the previous section; however it implements a divide and conquer algorithms to recursively decompose the input list into halves that are processed in parallel:

```
# ------------------------------------------------------------------------
#
# distributed Maple
# distributed integer factorizations (recursive version)
#
# ------------------------------------------------------------------------

# ------------------------------------------------------------------------
# return a list of `length` elements each of which is a pair [i, f] where
# `i` <= `size` is a random integer and `f` is the factorization of `i`
# ------------------------------------------------------------------------
main := proc(length, size)
  local
    numbers, results, ti;

  # let machines load factorization library and this file
  dist[all](`readlib(ifactors);`);
  dist[all](`read ``/home/distmaple/examples/ifactor2.maple``;`);

  # generate list of numbers
  numbers := rands(length, size);

  # compute result
  results := dfactors(numbers);

  # return list of argument/result pairs
  RETURN ([seq([numbers[i], results[i]], i = 1..length)]);
end:

# ------------------------------------------------------------------------
# return list of facorizations of `numbers` list
# ------------------------------------------------------------------------
dfactors := proc(numbers)
  local
    length, m, i, numbers1, numbers2, task1, result1, result2;

  # recursion base
  length := nops(numbers);
  if length = 0 then
    RETURN ([]);
  elif length = 1 then
    RETURN ([ifactors(numbers[1])]);
  fi;

  # divide problem
  m := trunc(length/2);
  numbers1 := [ seq(numbers[i], i=1..m) ];
  numbers2 := [ seq(numbers[i], i=m+1..length) ];

  # conquer in parallel
  task1 := dist[start](dfactors, numbers1);
  result2 := dfactors(numbers2);

  # combine results
  result1 := dist[wait](task1);
  RETURN ([op(result1), op(result2)]);
end:

# ------------------------------------------------------------------------
# return a list of `length` random numbers less than `size`
# ------------------------------------------------------------------------
```

```
rands := proc(length, size)
  local
    r, i, n, numbers;

  # random number generator
  r := rand(size);

  # generate list
  numbers := [];
  for i from 1 to length do
    n := r();
    numbers := [op(numbers), n];
  od;

  # return list
  RETURN (numbers);
end:
```

Since each of the tasks (executed on any Maple kernel connected to the session) needs to call the function `dfactors`, we call `dist[all]` to let each Maple kernel read the Maple file from a path in the global file system. The computation itself is pursued by splitting one recursive call of `dfactors(numbers1)` as a separate task, computing `dfactors(numbers2)` in the current task, waiting for the result of the first task, and then joining both results to the final result.

Above program demonstrates the application of *nested parallelism* where each task in turn may create additional tasks if the input is not simple enough to be solved directly.

2.4 Shared Data Objects

Distributed Maple allows a task to put data into a global pool for retrieval by other tasks.

```
d := dist[data]();
```

creates a new shared data object whose handle `d` may be communicated to other tasks. However, initially the data object is in an empty state; every attempt to retrieve the value of the object by

```
v := dist[get](d);
```

will block the corresponding task. Only after some task has written a value into the shared data object by

```
dist[put](d, v);
```

the corresponding tasks are released to continue execution. A shared data object may be reset into the empty state by

```
dist[clear](d);
```

and thus reused for further synchronization. However, even without intermediate clearing, the value of a shared data object may be reset by further `dist[put]` calls.

Since a shared data object may contain references to other shared objects, continuous communication between tasks becomes possible via "streams" of shared objects. For instance, the following code fragment starts a task executing `f(d)` where `d` is a shared data object whose value becomes either the empty list `[ ]` or a tuple `[v,r]` where `v` is the next value to be communicated and `r` is a shared data object that is constructed analogously.

```
d := dist[data]();
t := dist[start](f, d);
while not term(x) do
  r := dist[data]();
  dist[put](d, [x, r]);
  x := g(x);
  d := r;
od;
dist[put](d, []);
```

This concept allows to generate arbitrary networks of tasks that communicate with each other via communication streams. However, in such situations the standard scheduling mechanism (where a fixed set of Maple processes executes an arbitrary number of tasks) does not suffice any longer because it relies on an acyclic graph of task dependencies. If cyclic dependencies arise, `dist[process]` (with the same syntax as `dist[start]`) must be applied to have the task be executed by a separate process. This process is initialized with all the `dist[all]` calls that have been previously issued.

Then a construction like

```
s1 := dist[data]();
s2 := dist[data]();
t1 := dist[process](f, s1, s2);
t2 := dist[process](g, s2, s1);
```

becomes possible where `s1` may be used for the transmission of values from (the process executing) `f` to (the process executing) `g` and `s2` handles communication in the other direction. If the tasks were created by the use of `dist[start]`, then they might be scheduled on the same Maple process yielding a deadlock.

2.5 Visualization

The system supports two kinds of visualization features for investigating the dynamic behavior of a program:

- *Online Visualization:* In a separate window, the execution status of the machines participating in the session and the total number of active machines are displayed.

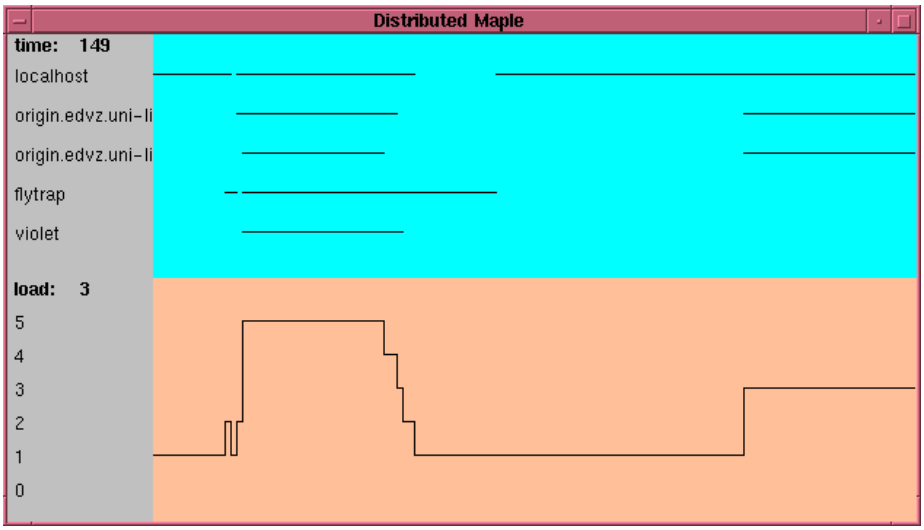

Figure 1: Online Visualization

- *Post-Execution Visualization:* During program execution, a trace file is generated that captures various kinds of events. After execution, a program generates from this file some figures, e.g., for inclusion into a paper.

In both cases, the results are generated from information that is locally available on the host running the interactive interface. The execution state of other machines is inferred from messages that are received and processed by the host with some delay. The visualizations therefore represent the status of the distributed system as experienced by the local host which does not necessarily reflect the actual status of the system at any time.

