Distributed *Mathematica* – User and Reference Manual

Cleopatra Pau
Wolfgang Schreiner

July 5, 2000

Abstract

Distributed *Mathematica* is a system for writing parallel programs in the integrated system for technical computing *Mathematica*. It allows to create concurrent tasks and have them executed by *Mathematica* 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 *Mathematica* and implements a parallel programming model on top.

The system can be freely downloaded from:

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

Contents

1 Introduction

This report and the package described in it are the result of our work in the "Parallel and Distributed Software and Algorithms" seminar held at RISC-Linz during the summer semester 1999 and the winter semester 1999/2000.

The report describes the use of the Distributed *Mathematica* package, that is a system for writing distributed applications in *Mathematica* [2]. *Mathematica* is a very popular fully integrated environment for technical computing.

Our goal was to realize a system similar to Distributed Maple [1], so our package provides an environment that makes it easy to implement parallel algorithms in *Mathematica* and that can be easily installed in any environment in order to facilitate the distribution of these implementations.

Distributed *Mathematica* is not the only approach to parallel programming with *Mathematica*. Wolfram Research developed a commercial package *Parallel Computing Toolkit* [3], based on MathLink for Interkernel communication with support for master/slave parallelism and other programming models. This package is written entirely in the *Mathematica* programming language.

Our package contains a *Mathematica* interface, that allows the user to write parallel programs in the environment provided by *Mathematica*, and that communicates with a scheduling program which may be used to start and connect external application programs on different machines. This latter program is also a component of Distributed Maple, is written in Java and can be executed on any machine running some implementation of the Java Virtual Machine. The user interacts with Distributed *Mathematica* via the *Mathematica* frontend which communicates with the scheduling program running in the background in order to transfer tasks to other Mathematica kernels running on various machines (see Figure 1).

Both the Java scheduler (compiled Java classes) and the *Mathematica* interface (*Mathematica* expression file in plain text format) are freely available under the GNU Library Public License at:

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

The fact that both Distributed *Mathematica* and Distributed Maple use the same Java scheduler makes it possible in the future to integrate these two systems, such that from *Mathematica* the user will be able to start Maple processes and vice-versa.

2 Examples

2.1 A Simple Example

The below excerpt from a *Mathematica* notebook shows the use of the system, by a simple example:

```
In[1]:=<< dist.m
       Distributed Mathematica V1.0.0
         (c) 2000 Cleopatra Pau (RISC-Linz)
       See http://www.risc.uni-linz.ac.at/software/distmath
In[2]:=InitializeD[{{deneb,solaris},{milkyway,octane}}]
       Connecting deneb...
       Connecting milkyway...
Out[2]=okay
In[3]:=t1=StartD["Integrate[x^n,x]"]
```

