User's Guide
to Macro Parallelism
in MuPAD 1.4.1

By Torsten Metzner, Manfred Radimersky,
Andreas Sorgatz and Stefan Wehmeier
Universität-Gesamthochschule Paderborn (Germany)
The Authors:
Torsten Metzner, Manfred Radimersky, Andreas Sorgatz and Stefan Wehmeier are members of the MuPAD research group at the Department of Mathematics and Computer Science of the University of Paderborn. They are working on the design and implementation of computer algebra systems and libraries.

Contact via email {tom, maradim, andi, stefanw}@mupad.de or bugs@mupad.de for questions, problem reports or suggestions for improvement as well as to provide your contributions.

The authors accept no responsibility for this manual or the programs described in it. The authors of this manual and the developers of MuPAD and contributions available on the accompanying CD-ROM assume no responsibility for direct or indirect damages, for losses, costs, claims for loss of profits or charges and expenses of any kind.


This work was supported by DFG Sonderforschungsbereich 376: Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen.
Preface of the Editor

Mathematics is the basis of technological progress and technological progress is a key for international competitiveness. Automating an important part of the mathematical problem solving process is a key technology for a nation that wishes to control structure and accelerate technological progress. The automation of the solution of mathematical problems is a powerful lever with which human productivity and expertise can be amplified many times. –


Computer algebra systems (CAS) provide engineers and scientists with almost all mathematics they need in their daily work: these systems intelligently manipulate formulas, determine derivatives, solve equations, draw graphs, visualize geometry, and compute results with arbitrary precision. Furthermore they offer high level languages for rapid prototyping of sophisticated problems. For coming generations computer algebra will considerable change their attitude towards mathematics and its applications in science.

Alas, the power of these systems leads many to apply computer algebra as a black box tool without taking into critical account neither the weaknesses nor the strong points of the underlying technology. However, efficient and creative use of that technology can only be based on considerable knowledge of internal structures and the basics of the internal design of these systems, only if such information is taken into account, CAS can develop its full power. Since most system designers are still reluctant to offer inside into these details, the present series of MuPAD Reports aims at making such information widely available.

Paderborn im October 1998

Benno Fuchssteiner
Interactive general purpose computer algebra systems (CAS) gain more and more relevancy in mathematics, education and engineering but also in related science. They are developed from monolithic systems to more open integrated working environments providing object oriented programming languages (see Domains in MuPAD) to support polymorphic programming and to ease the construction of mathematical data structures. They offer flexible and efficient concepts of software integration (see Dynamic Modules in MuPAD) enabling users to integrate external special purpose tools and algorithms (which may be written in C/C++) as native elements of the CAS programming language. Thus users can extend the CAS and adapt it according to their need.

Another aspect of current research concerning the design and implementation of CAS is parallelism. Parallelism allows new approaches in writing mathematical algorithms, e.g. executing a heuristic and an exact algorithm in parallel, hoping the first one delivers the result in a very short time but also knowing the second (slower) one cannot fail.

Parallelism in general speeds up execution times and, when implemented on a computer network, allows users to utilize all distributed resources of this network, e.g. first and second level memory, processing time, etc. Since heterogeneous computer networks are a standard part of modern working environments, supporting parallel programming on them is also a low-prized solution for high-performance symbolic computations - the network is the computer.

This manual describes Macro Parallelism in MuPAD. This is based on the concepts of message passing, global variables and work groups and enables users to implement parallel algorithms for distributed computations in a heterogeneous computer network in a very easy and convenient way.

In combination with the concept of Domains and Dynamic Modules, Macro Parallelism is the third pile for using MuPAD as an open parallel problem solving environment for mathematical applications.

Paderborn in October 1998

Andreas Sorgatz
Contents

1 Introduction 1
  1.1 The Concept in Brief .......................... 1
  1.2 Quick Start .................................. 3
  1.3 Where to Find Further Information ............... 4
  1.4 Challenges and Contributions .................... 4

2 Programming Interface 7
  2.1 Overview and Tutorial ......................... 7
    2.1.1 Network Configuration *(Startup and Shutdown)* .... 7
    2.1.2 Network Topology Information ................. 9
    2.1.3 Message Passing *(Queues and Pipes)* .......... 9
    2.1.4 Network Variables *(Global Variables)* ........ 10
    2.1.5 Locks and Semaphores ....................... 11
    2.1.6 Automatic Scheduling Primitives *(Work Groups)* ... 11
    2.1.7 Miscellaneous Functions ..................... 12
  2.2 Reference Manual ................................ 13

3 Examples of Parallel Algorithms 49
  3.1 A very simple example: search parallelism .......... 50
  3.2 Factoring Integers Using Elliptic Curves .......... 52
  3.3 Factoring Integers by Means of a Quadratic Sieve ... 54
  3.4 Gcd over Algebraic Number Fields .................. 57
  3.5 Computing Fibonacci Numbers in Parallel .......... 59
  3.6 Karatsuba’s Method for Multiplying Polynomials .... 60
4 Programming Guidelines 63
  4.1 Some Efficiency Considerations 63
  4.2 Writing Good MuPAD Code 65
  4.3 Illegal Functions and Commands 65
  4.4 Tips and Tricks 66

5 Additional Aspects 69
  5.1 Debugging Parallel Algorithms 69
  5.2 Profiling Parallel Algorithms 69
  5.3 Software Integration 70
  5.4 A Math Problem Solving Environment 71