For online visualization, execution of a command like

```
dist[visualize](600, 300, 2, 120);
```

creates a window as depicted in Figure 1. This window of size 800×450 displays in its upper half the execution status of each machine during the last 120 seconds with probes taken every 2 seconds. The diagram in the lower half of the window depicts the number of machines active at any moment (as experienced by the local host). Calling this command several times with varying parameters yields different windows with different execution snapshots. Windows may be individually resized and closed (by the "Close" button in the window frame).

For post-execution visualization, a command like

```
dist[trace](file);
```

has to be executed *before* the `dist[initialize]` command. This causes the system to log relevant events in a file *file* stored on the local host. *After* the `dist[terminate]` command has been executed, one may run the Java program `visual.Main`, e.g.,

```
% java visual.Main file initialize terminate 100 100
```

This generates some files *file-∗.fig* each of which contains a diagram of size 100×100 mm that visualizes some aspect of the program's runtime behavior as seen in Figure 2: the first diagram denotes by horizontal bars those times when a machine was active (with the short vertical lines exhibiting particular events); the second diagram displays the number of machines active at a time; the last diagram shows from when to when and by which machine each task was executed. The numbers at the bottom denote the range of the visualization in milliseconds; the vertical dashed lines represents execution labels (see below).

The files are in the Fig 3.2 format; corresponding programs such as "xfig" may be used to process them and to generate images in various formats.

The (predefined) labels `initialize` and `terminate` in the call of `visual.Main` denote the range of the visualization. By calling the command

```
dist[label](key);
```

during program execution, additional labels ("`key`") are generated. These labels create corresponding vertical dashed lines in the diagrams and may be used for restricting the visualization range.

3 Programming Interface

The start of a distributed Maple session requires the loading of the Maple package *dist.maple*.

```
deneb!1> maple
    |\^/|     Maple V Release 4 (Johannes Kepler University Linz)
._|\|   |/|_. Copyright (c) 1981-1996 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read `dist.maple`;
Distributed Maple V1.0 (c) 1998 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
```

For debugging purposes, the Maple package *seq.maple* can be loaded instead. This package provides the same functions as *dist.maple* but implements them in a sequential way.

The following sections describe each routine of the distributed Maple section in turn by its synopsis, a verbal description, and a short example.

3.1 Session Setup

`dist[initialize]`: Initialize a Session

```
dist[initialize](machineList)
```

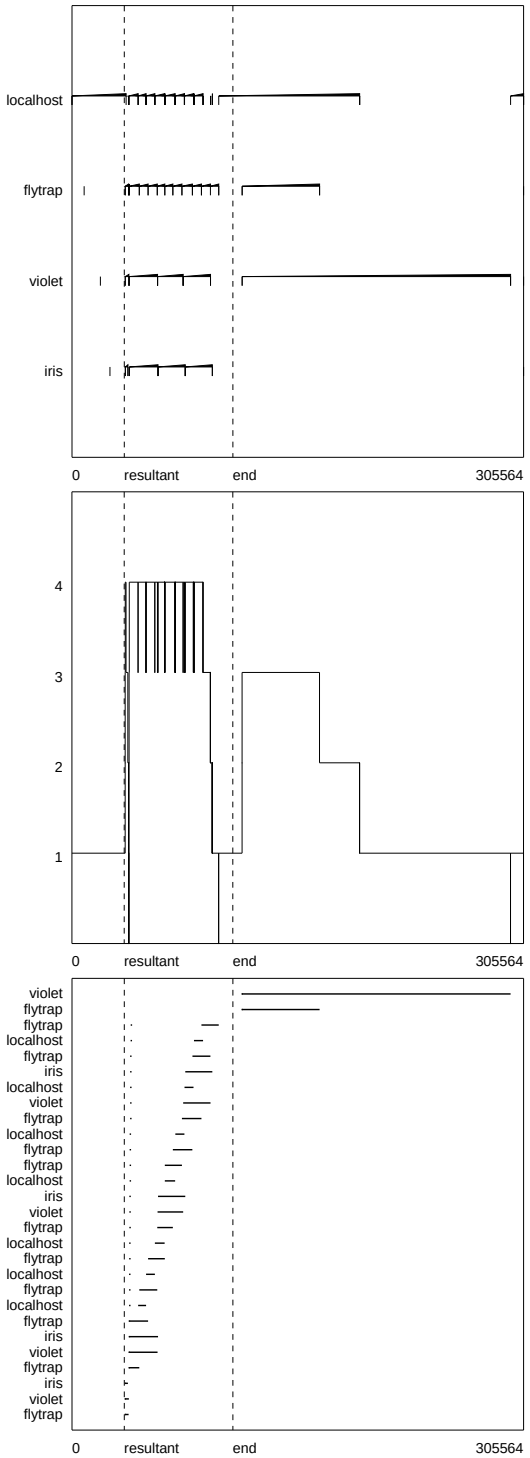

Figure 2: Post-Execution Visualization

This call initializes a distributed Maple session by starting remote processes and establishing the corresponding network connections. Each element of *machineList* is a list [*machine*, *type*] where *machine* is the domain name (or numeric Internet address) of a machine and *type* is the corresponding key in *dist.systems* (see Section 4) that describes how to contact this machine. If *machineList* is the empty list, only the current Maple is subsequently employed.

If the connection to some machine fails, the other connections are nevertheless established. If a previous session is active, then that session is terminated before the new session is established.

See also `dist[initializep]`.

```
> dist[initialize]([[deneb, solaris],
                    [iris.risc.uni-linz.ac.at, irix]]);
Connecting deneb...
connecting iris.risc.uni-linz.ac.at...
                                        okay
```

`dist[initializep]`: Initialize a Session in Parallel

```
dist[initializep](machineList)
```

This call initializes a distributed Maple session by starting remote processes and establishing the corresponding network connections. Each element of *machineList* is a list [*machine*, *type*] where *machine* is the domain name (or numeric Internet address) of a machine and *type* is the corresponding key in *dist.systems* (see Section 4) that describes how to contact this machine. If *machineList* is the empty list, only the current Maple is subsequently employed.

In contrast to `dist[initialize]`, this function establishes all connections *in parallel*, which is much faster if the number of machines is large (however, it is recommended to use the sequential version first to find errors in the machine setup). If the connection to some machine fails, *none* of the connections is established. If a previous session is active, then that session is terminated before the new session is established.

```
> dist[initializep]([[deneb, solaris],
                    [iris.risc.uni-linz.ac.at, irix]]);
connecting all machines in parallel...
                                        okay
```

`dist[terminate]`: Terminate a Session

```
dist[terminate]()
```

This call terminates the current distributed Maple session by destroying all Maple kernels (apart from that one connected to the user interface) and closing all network connections. This procedure should be called before quitting the interactive Maple session.

```
> dist[terminate];
                                        okay
```

3.2 Basic Model

`dist[all]`: Execute a Command

```
dist[all](command)
```

This call executes *command* on every Mapel kernel connected to the distributed session (including that one connected to the user interface).

```
> dist[all](‘load ifactors;‘);
                                        okay
```

`dist[start]`: Create a Task

```
task := dist[start](fun, args...)
```

This call creates a *task* that computes *fun*(*args*...) on some Maple kernel connected to the distributed session.