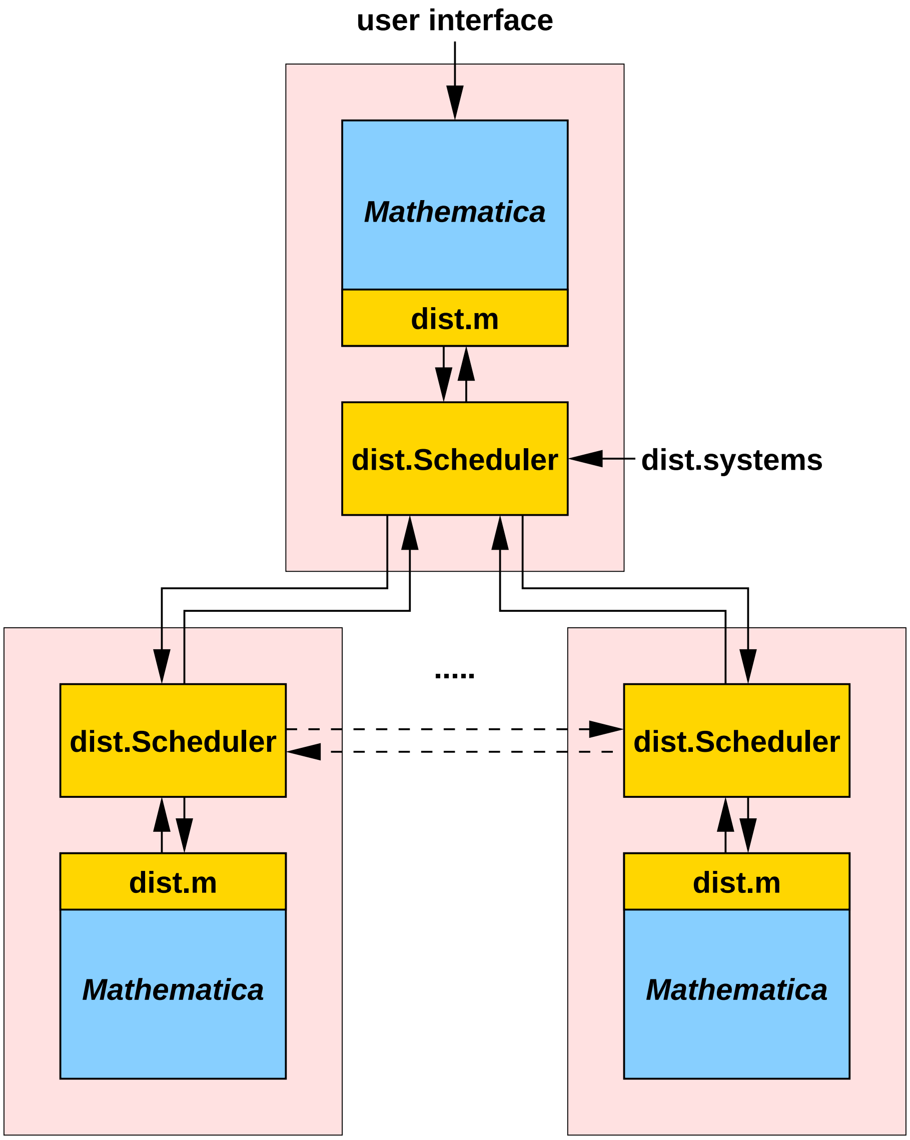

Figure 1: The architecture of a distributed *Mathematica* session

```
Out[3]=0
In[4]:=t2=StartD["Integrate[x^n,n]"]
Out[4]=1
In[5]:=WaitD[t1]+WaitD[t2]
       x^(1 + n)    x^n
Out[5]=--------- + ------
         1 + n     Log[x]
In[6]:=TerminateD[]
Out[6]=okay
```

In this example, we first load the package `dist.m`, which prints a copyright message. Then the distributed session is initialized, by the call of `InitializeD` with a list argument that tells the system to connect to the local kernel other two kernels, one on the machine `deneb` of type `solaris`, the other on the machine `milkyway` of type `octane`. After the connection is established successfully, we create two tasks, by using the `StartD` function. The first task evaluates `Integrate`$[x^n, x]$, and it gets the id 0, while the second task evaluates `Integrate`$[x^n, n]$, and it gets the id 1. The next input calls `WaitD`, once for each task. For each call, the execution of the current kernel is blocked, until it gets the corresponding result. After we get the result, we close the distributed system, by the call of `TerminateD`.

2.2 Distributed Factorization of Integers

The following program uses the features of Distributed *Mathematica* for computing the factorization of a list of cubes of integers.

```
main[length_,size_]:=Module[
  {numbers,results,ti},

  (* load to all machines the Cube function *)
  AllD["<< test.m"];

  (* generate list of numbers *)
  numbers=rands[length,size];

  (* compute result *)
  results=dfactors[numbers];

  (* return list of argument/result pairs *)
  Table[{numbers[[i]],results[[i]]},{i,length}]
]

dfactors[numb_]:= Block[
  {length,tasks,results, i,t,r,numbers=numb},

  (* length of list *)
  length=Length[numbers];

  (* generate list of tasks *)
  tasks={};
  For[i=1,i<=length,i++,
    t=StartD["FactorInteger[Cube[numbers[[i]]]]"];
    tasks=Append[tasks,t];
  ];

  (* wait for result values *)
  results={};
  For[i=1,i<=length,i++,
    r=WaitD[tasks[[i]]];
```

```
    results=Append[results,r];
  ];

  (* return results *)
  results
]
rands[length_,size_]:=Module[
  {i,n,numbers},

  (* generate list *)
  numbers={};
  For[i=1,i<=length,i++,
    n=Random[Integer,size];
    numbers=Append[numbers,n];
  ];

  (* return list *)
  numbers
]
```

The `main` function calls `AllD`, which executes the command given as argument on all connected kernels, and also on the local one. The executed command loads a small program, that contains the definition of the function `Cube`.