6 Trouble Shooting 73
  6.1 Common Errors in Writing Algorithms 73
  6.2 Problems Concerning Module net 74
  6.3 Problems Concerning PVM 75
  6.4 Other Technical Problems 76
  6.5 Questions and Answers 77

7 Design and Implementation 81
  7.1 How does it Work Inside 83
  7.2 Concerning Global Variables 84
  7.3 Concerning Locks and Semaphores 84
  7.4 Concerning Work Groups 85
  7.5 Concerning the PVM Library 85
  7.6 Technical Information and Discussion 86
    7.6.1 Technical Concepts and Details 86
    7.6.2 About Module Source Code and Development 92
    7.6.3 Open Problems and Missing Features 95
    7.6.4 Desired Features and Future Extensions 97
8 Mathematical Benchmarks
  8.1 Gcd over Algebraic Number Fields ............................... 99
  8.2 Karatsuba’s Method for Multiplying Polynomials ............... 100
  8.3 Factoring Integers by the Elliptic Curve Method ............. 101
  8.4 Factoring Integers by Means of a Quadratic Sieve .......... 101

A The Accompanying CD-ROM ........................................ 105
  A.1 System Requirements ........................................... 105
  A.2 Using the CD-ROM Live System ................................ 106
    A.2.1 Copying on Hard Disk ................................... 107
  A.3 MuPAD License Agreements ..................................... 107
    A.3.1 General License .......................................... 108
    A.3.2 Educational License ...................................... 110
    A.3.3 About PVM ................................................ 111
    A.3.4 How to Register MuPAD .................................. 111
    A.3.5 MuPAD Manuals ............................................ 112

B The Net Package .................................................. 113
  B.1 Copying the Net Package ...................................... 113
  B.2 Installing the Net Package ................................... 113
  B.3 Configuring the net Module ................................... 114
  B.4 Special Tools .................................................. 115

C Changes ..................................................................... 119
  C.1 With Respect to the ALPHA Release ......................... 119
  C.2 With Respect to Release 1.4 ................................. 119
  C.3 With Respect to Release 1.3 ................................. 120

D Glossary .................................................................. 121
List of Figures

1.1 Simplified Scheme of Macro Parallelism ................. 1
1.2 Concepts of MuPAD Macro Parallelism .................... 2
1.3 The Concept of Work Groups .......................... 3
1.4 A First Example on Using Macro Parallelism in MuPAD .... 4

2.1 Configuration of a Macro Parallel Network ................ 8
2.2 Using Topology Information Functions .................... 9
2.3 Using Message Passing Functions ........................ 10
2.4 Using Global Variables ................................ 10
2.5 Using Locks and Semaphores ............................. 11
2.6 Using Work Groups ................................... 12
2.7 Miscellaneous Functions of Macro Parallelism .............. 13

3.1 A Parallel Search Method ................................ 51
3.2 Integer Factorization by Elliptic Curves .................. 53
3.3 Module Source Code for Setting the PARI seed Variable ... 54
3.4 Gcd over Algebraic Number Fields - Part I ................ 58
3.5 Gcd over Algebraic Number Fields - Part II ................ 58
3.6 Karatsuba's Method for Multiplying Polynomials I ........ 61
3.7 Karatsuba's Method for Multiplying Polynomials II ....... 62

4.1 How to Replace the Default print Method on Slaves ........ 67

5.1 Components of the MuPAD Working Environment ............ 71
## List of Tables

2.1 User Interface Navigator ........................................... 14

7.1 Starting up and Running the Network ............................. 88
7.2 Source Files of Module net ......................................... 92
7.3 Files of the Test Suite ............................................... 94

8.1 Times Used by the Gcd Algorithm Measured in Seconds ....... 100
8.2 Times Used by karazuba Measured in Seconds ................. 100
8.3 Times Used for Factoring Integers Measured in Seconds .... 103
Chapter 1

Introduction

This CD-ROM is addressed to users and developers of macro parallel algorithms in MuPAD. It contains trial versions of MuPAD for Linux 2.0 and Solaris 2.5 operating systems, the net module which provides macro parallelism to MuPAD, the complete C/C++ source code of the dynamic module net and an online tutorial and reference manual makman.

1.1 The Concept in Brief

Macro parallelism follows the master-worker paradigm, where the first cluster of the network is interactive and is called master. While the master is connected to a terminal, slaves use a so-called work queue to successively fetch and evaluate tasks [18]. Figure 1.1 shows a simplified scheme of MuPAD macro parallelism.

![Figure 1.1: Simplified Scheme of Macro Parallelism](image)

MuPAD macro parallelism covers three fields of parallel programming: pipes and queues (message passing), global (network) variables and work groups and allows users to re-configure and re-start a heterogeneous network of MuPAD clusters at any time during a MuPAD session.
MuPAD data -i.e. objects and programs- can be distributed by writing to and reading from so-called pipes and queues. Writing can be done locally as well as to remote pipes and queues, while reading is a strictly local operation. Queues may receive data from each cluster of the network, whereas pipes are always one-way connections between two specific clusters (Figure 1.2). Neither the number of pipes and queues nor their naming conventions impose any restrictions to the user. The concept makes it easy to place tasks, input data and results on any cluster of the network.

Furthermore, sharing data between all clusters of the network can be done by use of so-called global (network) variables. In contrast to message passing, this is a more convenient method to distribute and share information. Figure 1.2 gives a brief demonstration how to use these features. Write access to global variables is realized as an atomic operation (see function global) in the sense that they can be used to synchronize tasks over the network. The parallel programming interface offers locks and semaphores for this.