All created tasks are scheduled on a fixed number of Maple kernels. If a task is suspended (e.g. by `dist[wait]`), then another task may be executed on the respective kernel such that in general a kernel hosts a whole *stack* of tasks of which only the task on top is active.

This scheme is deadlock-free if *args* does not contain other tasks or shared objects. Otherwise, do not use `dist[start]` but `dist[process]`.

```
> t := dist[start](int, x^n, x);
                                        t := 0
```

`dist[wait]`: Wait for a Task Result

```
result := dist[wait](task)
```

This call suspends the current execution until *task* has completed and then returns its *result*. While the current execution is suspended, other tasks can be executed by the corresponding Maple kernel.

```
> t := dist[start](int, x^n, x):
> r := 1 + dist[wait](t);
                                          (n + 1)
                                         x
                                  r := 1 + --------
                                           n + 1
```

`dist[select]`: Wait for Some Task Result

```
selector := dist[select](taskList)
```

This calls supsends the current execution until the execution of some task in *taskList* has completed; it then returns a *selector* for this task, i.e., a list [*index*, *result*] where *index* denotes the position of the task in the list and *result* is its result.

```
> l := [dist[start](int, x^n, x), dist[start](int, x^n, n)]:
> s := dist[select](l):
> t := l[s][1];
                                        t := 1

> r := l[s][2];
                                            (n + 1)
                                           x
                                    r := --------
                                            n + 1
```

3.3 Shared Data

`dist[data]`: Create a Shared Object

```
d := dist[data]()
```

This call creates a shared object d that may be used to communicate data between different tasks. The object is initially *empty*; any attempt to get a value from d blocks the corresponding task.

```
> d := dist[data]();
                                        d := 0
```

`dist[put]`: Write Data into a Shared Object

```
dist[put](d, v)
```

This call puts the value v into the shared object d and releases all tasks blocked on an attempt to get a value from this object.

```
> dist[put](d, x+y);
                                         okay
```

`dist[get]`: Get Data from a Shared Object

```
v := dist[get](d)
```

This call returns the value v of the shared object d; if d is still empty, the current task is blocked until a value is put into d.

```
> v := dist[get](d);
                                      v := x + y
```

`dist[clear]`: Clear a Shared Object

```
dist[clear](d)
```

This call resets the shared object d into the empty state. Further attempts to get a value from d blocks the corresponding task.

```
> dist[clear](d);
                                        okay
```

`dist[process]`: Execute a Task by a Separate Process

```
task := dist[process](fun, args...)
```

This call creates a *task* that computes *fun*(*args*...) on a separate (possibly newly created) Maple kernel. This kernel has executed all the commands issued through previous calls of `dist[all]` in the corresponding order. During the execution of the task, no other task is scheduled on the kernel.

One must use this operation (rather than `dist[start]`) if *args* contains other tasks or shared objects.

The execution of `dist[process]` may involve a larger runtime and especially memory overhead than the execution of `dist`[start].

```
> t := dist[process](int, x^n, x);
                                        t := 0
```

`dist[place]`: Execute a Task by a Placed Process

```
task := dist[place](machine, fun, args...)
```

This call behaves like `dist[process]` except for the additional argument *machine* which must denote an integer number in the range 0 to N where N is the number of machines given in `dist[initialize]`. The corresponding task does not participate in load balancing but is *immediately* placed on the denoted machine (where 0 denotes the original host) and is executed by a separate process.

```
> t := dist[place](0, int, x^n, x);
                                        t := 0
```

3.4 Miscellaneous

`dist[delete]`: Delete a Task

```
dist[delete](task)
```

This call deletes *task* from the distributed Maple session. Subsequently, *task* must not be referenced any more in the current session.

If the corresponding task has already completed execution, its result is deleted from the system. If the task has not yet started execution, it will probably not get executed any more. If the task is under execution, it will continue until completion (but its result will be deleted afterwards).

```
> dist[delete](t);
                                        okay
```

`dist[flush]`: Flush the Result Caches

`dist[flush](`*code*`)`

This call flushes the internal result caches of the system. If *code* is 0, result caching is subsequently switched off. If *code* is 1, result caching remains switched on.

```
> dist[flush](0);
                                        okay
```

`dist[reset]`: Reset the Session

`dist[reset]()`

This call resets the session, i.e., it deletes all tasks and their results, and flushes all caches; in particular, it also destroys and restarts all Maple kernels connected to the system (except for the kernel connected to the user interface). However, the network connections remain open during this operation; this operation is therefore faster than a termination and re-start of the session.

```
> dist[reset]();
                                        okay
```

`dist[abort]`: Abort the Session

`dist[abort](`*message*`)`

This call aborts the session by terminating all Maple kernels (except for the kernel connected to the user interface) and closing all network connections. The denoted *message* is printed at the user interface and written to file */tmp/dist.error* (if possible).

```
> dist[abort](`invalid type`);
Error, (in dist/abort) invalid type
dist error: invalid type
program aborted
```

`dist[logging]`: Switch Logging Status

`dist[logging](`*code*`)`

If *code* is 1 or 2 or 3, result logging is enabled. In the current directory, a subdirectory *logging* is created into which computed task results and shared data values are written. If the computation fails abnormally and the session is restarted, the logged results will be used instead of computing the corresponding tasks again. If the session is terminated normally, the logging directory is erased.

If *code* is 1 ("fast mode") logging is cheap but it may happen that after a session failure the restarted session breaks with an error message. If *code* is 2 ("safe mode") logging is more expensive, but the restarted session is guaranteed to continue with the logged results. If *code* is 3("rescheduling mode"), machines to which the root machine loses connectivity are automatically excluded from the session; all tasks being executed on these machines are rescheduled to other machines.

If *code* is 0, logging is switched off.

Unless you know what you are doing, switch logging on at the beginning of a session (if at all) and leave the logging status unchanged till the end of the session.

```
> dist[logging](2);
                                        okay
```

3.5 Visualization

`dist[visualize]`: Create Visualization Window

`dist[visualize](`*width*`, `*height*`, `*updates*`, `*range*`)`

This call creates a window of dimension *width*×*height* that displays at most *range* seconds of program execution with samples of the execution status taken every *updates* seconds. In the upper half of the window, a horizontal bar denotes those times when a machine is active executing a task; in the lower half of the window, the number of machines active at a time is displayed.