The function `dfactors` generates a list of tasks, each of it factorizing the cube of an integer number. The tasks are distributed to the connected kernels, and the results are collected in a list. The function `dfactors` is defined into a `Block`, and the argument is copied into a local variable to allow the evaluation of the arguments of `FactorInteger` by the local kernel. The tasks are executed in parallel, using a common input structure, and the results are gathered into a common output structure.

We can use the program in the context of the following *Mathematica* session:

```
In[1]:=<< dist.m
       Distributed Mathematica V1.0.0
         (c) 2000 Cleopatra Pau (RISC-Linz)
       See http://www.risc.uni-linz.ac.at/software/distmath
In[2]:=InitializeD[{{deneb,solaris},
                    {andromeda,octane},
                    {pinwheel,octane}}]
       Connecting deneb...
       Connecting andromeda...
       Connecting pinwheel...
Out[2]=okay
In[5]:=<< dfactor2.m
In[6]:=VisualizeD[600, 300, 2, 120]
In[7]:=main[20,2^25]
Out[7]={{26496159,{{3,3},{8832053,3}}}, ...}
In[8]:=TerminateD[]
Out[8]=okay
```

The call of `VisualizeD` opens a window, where one can see the status of each machine and the number of active machines in the last 120 seconds, image refreshed every two seconds.

3 Programming interface

A Distributed *Mathematica* session can be initiated either from a notebook interface, or from a text-based interface. In each case, first the package `dist.m` should be loaded as in the example bellow:

```
In[1]:=<< dist.m
       "Distributed Mathematica V1.0.0"
         (c) 2000 Cleopatra Pau (RISC-Linz)
       See http://www.risc.uni-linz.ac.at/software/distmath
```

In the following subsections we give, for each command, the syntax, a short description and a small example.

3.1 Session Setup

InitializeD: Initialize a Session

$$\texttt{InitializeD}[\textit{machinelist}]$$

Each element from *machinelist* is a pair $\{machine, type\}$, where *machine* represents the domain name or the numeric Internet address of a machine, and *type* is the corresponding key in the file *dist.system*, which gives the system informations about the way the machine can be connected. The system tries to connect and start a *Mathematica* kernel on each machine from the list. If a connection fails, it continues with the others. If the list is empty, then only the local kernel will be used. If there already is an active session, this call will first close it, and then it will start a new one.

```
In[2]:=InitializeD[{{galaxy.risc.uni-linz.ac.at,solaris},
                    {"193.170.37.165",octane},
                    {zeus,linux}}]
       Connecting galaxy.risc.uni-linz.ac.at...
       Connecting 193.170.37.165...
       Connecting zeus...
Out[2]=okay
```

TerminateD: Terminate a Session

$$\texttt{TerminateD}[]$$

This call terminates a distributed session, by closing all the kernels except the local one, and all the communications. If there is no distributed session started, it prints an acknowledgment message. `TerminateD` should be called before quitting *Mathematica*.

```
In[6]:=TerminateD[]
Out[6]=okay
```

3.2 Basic Commands

StartD: Create a Task

$$task = \texttt{StartD}[\text{“}command''\text{]}$$

This form of the call creates a new task, that evaluates the outermost function of *command* on some *Mathematica* kernel connected to the distributed session. When this form is called inside a function f, the user should use `Block`, and not `Module` to define f, in order to allow the evaluation of the arguments of *command*.

```
In[3]:=t=StartD["Integrate[x^n, x]"]
Out[3]=0
```

Another way of using `StartD` is:

$$task = \texttt{StartD}[name, args \ldots]$$

This call creates a new task, that executes $name[args \ldots]$ on some *Mathematica* kernel connected to the distributed session. The arguments are evaluated on the local kernel. Both forms of `StartD` return a handle of the task.

```
In[3]:=t=StartD[Integrate,x^n,x]
Out[3]=0
```