The third class of functions of the MuPAD macro parallelism concerns so-called work groups, realizing automatic scheduling of group jobs (Figure 1.3). Many mathematical algorithms produce subproblems, e.g. using the strategy of divide and conquer, which simply have to be processed by any MuPAD cluster. For this, a convenient parallel scheduling interface is provided: The user can specify one or more so-called work groups, each consists of one group master and an arbitrary number of group slaves. The clusters of a work group are also called workers. Once a job is delivered to a work group, the master selects an idle
worker to process this job and then collects its result. Job scheduling within work groups is fully automatic. The user does not need to keep track which worker is idle and needs a new job and which one is busy. She/He can check if the job is finished, and once it has finished, fetch the result from the group master. Group jobs may also generate new jobs recursively, which can be delivered to its own or any other work group of the network. While waiting for results, workers are able to process further jobs of their work group instead of pure busy waiting or simply suspending themselves. This way, it is very easy to run branch and bound or highly recursive algorithms like Karatsuba.

1.2 Quick Start

Macro parallelism is provided by the dynamic module [24][29] net, which is expected to be installed with MuPAD release 1.4.1 and configured correctly (see appendix A and B). To use a network of 1 + 4 clusters, the user inserts -for example- the following lines into the personal MuPAD startup file ~/.mupadinit:

```plaintext
NETCONF:= [ "wiwianka"=2, "gauss"=1, "poisson"=1 ]:
module( net ):
```

Starting MuPAD with mupad -U master or xmupad -U master runs a local MuPAD cluster as the interactive master, two clusters on the host wiwianka and one cluster on each of both gauss and poisson. The following commands give a first example on using macro parallelism:
Starting up PVM network. This may need some time...

```
>> topology();                          // get number of clusters
   5
>> net::ismaster(), topology(Cluster);  // is master? get cluster-id
   TRUE, 1
>> writequeue("work",3,hold(          // is cluster #3 the master
   writepipe(YesIam,1,net::ismaster()));  // let it write to 'YesIam'
>> readpipe(YesIam,3,Block);          // read answer from pipe
   FALSE
>> global(a,42);                      // set a global variable
>> global(a,global(a)+1), global(a);   // increment global variable
   42, 43
```

Figure 1.4: A First Example on Using Macro Parallelism in MuPAD

1.3 Where to Find Further Information

It is expected that the reader of this manual is familiar with MuPAD and its programming language. Both is described in the manuals [31] and [21]. Note, that the information about parallelism given in manual [31] is outdated.

In addition to *Macro Parallelism*, the concepts of Domains\(^1\) and *Dynamic Modules*\(^2\) perform the key technologies for using MuPAD as an open parallel problem solving environment for mathematical applications. Detailed information about Domains are given in the Manuals [31] [21] and the technical report [6]. The technology of Dynamic Modules is introduced with the *Dynamic Modules User’s Manual and Programming Guide* [29].

Latest information about MuPAD are always available at the MuPAD web site.\(^3\)

1.4 Challenges and Contributions

It is the developer’s concern to provide the general purpose computer algebra system MuPAD as an open and parallel working environment, which can easily be used to perform distributed computations in heterogeneous computer networks.

Since macro parallelism in MuPAD is still an aspect of research and the dynamic module net is still under development, there may be bugs and quirks. Please

---

\(^1\)Axioms and categories; user-defined types; an approach of object-oriented programming

\(^2\)Dynamic linking of C/C++ code; a flexible and efficient concept of software integration

\(^3\)The MuPAD web site: http://www.mupad.de
send bugs as well as your comments and suggestions for improvement to the authors\textsuperscript{4} or the MuPAD team.\textsuperscript{5}

Also if you like to make contributions and like to share macro parallel algorithms with the MuPAD user community, please contact the authors via email.

\textsuperscript{4}Email: \{tom,maradm,andi,steфан\}@mupad.de

\textsuperscript{5}Email: bugs@mupad.de
Chapter 2

Programming Interface

Main features of macro parallelism are message passing via pipes and queues, global (network) variables and automatic job scheduling in work groups. Additionally, methods to interactively configure, start and shutdown the network as well as to set network preferences are provided. This enables users to reconfigure the network at run-time according to their need, which may also be controlled by a MuPAD program like a modular algorithm.

While Section 2.1 gives an overview over all functions subdivided according to the different aspects of macro parallel programming, Section 2.2 lists corresponding help pages in lexicographical order with a detailed description for each of them.

2.1 Overview and Tutorial

The functions listed in this section are subdivided according the following aspects of macro parallel programming: Network Configuration, Network Topology Information, Message Passing, Network Variables, Locks and Semaphores, Automatic Scheduling Primitives and Miscellaneous Functions.

2.1.1 Network Configuration (Startup and Shutdown)

The network configuration specifies which hosts with how many clusters take part on macro parallelism. It is either required before the module net is loaded in a session for the first time -to start the network automatically- or before the
function net::master is called. After a network has been initialized, it can be shutdown at any time and a new network may be configured and initialized.

- net::pref(LogError|ComType)[=value]): get and/or set preferences.
  - LogError is TRUE (default on master) or FALSE. It controls if the master displays messages about errors which occur on slave clusters.
  - ComType is "MCODE" (default) or "ASCII". It controls if MuPAD data are transferred as plain text or in a MuPAD specific binary format.