The command may be called multiple times with different parameters yielding windows that may be individually resized and closed. For closing a window, use the "Close" button in the window frame. *Do not use the "Exit" button; because this terminates the scheduler and yields an undefined Maple state.*

```
> dist[initialize]([[deneb,ultrasparc], [flytrap, r5000]]):
> dist[visualize](600, 300, 1, 120);
                                        okay
```

`dist[trace]`: Create a Trace File

```
dist[trace](filename)
```

This call must be issued *before* `dist[initialize]` is executed. It triggers the generation of a file *filename* into which relevant events are logged.

```
> dist[trace](tracefile);
                                        tracefile
> dist[initialize]([[deneb,ultrasparc], [flytrap, r5000]]):
```

After `dist[terminate]` has been executed, this file may be used to create diagrams by calling on the command line

```
java visual.Main filename start end width height
```

This creates a number of files in the Fig 3.2 format each of which contains a diagram of dimension *width*×*height* millimeters:

- *filename-machines.fig* displays by a horizontal bar those times when a machine was active executing a task; a short vertical bar denotes some load-changing event occurring at that particular time (e.g., waiting for a new task to be executed);
- *filename-util.fig* displays the number of machines active at a time;
- *filename-tasks.fig* displays by a horizontal bar from which begin time to which end time and by which machine a task was executed; it is *not* shown at which times the execution of a task was interrupted by waiting for the result of another task.

At the bottom of each diagram the visualization time in milliseconds is shown.

The labels *start* and *end* restrict the range of the visualization; there are two predefined labels `initialize` and `terminate` bounding the execution. Other labels may be generated by `dist[label]`.

```
% java visual.Main tracefile initialize terminate 100 100
processing tracefile...
creating tracefile-machines.fig...
creating tracefile-util.fig...
creating tracefile-tasks.fig...
done.
```

`dist[label]`: Create a Label

```
dist[label](name)
```

This call creates a label called *name* that may be used in the visualization of a trace file (see Subsection 3.5). Each call of `dist[label]` must use a different *name*.

```
> dist[label](key1);
                                        okay
```

4 System Configuration

When `dist[initialize](...)` is called, the Maple kernel starts the Java scheduler *dist.Scheduler* and establishes a permanent connection to this process. The scheduler reads the file *dist.systems* which contains a list of entries of the following forms separated by empty lines:

```
key
starter
scheduler
command
initializer
speed
maxload
minload
```

The fields have the following interpretation:

key is the (arbitrary but unique) name for the class of machines described by this entry.

starter denotes the command to call in order to create a process on the denoted machine (typically `rsh`).

scheduler is the command to execute on the remote machine in order to create an instance of the scheduler (typically `java dist.Scheduler` with some `-classpath` option).

command is the command to fork by the scheduler on the remote machine in order to perform the computation (typically `maple -q`).

speed is a performance factor (an integer number greater than or equal 1) which is used by the scheduler for load distribution. The higher the number is, the faster this machine is expected to be. The scheduler will assign tasks to faster machines prior to slower machines.

maxload denotes the maximum number of executable but not yet started tasks that the machine may hold at a time. If the scheduler on the frontend machine knows that the number is below this treshold, it may forward newly created tasks to the machine. If *maxload* is -1, the scheduler may *always* forward newly created tasks to the machine (which may result in poor load balancing).

If you don't know better, use 0 in this place.

minload denotes the minimum number of executable but not yet started tasks that the machine should hold at a time. If the number falls below this treshold, the machine informs the scheduler on the frontend machine about this (and thus asks for new tasks). If *minload* is 0, the machine informs the scheduler *immediately* whenever it starts an executable task (which may result in many messages).

If you don't know better, use 0 in this place.

The configuration file must contain an entry whose key is `localhost`. In this entry only the fields *scheduler* and *command* are of relevance; they are used for the creation of additional kernels on the local host.

An alternative form of the *starter*/*scheduler* field may be used as well:

starter may be an integer number. In this case, this number is the *port* of the denoted machine on which a connection server is listening which will in turn start the desired process (see the next section).

scheduler is the *password* that was given to the connection server at its start.

Below you see some sample entries of *dist.systems*, once with a remote startup command (`rsh`) once with a port number (`9966`):

```
solaris
rsh
java -classpath /usr/local/lib/jdk/lib/classes.zip:/home/info/www/
 software/distmaple dist.Scheduler
maple -q
`dist/isclient` := true: read `/home/info/www/software/distmaple/
 dist.maple`:
1
1
0

solaris2
9966
mypasswd
maple -q
`dist/isclient` := true: read `/home/info/www/software/distmaple/
 dist.maple`:
1
1
0
```

The second form assumes that on a machine of kind `solaris2` a connection server (see the next section) is running, started by e.g.

```
java dist.Scheduler server 9966 "java dist.Scheduler"
mypasswd
```

Tip: You can monitor the communication between a remote scheduler and its external application program by calling in the `command` field a wrapper script that copies the standard input respectively standard output of the application program to some files:

```
#!/bin/sh
tee /tmp/maple.in.$$ | maple -q | tee /tmp/maple.out.$$
```

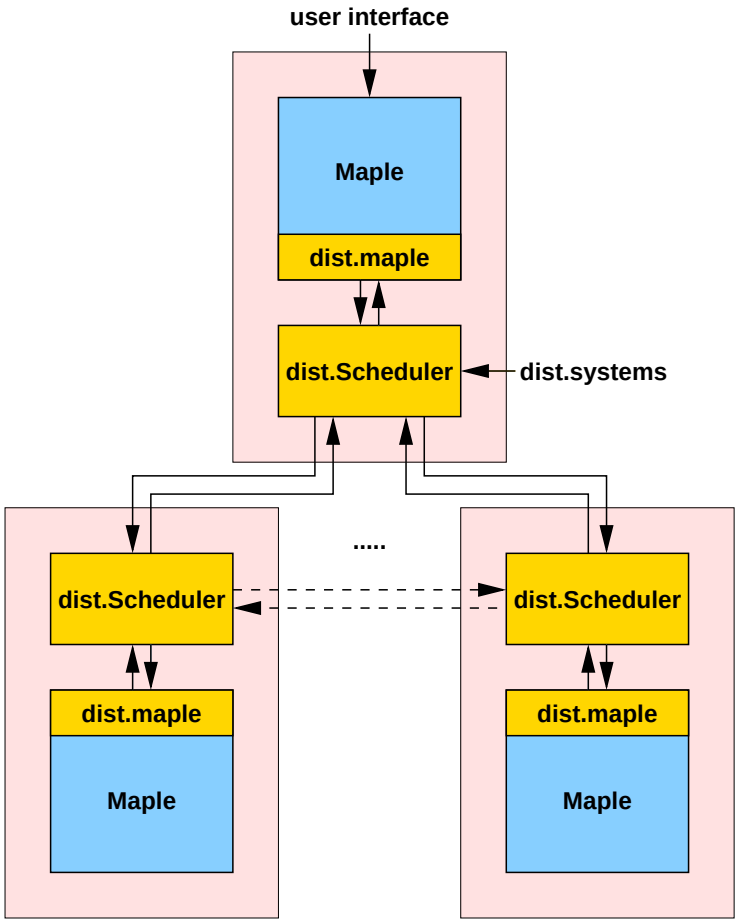

Figure 3: A Distributed Maple Session

5 Implementation

The architecture of a distributed Maple session is sketched in Figure 3. Each node connected to the session consists of two components:

Scheduler A scheduler program manages the node interaction and schedules tasks among nodes. This program is implemented by a Java class library with main class *dist.Scheduler* and is independent of Maple. The initial scheduler process reads all application-specific information from the configuration file *dist.systems* and may then start instances of the scheduler on other machines with which it communicates via sockets.