`WaitD`: Wait for the Result of a Task

$$result = \texttt{WaitD}[task]$$

This call blocks the current execution until the result of *task* is available and returns the result. As long as the execution is blocked, the kernel can execute other tasks assigned by the scheduler.

```
In[3]:=t=StartD[Integrate,x^n,x];
In[4]:=r=1+WaitD[t]
           x^(1 + n)
Out[4]=1 + ---------
             1 + n
```

`SelectD`: Wait for Some Result from a List of Tasks

$$selector = \texttt{SelectD}[tasklist]$$

This call blocks the current execution, until some task from *tasklist* is finished and its result is available. It returns a *selector*, i.e., a list $\{index,\ result\}$, where *index* is the position of the task in *tasklist*, and *result* is the corresponding result.

```
In[3]:=l={StartD["Integrate[x^n,x]"],StartD["Integrate[x^n,n]"]};
In[4]:=s=SelectD[l];
In[5]:=t=s[[1]]
Out[5]=1
In[6]:=r=s[[2]]
       x^(1 + n)
Out[6]=---------
         1 + n
```

`AllD`: Execute a Command on All Kernels

$$\texttt{AllD}[\text{``}command''] $$

This call executes *command* on each kernel connected to the distributed session, including the local kernel.

```
In[6]:=AllD["<< /home/ipau/distsoftware/distmath/test.m"]
Out[6]=okay
```

3.3 Shared Data

`DataD`: Create a Shared Object

$$\texttt{DataD}[]$$

This call creates a shared object that can be used to communicate data between different tasks, and returns a handle for it. The object is initially empty.

```
In[3]:=d=DataD[]
Out[3]=0
```

PutD: Write Data into a Shared Object

$$\texttt{PutD}[d, v]$$

This call assigns the value v to the shared data object d and releases all the tasks blocked by the attempt to get the value of this object.

```
In[4]:=PutD[d,x+y]
Out[4]=okay
```

GetD: Get Data from a Shared Object

$$\texttt{GetD}[d]$$

This call returns the value of the shared data object d. If d is empty, the task is blocked until a value is put into d.

```
In[5]:=v=GetD[d]
Out[5]=x+y
```

ClearD: Clear a Shared Object

$$\texttt{ClearD}[d]$$

This call resets the shared data object d to the empty state. Further attempts of getting the value of d will block the corresponding tasks.

```
In[6]:=ClearD[d]
Out[6]=okay
```

ProcessD: Execute a Task by a Separate Process

$$task = \texttt{ProcessD}[\text{“}command''\,]$$

This call evaluates the outermost function of *command* on a separate (possibly newly created) *Mathematica* process, after the rest of the functions contained in *command* have been evaluated on the local kernel. All the previous calls of `AllD` have been executed, in the corresponding order, by this separate process. A handle for the task is returned. During the execution of the task, no other task is scheduled on this process.

When `ProcessD` is called inside a `Module`, the arguments and local variables cannot be passed for the evaluation of *command*'s arguments, and therefore in these situations, `ProcessD` should be called inside a `Block`.

The execution of `ProcessD` may involve a much larger runtime overhead than the execution of `StartD`.

```
In[3]:=t=ProcessD["Integrate[x^n, x]"]
Out[3]=0
```

As for `StartD`, there is another form for `ProcessD`:

$$task = \texttt{ProcessD}[name, args \ldots]$$

This call creates a task as the previous form, that executes $name[args \ldots]$ on a separate process.

```
In[3]:=t=ProcessD[Integrate,x^n,x]
Out[3]=0
```

3.4 Global variables

Global variables can be created in two ways:

- by using the `AllD` command;
- by using the features of Distributed *Mathematica*. These features allow the user to create global variables whose names cannot create conflicts with the names of the local variables, to set and get their values. There is a default global variable, named "proc", that stores the number of machines involved in the distributed system.