- NETCONF:=["host_1"=clusters_1,...]: configures the network. host_k is a hostname and clusters_k specifies the number of clusters to be started. A new configuration comes into effect when net::master() is executed.

- net::master(): starts a network using NETCONF. It returns TRUE if the network was started successfully and FALSE otherwise.

- net::shutdown(): shuts down the network. The master simulates a network of topology 1 (meaning the network consist of one cluster) if module net is loaded but no network is running.

- net::halt(): halts the network and quits the master (emergency exit).

The example shows how to activate two network configurations successively:

```
>> NETCONF := [ "wiwianka" = 1, "goedel" = 1 ]:
>> net::master(); // starts with config-I
Starting up PVM network. This may need some time...
TRUE
>> topology(); // number of clusters?
3
>> net::shutdown(); // shutdown network
TRUE

>> NETCONF := [ "wiwianka" = 1, "goedel" = 1, "horner" = 1 ]:
>> net::master(); // starts with config-II
Starting up PVM network. This may need some time...
TRUE
>> topology(); // number of clusters?
4
>> net::shutdown(); // shutdown network
TRUE
```

Figure 2.1: Configuration of a Macro Parallel Network
2.1.2 Network Topology Information

The following network topology information functions are currently available.

- `net::ismaster()`: returns TRUE on the master and FALSE otherwise.
- `net::topology([Cluster])`: returns the current number of clusters or the cluster-id number of the local cluster if option Cluster is used.

The example below demonstrates how to use the topology information functions:

```
>> topology(); // current number of clusters?
4

>> writequeue("work", 2, hold(writequeue(res, 1, topology(Cluster)+100)));
>> readqueue(res, Block); // cluster #2 sends its id +100
102

>> net::ismaster(); // checks if this is the master
TRUE
```

Figure 2.2: Using Topology Information Functions

2.1.3 Message Passing (Queues and Pipes)

Queues and pipes of arbitrary names are supported. Queues with the reserved name work are used as the input device of slave clusters. Note, that in computer algebra systems like MuPAD there is no difference between data and programs. Thus, the term message passing may be used here in an unusual way.

- `net::readqueue(name [,Block])`: reads from local queue name. Block effects a blocking read, otherwise `DOM.NULL` is returned on an empty queue.
- `net::readpipe(name, cid [,Block])`: reads from local pipe name×cid. It behaves like function `net::readqueue`.
- `net::writequeue(name, cid, v)`: writes v to queue name of cluster cid.
- `net::writepipe(name, cid, v)`: writes v to pipe name of cluster cid.

The example distributes tasks and collects their results via message passing:
2.1.4 Network Variables (Global Variables)

Global network variables can be used to share information in the whole macro parallel network. Writing to global variables is an atomic operation which cannot be spoiled by parallel write accesses from other clusters.

- `net::global(var)`: reads out the global network variable `var`.
- `net::global(var, val)`: assigns `val` to `var`. `var` is not evaluated. While evaluating `val`, `var` cannot be changed by others. The previous value of `var` (or `DOM_NULL`, if it was not assigned a value before) is returned.
- `net::global(level, var [, , val] )`: like function `global`, but evaluates `var` with the substitution level `level`.

The following example demonstrates how to place tasks within a macro parallel network using global variables to share information:

```plaintext
>> global(count, 0); // no results yet

>> writequeue("work", 2, hold(( // spawn task #1
  &> global(res1, cos(PI/2));
  global(count, global(count)+1) ));

>> writequeue("work", 3, hold(( // spawn task #2
  &> global(res2, exp(PI*I));
  global(count, global(count)+1) ));

>> while (global(count) < 2) do end_while; // wait for results

>> global(res1), global(res2); // collect results
  0, -1
```

Figure 2.4: Using Global Variables
2.1.5 Locks and Semaphores

Since writing to global variables is an atomic operation, globals are also used as locks and semaphores for network synchronizations.

- `net::initlock(var)`: initializes the lock variable var.
- `net::setlock(var)`: tries to set the lock variable var.
- `net::spinlock(var)`: waits (blocked) until it is allowed to lock var.
- `net::unsetlock(var)`: resets the lock variable var.
- `net::initsem(var)`: initializes the semaphore var, also refer to the functions `net::spinsem`, `net::acquiresem` and `net::freesem`.

The next example illustrates how locks can be used to share information:

```
>> net::initlock(test); net::setlock(test); // init and set lock
>> global(input, 1); // init global variable
>> writequeue("work", 2, hold(
   &> net::spinlock(test); // spawn task on #2
   &> writequeue(res,1,global(input)); // pass input to master
   &> net::unsetlock(test))); // unset lock variable

>> global(input, 2); // set global variable
>> net::unsetlock(test); // starts slave #2
>> readqueue(res, Block); // read the result
```

Figure 2.5: Using Locks and Semaphores

2.1.6 Automatic Scheduling Primitives (Work Groups)

Several work groups can be specified to work independently from another or in combination. They provide automatic job scheduling in a convenient way.