Maple Interface The package *dist.maple* running on each Maple kernel implements the interface between Maple and the scheduler. Communication between both components is based on pipes (named pipes for the Maple kernel connected to the user interface and standard input/output streams for the backend kernels).

During a session, additional socket connections between remote scheduler instances are dynamically established on demand.

5.1 Session Startup

When the Maple kernel connected to the user interface calls `dist[initialize]`, it forks an *dist.Scheduler* process on the local machine, typically by a call of

```
java dist.Scheduler local /tmp/maple.in.pid /tmp/maple.out.pid
```

where the file paths denoted by the command arguments refer to some files or named pipes (an argument '`-`' would denote the standard input respectively standard output stream of the process).

The scheduler process thus runs in mode "local" and reads the configuration file *dist.systems* in which all system-specific information about each class of machine is stored. The scheduler receives from the Maple kernel the sequence of machines to be connected to the session. It spawns a process on the corresponding machine respectively contacts a server on that machine for spawning the process (see the next subsection). Subsequently the local scheduler waits for a connection request of the remote instance.

The remote process is an instance of *dist.Scheduler* running in mode "remote", typically created by a call of

```
java dist.Scheduler remote machine port
```

where *machine* is the domain name of the host on which the local scheduler is waiting on machine port *port* for a connection request of the remote instance.

Furthermore, during session startup, each remote process is informed about each other. If a remote process needs to communicate with another such process, a "peer" connection between remote processes is dynamically established.

After a connection between local and remote scheduler has been established, the local instance forwards to the remote instance the command string that the remote instance uses to fork the external application process, typically

```
maple -q -v
```

It also sends the string that is used to initialize the process, typically

```
`dist/isclient` := true: read `path/dist.maple`:
```

This invokes a server loop on the Maple kernel that is ready to receive task descriptions from the scheduler process, execute the denoted tasks, and return the results of these tasks. In the context of Maple, tasks are Maple expressions to be evaluated; they can be transmitted in any external format that Maple can convert to its internal form (currently either readable Maple syntax or Maple's *.m* file format may be used).

5.2 Connection Servers

An alternative form of session startup is supported where the local scheduler instance does not spawn the remote instance itself but instead contacts a server running on the corresponding machine which spawns the instance. Such a server is started on the remote machine by invoking typically

```
java dist.Scheduler server port command
```

This scheduler instance runs in mode "server" and listens on machine port *port* for connection requests from the local scheduler (if *port* is 0, any free port is used). The server responds to such a request by forking the program denoted by *command* (typically `"java start.Scheduler"`).

For security reasons, the server is protected by a password that the user has to interactively type in when she starts the server and that the local scheduler has to provide when it requests a connection (the password is read from *dist.systems*, see the previous section).

A typical invocation of the server thus is

```
java dist.Scheduler server 9966 "java dist.Scheduler"
mypasswd
```

The server responds by printing the port that it uses (9966) and subsequently runs in a forever loop waiting for connection requests.

Please note that the use of the connection server is a potential security hazard. Any user on the Internet that knows the password of the connection server can contact the server and get on the corresponding machine all privileges of the account under which this server is running.

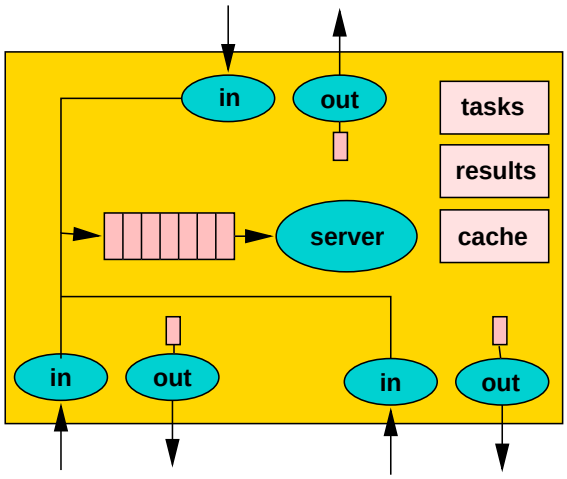

Figure 4: Scheduler Architecture

5.3 Session Operation

After a distributed session has been established, the scheduler accepts tasks from the Maple process and schedules these tasks among any node connected to the session. A task is a pair (*taskid*, *exp*) where *taskid* is an integer number identifying the task and *exp* is a string to be submitted to the Maple process. Initially, the scheduler informs each Maple process about the range of task identifiers it may use for assignment to new tasks such that each node in the system can independently create new tasks.

The following paragraphs sketch the operation of the scheduler; the protocol of interactions between a scheduler and its external application process (e.g. Maple kernel) is formally specified in the appendix.

Multi-Threading The scheduler is implemented by a number of concurrent threads as shown in Figure 4. A central server thread sequentially processes messages that were received from any input channels and put into a central buffer by a thread listening on that channel. The scheduler thread puts every output message into a buffer that is read by a thread writing the message on the corresponding output channel.

Watchdogs An additional watchdog thread in each remote scheduler controls in regular intervals if messages have been received from the local scheduler. If after the last control period no message has been received, the watchdog sends a "ping" message to the local scheduler. If during the next control period no reply has been received, the watchdog assumes that the connection is broken and aborts the external application process and the scheduler. Thus we ensure that failed sessions do not lead to stalled remote processes.

Task Results Each scheduler holds a local task queue, a result table, and a result cache. When a new task is created by the external application process, a corresponding "empty" entry is created in the result table (which will be filled by the result the result has been executed). Thus the result of a task is stored on that node of the distributed session where the task has been created; from the identifier of a task, we can uniquely determine the node holding the task and route requests for its result appropriately.

Task Scheduling All remote schedulers send the tasks created by the corresponding application processes to the local scheduler; from here they are centrally managed and distributed among all session nodes. Currently a simple load balancing scheme is used where the local scheduler assigns new tasks to remote schedulers until the number of not yet started tasks reaches an upper treshold (specific to the node class) and a remote scheduler asks for new tasks whenever the number of not yet started tasks falls below a lower treshold.

Result Caches When an application process asks for the result of a task, this request can be locally satisfied if the corresponding scheduler is the holder of the task result. Additionally each scheduler holds a cache of all results that it has "seen" (because the task was executed on this node or, in case of the local scheduler, the task result has passed this node on its way from the node that computed the result to the node that actually holds the result).

Recursive Server Loops If the request of a task result cannot be satisfied (because the result is not yet available or another node holds the task result), the requesting application process receives another task for execution. The Maple kernel implements this request by invoking the server loop recursively and thus starting a new "level" of task execution. If after the completion of this task the requested result is available, this result is delivered. Therefore every Maple kernel (also the kernel connected to the user interface) permanently computes as long as sufficiently many tasks are available.