The commands for manipulating global variables are:

`SetParamD`: Set the Value of a Global Variable

`SetParamD`["*name*", *value*]

This call assigns *value* to the global variable *name*. If this variable does not exist, it will be created. After this call is executed, all the processes will have access to the value of the variable.

```
In[3]:=SetParamD["n",10]
```

`GetParamD`: Get the Value of a Global Variable

`GetParamD`["*name*"]

This call returns the value of the global variable *name*. If this variable does not exist, an error will be generated.

```
In[4]:=GetParamD["n"]
Out[4]=10
In[5]:=GetParamD["proc"]
Out[5]=4
```

`HasParamD`: Test the Value of a Global Variable

`HasParamD`["*name*"]

This call tests whether the global variable *name* exists or not and returns `True`, respectively `False`.

```
In[6]:=HasParamD["n"]
Out[6]=True
In[7]:=HasParamD["m"]
Out[7]=False
```

3.5 Miscellaneous

`DeleteD`: Delete a Task

$$\texttt{DeleteD}[\mathit{task}]$$

This call deletes *task* from the current distributed session. After the task is deleted, it should not be referenced anymore.

If the task is finished, the result will be deleted from the result caches. If the task is still executed, then the system waits until the result is computed, but afterwards the result will be deleted. If the execution of the task has not yet started, then it will be canceled before start.

```
In[5]:=DeleteD[t]
Out[5]=okay
```

`FlushD`: Flush the Result Caches

$$\texttt{FlushD}[\mathit{code}]$$

This call flushes the result caches. If *code* is 0, then the caching is switched off, if it is 1, then the caching is kept on. Any other value will generate an error.

```
In[5]:=FlushD[0]
Out[5]=okay
```

`ResetD`: Reset the Session

$$\texttt{ResetD}[]$$

This call resets the current distributed session, i.e., it deletes all the tasks and their results, flushes the result caches, destroys and restarts all the *Mathematica* kernels connected to the distributed session, except the kernel that started the session. The network connections remain open, therefore reseting the session is faster than terminating the session and restarting it.

```
In[5]:=ResetD[]
Out[5]=okay
```

`AbortD`: Abort the Session

$$\texttt{AbortD}[\mathit{message}]$$

This call aborts the current distributed session, by closing all the connected kernels except the one that started the session, and closing the corresponding network connections. It returns *message*.

```
In[11]:=AbortD["invalid type"]
Out[11]=invalid type
```

3.6 Visualization

The local host can generate informations about the dynamic behaviour of the distributed system. These informations are inferred from messages that are received from the other machines, and therefore they might be delayed. There are two possibilities for the visualization of these informations:

1. *Online Visualization*: It opens a window of desired size, which displays the execution status of all connected machines for the last period of time. In the upper half of the window, it is shown for each machine whether it is active or not (a horizontal line indicates that the corresponding machine is executing a task). In the lower half of the window, a diagram is shown that indicates how many machines are active at each moment (see Figure 2).

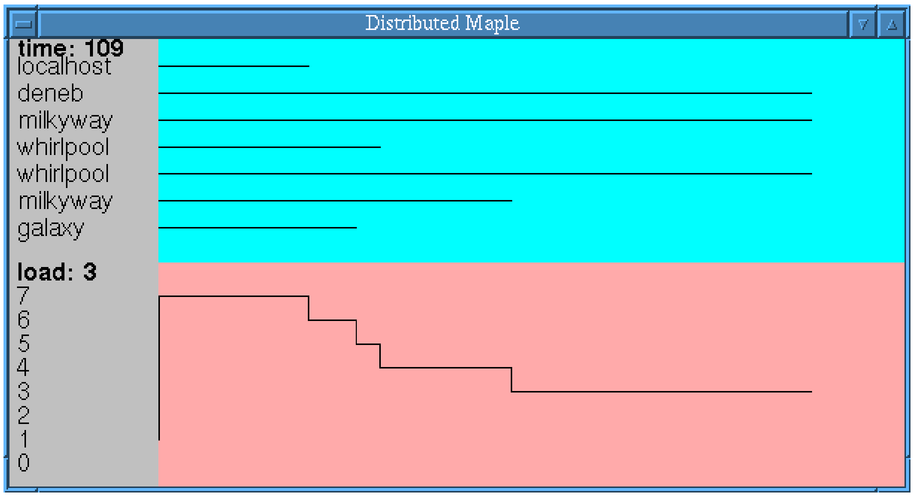

Figure 2: Online visualization

2. *Post-Execution Visualization*: During a distributed session, a tracefile with the relevant events can be generated. After the session is ended, the following call from the command line:

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

   will generate some files, each of them containing a diagram of dimension *width* × *height* millimeters that visualizes some aspects of the program's runtime behavior, for the interval between the labeled moments *start* and *end*. The generated files are the following:

   - *filename−machines.fig* (Figure 3) 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* (Figure 4) displays the number of active machines at a time;

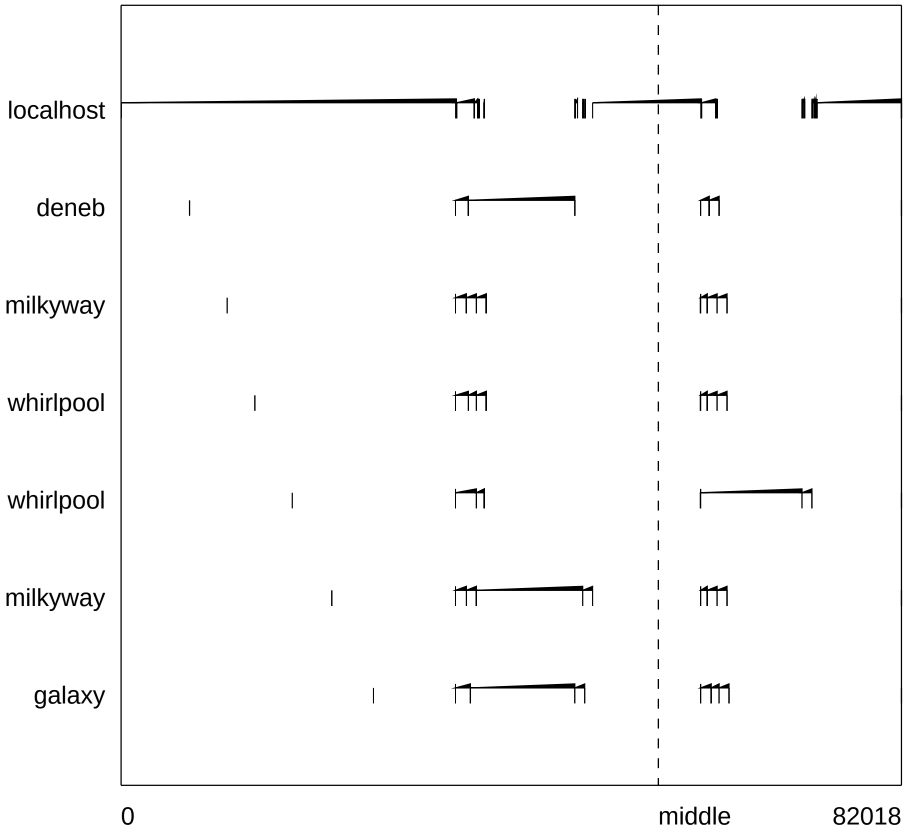

Figure 3: Post-Execution Visualization: *filename-machines.fig*

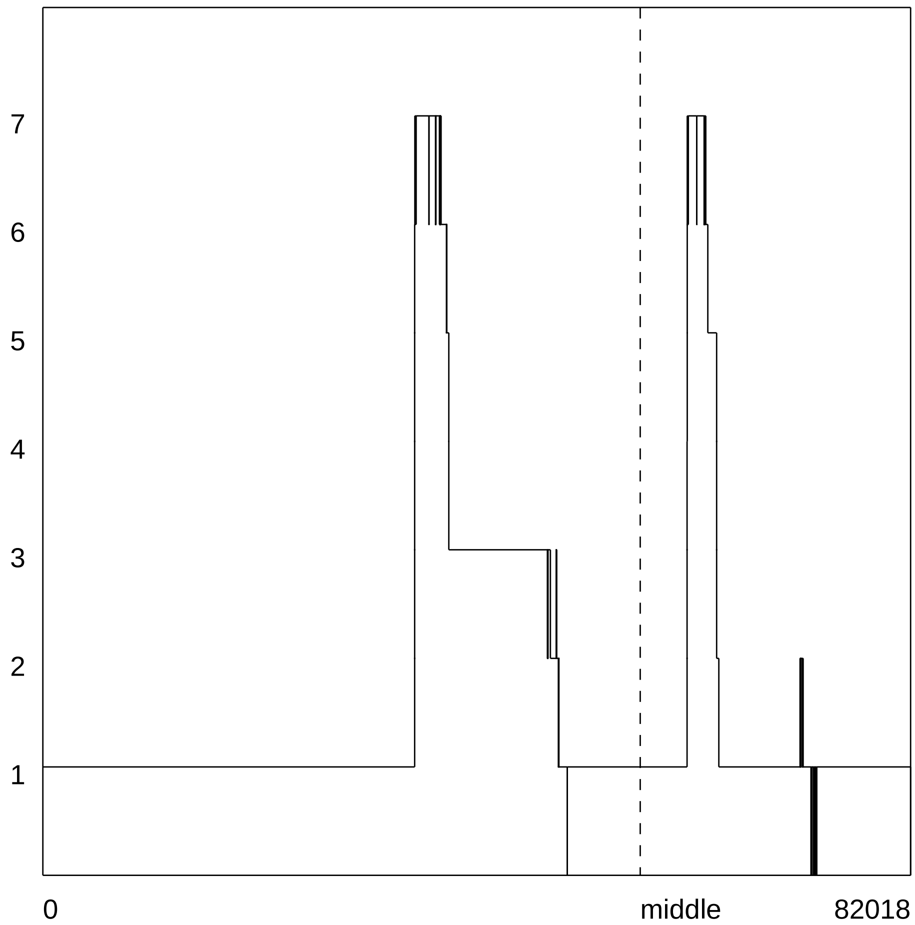