- `net::gmasterinit()` initializes the local cluster as a group master.
- `net::gmasterquit()` terminates the cluster’s function as group master and frees its group slaves. This function fails if there are unfinished jobs.
- `net::gslaveinit(gmaster)`: initializes the cluster as slave of `gmaster`.
- `net::gslavequit()`: terminates one cluster's function as group slave.
- `net::gmaster()`: determines the group master or returns 0 otherwise.
- `net::gcluster()`: The function returns 0=cluster not assigned to a work group, 1=is a group master or 2=is a group slave.
- `net::gfree(withmaster)`: locates all clusters which are not part of any work group. Set `withmaster` to `FALSE` to exclude the master of macro parallelism from the search. Set it to `TRUE` otherwise.
- `net::gjob(gmaster, job)`: delivers job to group master `gmaster` and returns a handle.
- `net::gstatus(gmaster, ghandle)`: determines the status of a job `ghandle` at the group master `gmaster`: 0=not finished, 1=finished.
- `net::gresult(gmaster, ghandle)`: gets the result of the job specified by `ghandle` from the group master `gmaster`.
- `net::gprocess()`: recursively processes a new group job.

The next example demonstrates in brief how work groups can be used for automatic job scheduling in macro parallel networks:

```plaintext
>> net::gmasterinit();  // init group master
>> writequeue("work", 2, hold( net::gslaveinit(1) )); // hire a worker
...

>> h1:= net::gjob(1, hold(fact)(3000) );  // spawn a job #1
>> h2:= net::gjob(1, hold(fact)(2980) );  // spawn a job #2
>> h3:= net::gjob(1, hold(fact)(20) );    // spawn a job #3
>> while net::gstatus(1,h3) = 0 do net::gprocess() end_while;
...

>> net::gresult(1,h1) / net::gresult(1,h2) - net::gresult(1,h3)

3272351806998946601411115705186520762217264822747415575872950927360000
```

Figure 2.6: Using Work Groups

### 2.1.7 Miscellaneous Functions

This section lists some miscellaneous functions of macro parallelism, added to the user interface to make programming more convenient.
Reference Manual

• `net::compute(set, task)`: computes a task on a set of clusters.
• `net::define(set, var, val)`: assigns the expression `val` to the (local) variable `var` on a set of clusters.
• `net::doc([func])`: displays plain text online documentation.
• `net::map(list, func [,arg1 ...])`: applies the function `func` to the elements of list `list` in parallel and returns the list of results.
• `net::mprint(data)`: displays object `data` on the master's console.

The example below shows how to use these functions in parallel programs:

```plaintext
>> net::map([1.0, 2.0, PI], cos);
[0.5403023058, -0.4161468365, -1]
>> writequeue("work", 2, hold(net::mprint("Message from cluster #2")));
>> writequeue("work", 3, hold(net::mprint("Message from cluster #3")));
"Message from cluster #2"
"Message from cluster #3"
>> net::define(All, hold(topo), net::topology);
{2, 3}
>> net::compute(All, hold((cid:=topo(Cluster):
    net::mprint("Cluster #".cid." is alive")) ));
{2, 3}
"Cluster #2: is alive"
"Cluster #3: is alive"
```

Figure 2.7: Miscellaneous Functions of Macro Parallelism

Using this basic set of functions, macro parallelism can be extended with user library functions to make macro parallel programming even more convenient.

2.2 Reference Manual

This section lists help pages with detailed descriptions for all functions available with the dynamic module `net`. The following help pages are listed in lexicographical order.
For a brief overview over all functions, subdivided according to the different aspects of macro parallel programming, refer to tutorial part in Section 2.1.

Table 2.1 lists all functions available and helps to navigate quickly between the tutorial and reference part when using the online version of this manual.

<table>
<thead>
<tr>
<th>Tutorial Section</th>
<th>Page</th>
<th>Function Help Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Online Documentation</td>
<td>12</td>
<td>doc</td>
</tr>
<tr>
<td>Network Configuration</td>
<td>7</td>
<td>NETCONF, master, halt, pref, shutdown</td>
</tr>
<tr>
<td>Topology Information</td>
<td>9</td>
<td>ismaster, topology</td>
</tr>
<tr>
<td>Message Passing</td>
<td>9</td>
<td>readpipe, readqueue, writepipe, writequeue</td>
</tr>
<tr>
<td>Network Variables</td>
<td>10</td>
<td>global, globale</td>
</tr>
<tr>
<td>Network Locks</td>
<td>11</td>
<td>initlock, setlock, spinlock, unsetlock</td>
</tr>
<tr>
<td>Network Semaphores</td>
<td>11</td>
<td>acquiresem, freesem, initsem, spinsem</td>
</tr>
<tr>
<td>Work Groups</td>
<td>11</td>
<td>gcluster, gfree, gjob, gmaster, gmasterinit, gmasterquit, gprocess, gresult, gslaveinit, gslavequit, gstatus</td>
</tr>
<tr>
<td>Parallel Evaluation</td>
<td>12</td>
<td>compute, define, map</td>
</tr>
<tr>
<td>Output Function</td>
<td>12</td>
<td>mprint</td>
</tr>
</tbody>
</table>

Table 2.1: User Interface Navigator

`net::acquiresem` - tries to acquire a network semaphore

V1.4.1

Call:

`net::acquiresem(ident)`

Parameter:

`ident` — identifier, name of a network variable

Synopsis:

`net::acquiresem(ident)` tries to decrement the network semaphore associated with the identifier `ident`. If the semaphore was zero before, i.e. all instances
of the protected resource are in use, the value of the semaphore remains zero and the function returns TRUE. Otherwise, the semaphore is decremented and FALSE is returned, which means that one instance of the protected resource can be used. net::acquiresem is non-blocking.

If the value FALSE is returned, net::freesem must be used later to indicate that the protected resource is not needed any longer. Network semaphores must be initialized via net::initsem before use.