Additional Maple Kernels The execution of `dist[process]` causes an additional Maple kernel to be created and attached to some instance of a scheduler. The new kernel is appropriately initialized (also by executing all commands previously issued through `dist[all]` and reserved for the execution of the denoted task. After this task has terminated, the process is preserved for later use.

Peer Connections Initially, there exist only connections between the frontend node to each remote node. However, all nodes know of each other, i.e., of a machine address and the number of a port on which a thread is listening for connection requests. When a node needs to send a message to one of its peers, thus a direct connection is established for the message transfer. The connection remains persistent through the rest of the session such that no more startup overhead is involved.

Peer connections have been introduced in version 1.10 of the scheduler; they can be switched off by the command line option `-n`.

Message Buffers By default, every transmitted object has to be stored in the memory of the scheduler process, temporarily (auxiliary objects that are soon reclaimed by garbage collection) but perhaps also permanently (task results which can be only cleared by the user command `dist[clear]`clear). If this becomes a problem (e.g. because the Java Virtual Machine runs out of memory), the content of a large message can be automatically evacuated to disk: the message is read in pieces of fixed size which are written to a temporary file (which is read again when the message content has to be forwarded to another process).

Message buffers have been introduced in version 1.10 of the scheduler; they are disabled by default but can be enabled by the command line option `-b`.

References

[1] Angel Diaz and Erich Kaltofen. FoxBox: A System for Manipulating Symbolic Objects in Black Box Representation. In O. Gloor, editor, *ISSAC 98 International Symposium on Symbolic and Algebraic Computation*. ACM Press, New York, 1998. http://www.math.ncsu.edu/~kaltofen/bibliography/98/DiKa98.ps.gz.

[2] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994. http://www.acm.org/pubs/citations/journals/toplas/1994-16-3/p872-lamport/.

[3] MapleSoft. Maple V Release 5, April 1998. http://www.maplesoft.com/maplevr5.html.

[4] Kurt Siegl. Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–186, San Diego, California, May 19-22, 1993. ACM Press, New York. ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1993/93-08.ps.gz.

[5] Quoc-Nam Tran and Franz Winkler. CASA Reference Manual (Version 2.3). Technical Report 97-33, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, October 1997. ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1997/97-33.ps.gz.

[6] Dongming Wang. On the Parallelization of Characteristic-Set-Based Algorithms. In H.P. Zima, editor, *Parallel Computation — First International ACPC Conference*, volume 591 of *Lecture Notes in Computer Science*, pages 338–349, Salzburg, Austria, September 30–October 2, 1991. Springer, Berlin.

A The Scheduler Protocol

The following specification does not yet incorporate the messages used for triggering the creation of a trace file (`trace`*), for the generation of a trace label*

(`label`), for the creation of a visualization window (`visual`), for creating a shared data object (`data`), for writing such an object (`put`), and for having a task be executed by a separate process (`proc`).

In this section we formally specify the main part of the protocol by which an external application program communicates with the scheduler. The specification is expressed in the language of temporal logic [2] in the form

$$I \Rightarrow \Box\ (A_0 \vee \ldots \vee A_n)$$

which reads "if initially the condition I holds, then in every step one of the actions A_i is executed". Here we consider as an execution step the sending/receiving of a message from/to the scheduler; thus between two actions arbitrary real-time delays may occur.

An action is represented by a predicate $A_i(V, V')$ that describes the relationship between the values V of the program variables before the action was performed and the corresponding values V' after completion of the action. An action

$$\begin{array}{l} Action \equiv \\ \quad \wedge\ \mathrm{X'} = \mathrm{X}+1 \\ \quad \wedge\ \mathrm{Y'} = \mathrm{Y} \end{array}$$

therefore increases the value of the program variable X by 1 and does not change the value of Y. As a convention, we omit all predicates that only demand that a variable does not change its value, i.e., whenever there is no other predicate $V' = \ldots$ for some variable V in an action, we implicitly assume $V' = V$.

A.1 Protocol Variables

The protocol is expressed in terms of the following program variables:

S*: *String denotes the current status of the program.

T*: *String denotes a substatus, i.e., a refinement of the current status; T may take the special constant 'null'.

version*: *String identifies the version of the protocol that the application program understands.

taskStack*: *Array[Message] denotes the stack of task messages that were received by the external application program and for which it has not yet delivered corresponding results; a result may be only delivered for the task on the top of the stack.

taskDepth*: *Natural denotes the number of elements on the task stack.

waitStack*: *Array[Message] denotes the stack of wait messages that were sent by the external application program and for which it has not yet received corresponding replies; a reply can be only received for the wait message on the top of the stack.

waitDepth*: *Natural denotes the number of elements on the wait stack.

taskId*: *TaskId describes the identifier used to denote the next task to be created.

taskMin*: *TaskId describes the lower bound of the range of identifiers that may be used to denote new tasks.

taskMax*: *TaskId describes the upper bound of the task identifier range.

We use the following types in variable annotations of the form v: T to assert that v must be an element of the set T:

String An arbitrary sequence of characters not including the newline character.

Command A string that may be evaluated as a command by the application program.

Expression A string that may be evaluated as an expression by the application program.

Integer An integer number.

Natural A non-negative integer number.

TaskId A natural number that identifies a task.

***Array*[*T*]** An indexable array of elements of type T.

Message A record m such that m.key is a string denoting the type of a message and m.body is an array of strings.

In our protocol specification, Integers are implicitly converted to strings (using thee textual representation of an integer as its string encoding). Likewise we assume that there is some encoding of commands and expressions as strings.

A.2 Message Passing

The application program communicates with the scheduler by sending respectively receiving sequences of messages via some input respectively output channel. A message is written to a channel as a sequence of strings seperated by newline characters such that the first string denotes the message key and the remaining strings denote the body of the message. We use in our specification the predicate

$send(m, n)$

to denote the action that writes the sequence of lines

m.key
m.body[0]
...
m.body[$n-1$]

to the output channel. Likewise, the predicate

$$receive(m,\ n)$$

denotes the action of writing n+1 lines from the input channel and constructing a corresponding message m. The actions

$$write(s)$$
$$reads$$

write/read a string s to/from the output channel.

A.3 The Protocol

The protocol is based on the following condition that describes the initial values of some program variables:

$$\begin{array}{l} Init \equiv \\ \quad \wedge\ T = \text{null} \\ \quad \wedge\ taskDepth = 0 \\ \quad \wedge\ waitDepth = 0 \end{array}$$

The protocol is then specified by the following temporal formula:

$$\begin{array}{l} Protocol \equiv \\ \quad \wedge\ (Init \wedge\ S = \text{initExecuting}) \Rightarrow\ \Box\ \text{Program} \\ \quad \wedge\ (Init \wedge\ S = \text{initServer}) \Rightarrow\ \Box\ \text{Program} \end{array}$$

This formula covers the two cases of an external application program that starts its life with the execution of a program (such as the Maple kernel connected to the user interface) respectively of a program that starts its live as a server for evaluating tasks (such as the Maple kernels that are remotely started).