Figure 4: Post-Execution Visualization: *filename-util.fig*

- $filename - tasks.fig$ (Figure 5) displays the interval of time during which each task was executed, and by which machine;

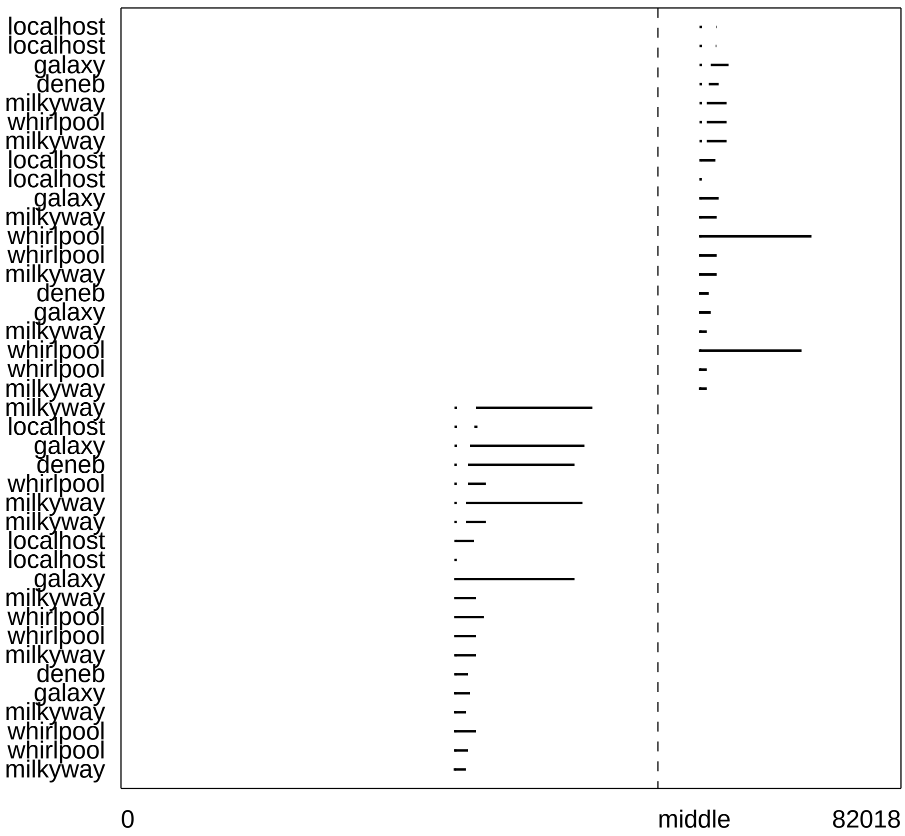

Figure 5: Post-Execution Visualization: *filename-tasks.fig*

The numbers at the bottom of each diagram denote the range of the visualization time in milliseconds. The vertical dashed lines represents execution labels. There are two predefined labels, *initialize* and *terminate*, but the user can define at any time of the execution other labels. These labels can be used for restricting the visualization range.