Examples:

```c
>> net::initsem(s,2):
    net::acquiresem(s);

    FALSE

>> writequeue("work",2,hold((b:=net::acquiresem(s);global(L,b))));
   writequeue("work",3,hold((b:=net::acquiresem(s);global(K,b))));

>> global(L);
    global(K);

    FALSE
    TRUE
```

See also:

net::freesem, net::global, net::initsem, net::setlock, net::spinsem

---

**net::compute** – computes a task on a set of clusters

V1.4.1

Call:

`net::compute(cset, expr)`

Parameter:

- **cset** — a finite set of clusters or the identifier `All`
- **expr** — expression, the expression to be computed
Synopsis:

`net::compute` computes the expression `expr` on each cluster specified in the finite set `cset`. If option `All` is given instead of a set, `expr` is computed on all slave clusters. Note, that the master of the network is always excluded. `net::compute` returns the set of clusters on which `expr` will be executed.

Examples:

```cpp
>> net::compute(All, hold((cid:=expr2text(net::topology(Cluster));
  net::mprint("Cluster ", cid, ": is alive")))

{2, 3}
"Cluster #2: is alive"
"Cluster #3: is alive"
```

See also:

`net::define, net::global, net::writequeue`

---

**net::define** – distributes a user definition

V1.4.1

Call:

`net::define(cset, ident, expr)`

Parameter:

- `cset` — a finite set of clusters or the identifier `All`
- `ident` — identifier, the identifier to be defined
- `expr` — expression, the value to be assigned

Synopsis:

`net::define` distributes the definition `ident:=expr` to each cluster specified in the finite set `cset`. If option `All` is given instead of a set, the definition is distributed to all slave clusters. Note, that the master of the network is always
net::define returns the set of clusters on which definition will be performed.

Note, that some MuPAD variables are protected (e.g. names of built-in functions) and thus cannot be directly overwritten using this function.

Examples:

```
>> net::define (All, hold(mycid), net::topology);

{2, 3}

>> net::compute (All, hold((cid:= mycid(Cluster):
    net::mprint("Cluster #".cid." is alive"))):

"Cluster #3: is alive"
"Cluster #2: is alive"
```

See also:

- net::compute, net::global, net::writequeue

---

**net::doc** — plain text online documentation

V1.4.1

Call:

```
net::doc()
net::doc(func)
```

Parameter:

```
func — string, function name without the prefix 'net::'
```

Synopsis:

- `net::doc` displays a brief description of the dynamic module `net` respectively the help page of the module function `net::func`.

Examples:

```
>> net::doc("doc");
```
FUNCTION:
net::doc - online documentation of the net module

SYNOPSIS:
net::doc()
net::doc(func)

DESCRIPTION:
net::doc displays a brief description of the dynamic module net respectively the help page of the function net::func.

SEE ALSO:
module::help, info

See also:
module::help, info

net::freesem - frees a network semaphore
V1.4.1

Call:
net::freesem(ident)

Parameter:
ident — identifier, name of a network variable

Synopsis:
net::freesem(ident) increments the network semaphore associated with the identifier ident, indicating that one instance of the protected resource is no longer in use. The function returns the old value of the network semaphore (see net::initsem) ident.

Examples:
>> net::initsem(s,2):
    net::acquiresem(s);
FALSE

>> writequeue("work",2,hold((b:=net::acquiresem(s);global(L,b))));
   writequeue("work",3,hold((b:=net::acquiresem(s);global(K,b))));

>> global(L), global(K), net::freesem(s);
   writequeue("work",3,hold((b:=net::acquiresem(s);global(K,b))));

   FALSE, TRUE, 0

>> global(K);

   FALSE

See also:

net::acquiresem, net::global, net::initsem, net::spinsem,
net::unsetlock

net::gcluster – determines a cluster’s group status

V1.4.1

Call:

net::gcluster()

Synopsis:

net::gcluster returns the group status of the local cluster: 0 means that it is not a member of any work group, 1 means that it is a group master, 2 means that it is a group slave. To use work groups, a network must be started up.

Examples:

   >> net::gcluster();

   0

See also:

net::gfree, net::gmaster, net::master, net::topology
net::gfree – locates clusters not assigned to work groups
V1.4.1

Call:

net::gfree(withmaster)

Parameter:

withmaster — boolean, including the master

Synopsis:

net::gfree(withmaster) locates all clusters within the macro parallel network that are not assigned to any work group yet. Set withmaster to TRUE to include the master of the network in the search. Set it to FALSE to exclude it.

To use work groups, a network must be started up.

Examples:

>> net::gfree(FALSE);

   Error: network has not been started up [net::gfree]

>> NETCONF:= ["localhost"=2]:
   net::master();
   net::gfree(FALSE);
   net::gfree(TRUE);

   Starting up PVM network. This may need some time...
   {2,3}
   {1,2,3}

See also:

net::gcluster, net::gmaster, net::master, net::topology
net::gjob – delivers a job to a group master
V1.4.1

Call:

net::gjob(gmaster, expr)

Parameter:

- gmaster — integer, cluster number of a group master
- expr — expression, job to be delivered

Synopsis:

net::gjob delivers the job expr to the group master with the cluster identification number gmaster and returns a so-called group job handle of type DOM_INT.

This handle can be used to determine status information and to fetch the job’s result from this group master later. Every cluster of the macro parallel network can send jobs to any group master.

To use work groups, a network must be started up.

