A Distributed Computer Algebra System Based on Maple and Java[1]

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria

Abstract

Distributed Maple is a system for writing parallel programs in the computer algebra system Maple. It allows to create concurrent tasks and to execute them 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 is freely available.

Keywords: distributed systems, computer algebra, Maple, Java.

1 Introduction

This paper 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 [2]. In particular, the Maple library CASA of algorithms for solving problems in algebraic geometry has been developed in our institute [4]. 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 [5]; data communication was

[1] Supported by the Austrian Science Foundation FWF grant P11160-TEC "HySaX".

realized by a simple mechanism, the reading and writing of shared files in a global network file system.

- On the basis of the parallel language Strand88, Siegl developed ||Maple|| (read: "Parallel Maple") [3]. Strand coordinated 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 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 R
 \  MAPLE  /  reserved. Maple and Maple V are registered trademark
```

```
 <____ ____>  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:

```
> 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:

# ---------------------------------------------------
# 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.

Distributed Maple also allows 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.

3 Implementation

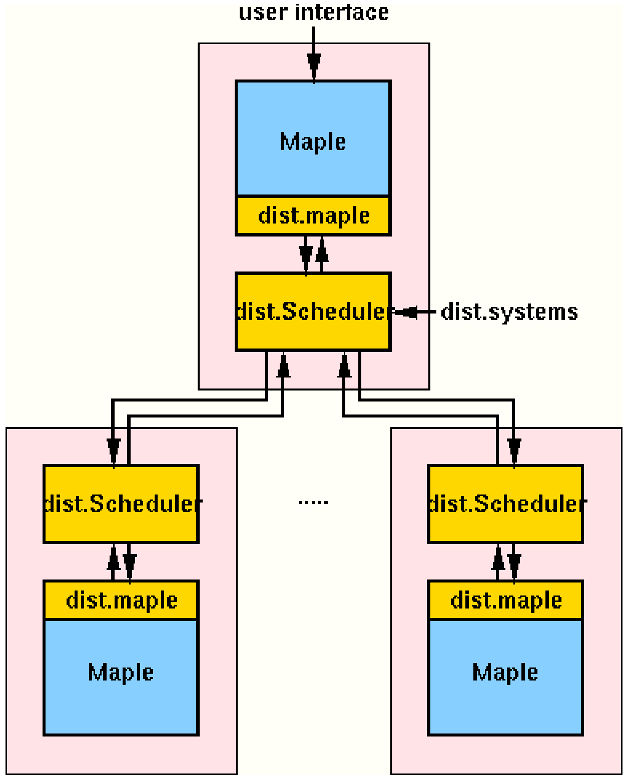
The architecture of a distributed Maple session is sketched on the next page. Each node connected to the session consists of two components:

Scheduler

This program manages the node interaction and schedules tasks among nodes. It 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

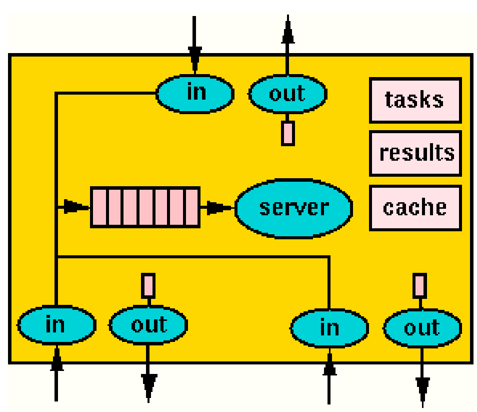
The package *dist.maple* running on each Maple kernel implements the interface between Maple and the scheduler. Communication between both components is based on pipes (named pipes for the Maple kernel connected to the user interface and standard input/output streams for the backend kernels).



A Distributed Maple Session

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.



Scheduler Architecture

The following paragraphs sketch the operation of the scheduler

Multi-Threading

The scheduler is implemented by a number of concurrent threads as shown in above figure. 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 completed tasks reaches an upper bound (specific to the node class) and a remote scheduler asks for new tasks whenever the number of not started tasks falls below a lower bound.

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.

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

[3] 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.

[4] 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.

[5] 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.