The following predicate describes the individual actions that may be performed; as a general convention we parameterize each predicate over the value of the successor state S':

$$\begin{array}{l} Program \equiv \\ \quad \vee\ (program\ entry) \\ \qquad \wedge\ S = \text{initExecuting} \\ \qquad \wedge\ initialize(\text{executing}) \\ \quad \vee\ (server\ entry) \\ \qquad \wedge\ S = \text{initServer} \\ \qquad \wedge\ initialze(\text{server}) \\ \quad \vee\ (execute\ a\ program) \\ \qquad \wedge\ S = \text{executing} \\ \qquad \wedge\ (actions) \end{array}$$

$\vee$ *sendTask*(executing)
$\vee$ *sendWait*(server)
$\vee$ *sendNWait*(server)
$\vee$ *sendAll*(executing)
$\vee$ *sendQuit*(terminate)
$\vee$ *sendAbort*(terminate)
$\vee$ *sendStart*(executing)
$\vee$ *sendDelete*(executing)
$\vee$ *sendFlush*(executing)
$\vee$ *sendReset*(executing)
$\vee$ *sendResult*(server)
$\vee$ (*run as a server*)
$\wedge$ S = server
$\wedge$ (*actions*)
$\vee$ *receiveTask*(executing)
$\vee$ *receiveAll*(server)
$\vee$ *receiveQuit*(terminate)
$\vee$ *receiveAbort*(terminate)
$\vee$ *receiveResult*(executing)

When the program is in status "executing", it may switch to status "server" by sending a (n)wait request (which may result in another task to be evaluated if the request cannot be immediately satisfied) or by sending the result of a task (that was previously received when in server mode). When the program is in status "server", it may switch to status "executing", if it receives a new task to execute or if it receives the result of a task it has previously sent a request for. In status "server", it also has to expect "all" messages (to evaluate some command) respectively "quit" or "abort" messages (to terminate execution).

The individual actions are described in the following sections.

Initialization Actions

When starting execution, the application program first makes sure that the scheduler understands the same version of the protocol. This is achieved by writing the version string to the output channel, receiving a corresponding string from the input channel and comparing both. If they do not match, execution is aborted. Otherwise, the range of task identifiers that may be used for the naming of new tasks is received and stored.

initialize(C) $\equiv$
$\vee$ (*send version*)
$\wedge$ T = null
$\wedge$ *write*(*version*)
$\wedge$ T' = receiveVersion
$\vee$ $\exists$ v: *String*
$\wedge$ T = receiveVersion
$\wedge$ *read*(v)
$\wedge$ $\vee$ (v = *version* $\wedge$ T' = receiveTaskId)

$$
\begin{array}{l}
\quad \vee\ (v \neq \mathit{version} \wedge\ S\text{'} = \text{terminate}) \\
\vee\ (\mathit{receive\ task\ identifier}) \\
\quad \wedge\ T = \text{receiveTaskId} \\
\quad \wedge\ \exists\ \mathit{message}\text{: } \mathit{Message} \\
\quad\quad \wedge\ \mathit{receive}(\mathit{message}, 2) \\
\quad\quad \wedge\ \mathit{message}.\text{key} = \text{taskid} \\
\quad\quad \wedge\ \mathit{taskMin}\text{'} = \mathit{message}.\text{body}(\mathit{0}) \\
\quad\quad \wedge\ \mathit{taskMax}\text{'} = \mathit{message}.\text{body}(\mathit{1}) \\
\quad\quad \wedge\ \mathit{taskId}\text{'} = \mathit{taskMin} \\
\quad \wedge\ T\text{'} = \text{null} \\
\quad \wedge\ S\text{'} = C
\end{array}
$$

Send Actions

When a new task is created, the task is assigned the next identifier in the task range.

$$
\begin{array}{l}
\mathit{sendTask}(C) \equiv \\
\quad \exists\ \mathit{message}\text{: } \mathit{Message} \\
\quad\quad \wedge\ \mathit{send}(\mathit{message}, 2) \\
\quad\quad \wedge\ \mathit{message}.\text{key} = \text{task} \\
\quad\quad \wedge\ \mathit{message}.\text{body}[\mathit{0}] = \mathit{taskId} \\
\quad\quad \wedge\ \exists\ \mathit{exp}\text{: } \mathit{Expression} \\
\quad\quad\quad\quad \mathit{message}.\text{body}[1] = \mathit{exp} \\
\quad\quad \wedge\ \mathit{taskId} < \mathit{taskMax} \\
\quad\quad \wedge\ \mathit{taskId}\text{'} = \mathit{taskId}{+}1 \\
\quad\quad S\text{'} = C
\end{array}
$$

When a "wait" request is issued, a message is sent that contains the identifier of the corresponding task.

$$
\begin{array}{l}
\mathit{sendWait}(C) \equiv \\
\quad \wedge\ \exists\ \mathit{message}\text{: } \mathit{Message} \\
\quad\quad \wedge\ \mathit{send}(\mathit{message}, 1) \\
\quad\quad \wedge\ \mathit{message}.\text{key} = \text{wait} \\
\quad\quad \wedge\ \exists\ \mathit{taskid}\text{: } \mathit{TaskId} \\
\quad\quad\quad\quad \mathit{message}.\text{body}[0] = \mathit{taskid} \\
\quad\quad \wedge\ \mathit{waitStack}[\mathit{waitDepth}] = \mathit{message} \\
\quad\quad \wedge\ \mathit{waitDepth}\text{'} = \mathit{waitDepth}{+}1 \\
\quad\quad \wedge\ S\text{'} = C \\
\quad \wedge\ S\text{'} = C
\end{array}
$$

An "nwait" request contains a list of task identifiers; we request the result of *any* of these tasks.

$$
\begin{array}{l}
\mathit{sendNWait}(C) \equiv \\
\quad \wedge\ \exists\ \mathit{message}\text{: } \mathit{Message}, n\text{: } \mathit{Natural} \\
\quad\quad \wedge\ \mathit{send}(\mathit{message}, n{+}1)
\end{array}
$$

$\wedge$ *message*.key = nwait
$\wedge$ *message*.body[0] = n
$\wedge$ $\forall$ i: $1 \leq i \leq n$
$\exists$ *taskid*: *TaskId*
message.body[i] = *taskid*
$\wedge$ *waitStack*[*waitDepth*] = *message*
$\wedge$ *waitDepth*' = *waitDepth*+1
$\wedge$ $S' = C$

An "all" message asks for the evaluation of a command on each application program.

sendAll(C) $\equiv$
$\wedge$ $\exists$ *message*: *Message*
$\wedge$ *send*(*message*, 1)
$\wedge$ *message*.key = all
$\wedge$ $\exists$ *command*: *Command*
message.body[0] = *command*
$\wedge$ $S' = C$