Examples:

```plaintext
>> handle:= net::gjob( 2, hold(int(cos(x)*sin(x),x)) ):
   while( net::gstatus(2,handle)<>1 ) net::gprocess() end_while:
   net::gresult(2,handle);

    2
   ----- 2

sin(x)
```

See also:

net::gmaster, net::gprocess, net::gresult, net::gstatus, net::master

net::global – accesses a network variable
V1.4.1

Call:

\[
\text{net}::\text{global(\text{ident})}
\]

\[
\text{net}::\text{global(\text{ident expr})}
\]

Parameter:

\[
\begin{align*}
\text{ident} & \quad \text{identifier, name of a network variable} \\
\text{expr} & \quad \text{expression, value to be assigned}
\end{align*}
\]

Synopsis:

\[
\text{net}::\text{global(\text{ident})} \quad \text{returns the current value of the network variable \text{ident}. If \text{ident} was not assigned a value before, the value of type \text{DOM_NULL} is returned. In contrast to \text{expr}, the parameter \text{ident} is not evaluated.}
\]

\[
\text{net}::\text{global(\text{ident, expr})} \quad \text{assigns the value \text{expr} to the network variable \text{ident} and returns its previous value. If \text{ident} was not assigned a value before, the value of type \text{DOM_NULL} is returned. This function call is atomic in the following sense: the function is blocked and \text{ident} cannot be changed by other processes during the evaluation of \text{expr}. \text{expr} must not contain another write access to any network variable, whereas read access is allowed explicitly.}
\]

Writing accesses to network variables involve network wide communication and synchronization mechanisms for atomicity which are highly expensive. Excessive use of network variables will sequentialize a parallel program and nullifies the effect of parallelizing problems.

Examples:

\[
\gg \text{net}::\text{global(c); net}::\text{global(c,2); net}::\text{global(c);}
\]

\[
2
\]

See also:

\[
\text{net}::\text{globale, net}::\text{initlock, net}::\text{initsem}
\]

\[
\text{net}::\text{globale} - \text{accesses a network variable}
\]
V1.4.1

Call:

\[
\begin{align*}
\text{net::globale(slevel, expr1)} \\
\text{net::globale(slevel, expr1, expr2)}
\end{align*}
\]

Parameter:

\[
\begin{align*}
\text{slevel} & \quad \text{integer, substitution level for expr1} \\
\text{expr1} & \quad \text{expression, name of network variable} \\
\text{expr2} & \quad \text{expression, must evaluate to identifier}
\end{align*}
\]

Synopsis:

In contrast to \text{net::global}, function \text{net::globale} first evaluates parameter \text{expr1} with substitution level \text{slevel}. \text{expr1} must evaluate to an identifier, otherwise the evaluation is aborted with an error message. This function is an analogue to \text{evalassign} for network variables.

\text{net::globale(slevel, expr1)} returns the current value of the network variable specified by \text{expr1}. If this variable was not assigned a value before, the value of type \text{DOM\_NULL} is returned.

\text{net::globale(slevel, expr1, expr2)} assigns the value \text{expr2} to the network variable specified by \text{expr1} and returns its the previous value. If this variable was not assigned a value before, the value of type \text{DOM\_NULL} is returned.

This function call is atomic in the following sense: the function is blocked and \text{ident} cannot be changed by other processes during the evaluation of \text{expr2}. \text{expr2} must not contain another write access to any network variable, whereas read access is allowed explicitly.

Writing accesses to network variables involve network wide communication and synchronization mechanisms for atomicity which are highly expensive. Excessive use of network variables will sequentialize a parallel program and nullifies the effect of parallelizing problems.
Examples:

```plaintext
>> global(c,3);
   a:=c:
   3

>> global(a);

>> globale(MAXLEVEL,a);
   3
```

See also:

evalassign, net::global, net::initlock, net::initsem

---

**net::gmaster** — determines a cluster's group master

**V1.4.1**

**Call:**

```plaintext
net::gmaster()
```

**Synopsis:**

`net::gmaster` determines the group master of the local cluster and returns its identification number. If the local cluster is not a member of any work group, this function returns the value 0. To use work groups, a network must be started up.

**Examples:**

```plaintext
>> net::gmaster();
   0
```

See also:

`net::gcluster, net::gfree, net::master, net::topology`
net::gmasterinit – initializes a group master  
V1.4.1

Call:

net::gmasterinit()  

Synopsis:

net::gmasterinit initializes the local cluster as group master of a new work group. Use the function net::gslaveinit to hire group workers.

To use work groups, a network must be started up.

Examples:

>> net::gmasterinit();

See also:

net::gmasterquit, net::gslaveinit, net::gslavequit, net::gmaster

net::gmasterquit – deinitializes a group master  
V1.4.1

Call:

net::gmasterquit()  

Synopsis:

net::gmasterquit terminates the local cluster’s function as group master. Its work group will be dissolved: connected group slaves will be freed similarly to issuing a separate net::gslavequit command on each of them. This function fails if there are unfinished group jobs.

To use work groups, a network must be started up.

Examples:

>> net::gmasterquit();
Error: gmasterquit on non-gmaster [net::gmasterquit]

See also:

net::gmasterinit, net::gslaveinit, net::gslavequit, net::gmaster

net::gprocess - processes another group job first
V1.4.1

Call:

net::gprocess()

Synopsis:

net::gprocess makes a group slave suspend the current job and process another job of its work group recursively before continuing with the current one. If the group job queue is empty, the current job is continued right away. If net::gprocess is executed on a group master, it aborts with an error message. This function supports highly recursive algorithms like Karatsuba multiplication or branch-and-bound algorithms within the work group concept.