Commands for visualization:

`VisualizeD`: Create a Visualization Window

$$\texttt{VisualizeD}[width, height, updates, range]$$

This call creates a visualization window of size $width \times height$, that displays the last *range* seconds of the program execution, with samples taken each *updates* seconds. In this window, it is shown for each machine whether it executes a task or not, and the number of active machines at each moment.

The command may be called more than once with different parameters yielding windows that may be individually resized and closed. For closing a window, use the "Close" button in the window frame. Using the "Exit" button will cause the termination of the scheduler.

```
In[6]:=VisualizeD[600, 300, 2, 120]
Out[6]=okay
```

`TraceD`: Create a Trace File

`TraceD`[*filename*]

This call creates a trace file into which the relevant events are logged. The call must be issued before `InitializeD` is executed.

```
In[2]:=TraceD["tracefile"]
```

From the *filename* file, some graphics can be generated after `TerminateD` is executed, by using a call from the command line:

```
pinwheel:29> java visual.Main tracefile initialize terminate
             100 100
processing tracefile...
creating tracefile-machines.fig...
creating tracefile-util.fig...
creating tracefile-tasks.fig...
done.
```

`LabelD`: Create a Label

`LabelD`[*labelname*]

This call creates a label called *labelname* that may be used in the visualization of a trace file. Each call of `LabelD` must use a different value for *labelname*.

```
In[6]:=LabelD["middle"]
Out[6]=okay
```

4 System Configuration

When `InitializeD[...]` is called, the *Mathematica* kernel starts the Java scheduler `dist.Scheduler` and establishes a permanent connection to this process. The scheduler starts Java schedulers on the requested machines, each of them starting a *Mathematica* kernel. The information about how to do these on each machine is read by the scheduler from the file *dist.systems* which contains a list of entries of the following form, separated by empty lines:

```
key
starter
scheduler
command

speed
minload
maxload
```

The fields have the following interpretation:

- **key** is the 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 that creates an instance of the scheduler on the denoted machine:

  ```
  java dist.Scheduler
  ```

  with some `-classpath` option.
- **command** is the command started by the scheduler on the denoted machine in order to run a Mathematica kernel:

  ```
  math -runfirst 'dist`isClient=True'
       -initfile "~/ipau/distsoftware/distmath/dist.m"
       -batchoutput
  ```
- an **empty line** is mandatory between *command* and *speed*.
- **speed** is a performance factor – an integer number greater than or equal to 1. The higher the number is, the faster this machine is expected to be.
- **minload** sets the maximum number of executable but not yet started tasks that the denoted machine may hold at a time. If the scheduler on the frontend machine knows that the number is below this threshold, it may forward newly created tasks to the denoted machine. If maxload is -1, the scheduler may always forward newly created tasks to that machine (which may result in poor load balancing).
- **maxload** sets the minimum number of executable but not yet started tasks that the denoted machine should hold at a time. If the number falls below this threshold, this machine informs the scheduler on the frontend machine about it (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).

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.

Below there are some sample entries of *dist.systems*:

```
localhost
dummy
dummy
math -runfirst 'dist`isClient=True' -initfile "/home/ipau/
 distsoftware/distmath/dist.m" -batchoutput

14
1
0

octane
rsh
java -native -classpath /usr/java/lib/dev.jar:/usr/java/lib/i18n.jar:
```

```
 /usr/java/lib/rt.jar:/home/ipau/distsoftware/distmath dist.Scheduler
math -runfirst Set[dist`isClient,True] -initfile "/home/ipau/
 distsoftware/distmath/dist.m" -batchoutput

14
1
0
```

References

[1] Wolfgang Schreiner. *Distributed Maple - User and Reference Manual.* Technical Report 98-35, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, 1997. ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1998/98-05.ps.gz.

[2] Wolfram Research. *Mathematica 4*, June 1999. http://www.wri.com/.

[3] Wolfram Research. *Parallel Computing Toolkit*, February 2000. http://www.wri.com/news/pct.html.