A "quit" message asks for the termination of the session and waits for a corresponding acknowledgement.

sendQuit(C) $\equiv$
$\vee$ (*send quit request*)
$\wedge$ T = null
$\wedge$ $\exists$ *message*: *Message*, *code*: *Integer*
$\wedge$ *send*(*message*, 1)
$\wedge$ *message*.key = quit
$\wedge$ *message*.body[0] = *code*
$\wedge$ T' = receiveAck
$\vee$ (*receive acknowledgement*)
$\wedge$ T = receiveAck
$\wedge$ $\exists$ *message*: *Message*, *code*: *Integer*
$\wedge$ *receive*(*message*, 1)
$\wedge$ *message*.key = quit
$\wedge$ *message*.body[0] = *code*
$\wedge$ T' = null
$\wedge$ $S' = C$

An "abort" message aborts the session immediately.

sendAbort(C) $\equiv$
$\wedge$ $\exists$ *message*: *Message*
$\wedge$ *send*(*message*, 1)
$\wedge$ *message*.key = abort
$\wedge$ $S' = C$

A start message asks for the connection of a new node to the session; we receive either an okay or an error message as an acknowledgement.

```
sendStart(C) ≡
  ∨ (send start request)
      ∧ T = null
      ∧ ∃ message: Message
          ∧ send(message, 2)
          ∧ message.key = start
      ∧ T' = receiveAck
  ∨ (receive acknowledgement)
      ∧ T = receiveAck
      ∧   ∨ (okay)
            ∧ receive(message, 0)
            ∧ message.key = okay
          ∨ (error)
            ∧ receive(message, 1)
            ∧ message.key = error
            ∧ ∃ string: String
                message.body[0] = string
      ∧ T' = null
      ∧ S' = C
```

A delete message asks for the deletion of a task result.

```
sendDelete(C) ≡
  ∧ ∃ message: Message
      ∧ send(message, 1)
      ∧ message.key = delete
      ∧ ∃ taskid: TaskId
          message.body[0] = taskid
  ∧ S' = C
```

A flush message asks for the session caches to be flushed and to switch flushing off (0) or on (1).

```
sendFlush(C) ≡
  ∧ ∃ message: Message
      ∧ send(message, 1)
      ∧ message.key = flush
      ∧   ∨ message.body[0] = 0
          ∨ message.body[0] = 1
  ∧ S' = C
```

A reset message restarts the session resetting the range of task identifiers.

```
sendReset(C) ≡
  ∧ ∃ message: Message
```

$\wedge$ *send*(*message*, 0)
$\wedge$ *message*.key = reset
$\wedge$ *taskId*' = *taskMin*
$\wedge$ *S*' = *C*

A result message contains the result of the task currently under evaluation (stored on top of the stack).

sendResult(C) $\equiv$
$\wedge$ $\exists$ *message*: *Message*
$\wedge$ *send*(*message*, 2)
$\wedge$ *message*.key = result
$\wedge$ *taskDepth* $>$ 0
$\wedge$ $\exists$ *taskid*: *TaskId*, *exp*: *Expression*
$\wedge$ *taskStack*[*taskDepth*$-$1].body[0] = *taskid*
$\wedge$ *message*.body[0] = *taskid*
$\wedge$ *message*.body[1] = *exp*
$\wedge$ *taskDepth*' = *taskDepth*$-$1
$\wedge$ *S*' = *C*

Receive Actions

When the program receives a task for evaluation, it pushes this task on the stack (its result must be delivered before the result of any of the other task on the stack).

receiveTask(C) $\equiv$
$\wedge$ $\exists$ *message*: *Message*
$\wedge$ *receive*(*message*, 2)
$\wedge$ *message*.key = task
$\wedge$ *taskStack*[*taskDepth*] = *message*
$\wedge$ $\exists$ *taskid*: *TaskId*, *exp*: *Expression*
message.body[0] = *taskid*
message.body[1] = *exp*
$\wedge$ *taskDepth*' = *taskDepth*+1
$\wedge$ *S*' = *C*

When the program receives an all message, this message contains a command to be executed.

receiveAll(C) $\equiv$
$\wedge$ $\exists$ *message*: *Message*
$\wedge$ *receive*(*message*, 1)
$\wedge$ *message*.key = all
$\wedge$ $\exists$ *c*: *Command*
message.body[0] = *c*
$\wedge$ *S*' = *C*

When the program receives a quit message, it terminates execution with a result code.

$$
\begin{array}{l}
\mathit{receiveQuit}(C) \equiv \\
\quad \wedge\ \exists\ \mathit{message}\colon \mathit{Message} \\
\qquad \wedge\ \mathit{receive}(\mathit{message}, 1) \\
\qquad \wedge\ \mathit{message}.\mathrm{key} = \mathrm{quit} \\
\qquad \wedge\ \exists\ i\colon \mathit{Integer} \\
\qquad\qquad \mathit{message}.\mathrm{body}[0] = i \\
\quad \wedge\ S' = C
\end{array}
$$

When the program receives an abort message, it terminates execution with an error message.

$$
\begin{array}{l}
\mathit{receiveAbort}(C) \equiv \\
\quad \wedge\ \exists\ \mathit{message}\colon \mathit{Message} \\
\qquad \wedge\ \mathit{receive}(\mathit{message}, 1) \\
\qquad \wedge\ \mathit{message}.\mathrm{key} = \mathrm{abort} \\
\qquad \wedge\ \exists\ \mathit{string}\colon \mathit{String} \\
\qquad\qquad \mathit{message}.\mathrm{body}[0] = \mathit{string} \\
\quad \wedge\ S' = C
\end{array}
$$

When the program receives the result corresponding to the wait/nwait message on the top of the wait stack, it can continue the suspended execution.

$$
\begin{array}{l}
\mathit{receiveResult}(C) \equiv \\
\quad \wedge\ \exists\ \mathit{message}\colon \mathit{Message} \\
\qquad \wedge\ \mathit{receive}(\mathit{message}, 2) \\
\qquad \wedge\ \mathit{message}.\mathrm{key} = \mathrm{result} \\
\qquad \wedge\ \mathit{waitDepth} > 0 \\
\qquad \wedge\ \exists\ \mathit{taskid}\colon \mathit{TaskId},\ \mathit{exp}\colon \mathit{Expression} \\
\qquad\quad \wedge\ \mathit{message}.\mathrm{body}[0] = \mathit{taskid} \\
\qquad\quad \wedge\ \mathit{message}.\mathrm{body}[1] = \mathit{exp} \\
\qquad\quad \wedge\ \exists\ i\colon \mathit{Integer} \\
\qquad\qquad\quad \mathit{waitStack}[\mathit{waitDepth}{-}1].\mathrm{body}[i] = \mathit{taskid} \\
\qquad \wedge\ \mathit{waitDepth}' = \mathit{waitDepth}{-}1 \\
\quad \wedge\ S' = C
\end{array}
$$