To use work groups, a network must be started up.

Examples:

>> net::gprocess();

See also:

net::gjob, net::gresult, net::gstatus, net::gmaster
Reference Manual

net::gresult – fetches the result of a group job

V1.4.1

Call:

net::gresult(gmaster, ghandle)

Parameter:

- gmaster — integer, cluster number of a group master
- ghandle — integer, group job handle

Synopsis:

net::gresult(gmaster, ghandle) returns the result of the group job with handle ghandle controlled by the group master with cluster identification number gmaster.

To use work groups, a network must be started up.

Examples:

```plaintext
>> handle:= net::gjob( 2, hold(int(cos(x)*sin(x),x)) ):
  while( net::gstatus(2,handle)<>1 )
    net::gprocess()
  end_while:
  net::gresult(2,handle);

  2
sin(x)
-----
  2
```

See also:

net::gjob, net::gprocess, net::gstatus, net::gmaster
net::gslaveinit - initializes a group slave

Call:

net::gslaveinit(gmaster)

Parameter:

gmaster — integer, cluster-id of a group master

Synopsis:

net::gslaveinit(gmaster) initializes the local cluster as group slave assigned to the group master with cluster identification number gmaster. To use work groups, a network must be started up.

Examples:

» net::gslaveinit(1);

See also:

net::gmasterinit, net::gmasterquit, net::gslavequit, net::gmaster

---

net::gslavequit - deinitializes a group slave

Call:

net::gslavequit()

Synopsis:

net::gslavequit terminates the local cluster's function as group slave. To use work groups, a network must be started up.
Examples:

```latex
>> net::gslavequit();
```

See also:

```latex
net::gmaster, net::gmasterquit, net::gsaveinit, net::gmaster
```

```latex
net::gstatus – determines the status of a group job
V1.4.1
```

Call:

```latex
net::gstatus(gmaster, ghandle)
```

Parameter:

```latex

gmaster — integer, cluster-id of a group master
ghandle — integer, group job handle
```

Synopsis:

```latex
net::gstatus returns the status of the group job with handle ghandle con­
trolled by the group master with cluster identification number gmaster. It
returns the value 1, if the job is finished and 0 otherwise. Once the job is
finished, one can fetch its result with the function net::gresult.
```

To use work groups, a network must be started up.

Examples:

```latex
>> handle:= net::gjob( 1, hold(diff(ln(1/x),x)) ) :
    net::gstatus(1,handle);

  0
```

See also:

```latex
net::gjob, net::gprocess, net::gresult, net::gmaster
```
**net::halt** – **halts the network**  
V1.4.1

Call:

```c
net::halt()
```

**Synopsis:**

`net::halt` terminates all clusters of the macro parallel network instantly. In contrast to function `net::shutdown`, here also the master of the network is terminated. This function is meant as an emergency stop.

**Examples:**

```c
>> net::halt();
```

**See also:**

`net::master, net::shutdown`

---

**net::initlock** – **initializes a network lock**  
V1.4.1

Call:

```c
net::initlock(ident)
```

**Parameter:**

`ident` — identifier, name of a network variable

**Synopsis:**

`net::initlock(ident)` initializes the network variable `ident` with the value `FALSE` for subsequent use with the functions `net::setlock`, `net::spinlock` and `net::unsetlock`. 
Remember, if the network lock ident has the value FALSE or TRUE, also the network variable ident is changed to this value.

Network locks are used for managing resources which can be used by only one cluster of the macro parallel network at a time.

Examples:

```plaintext
>> net::initlock(c): global(c);
   net::setlock(c), global(c);

   FALSE
   FALSE, TRUE

>> writequeue("work",2,hold((b:=net::setlock(c);global(L,b)));
   global(L), net::unsetlock(c);

   TRUE, TRUE
```

See also:

net::global, net::initsem, net::setlock, net::spinlock, net::unsetlock

---

net::initsem — initializes a network semaphore

V1.4.1

Call:

```plaintext
net::initsem(ident, integer)
```

Parameter:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ident</td>
<td>identifier, name of a network variable</td>
</tr>
<tr>
<td>count</td>
<td>positive integer, value to be assigned</td>
</tr>
</tbody>
</table>

Synopsis:

```plaintext
net::initsem(ident, count) initializes the network variable ident with the integer value count for subsequent use with the functions net::acquiresem, net::spinsem and net::freesem.
```
Remember, if the network semaphore ident has the value count, also the network variable ident is changed to this value.

Network semaphores are used for managing resources which can be used only count times on the different clusters of the macro parallel network.

Examples:

```plaintext
>> net::initsem(s,2): global(s);
    net::acquiresem(s), global(s);

  2
  FALSE, 1

>> writequeue("work",2,hold((b:=net::acquiresem(s);global(L,b)))):
    writequeue("work",3,hold((b:=net::acquiresem(s);global(K,b))));

>> global(L), global(K);

  FALSE, TRUE
```

See also:

- net::acquiresem, net::freesem, net::global, net::initlock, net::spinsem

---

**net::ismaster** - detects the master of macro parallelism

**V1.4.1**

Call:

```plaintext
net::ismaster()
```

Synopsis:

`net::ismaster()` returns TRUE if the local cluster is the master of the macro parallel network and returns FALSE otherwise.
Examples:

```>
net::ismaster();
```

TRUE

See also:

net::gmaster, net::master, net::topology

---

**net::map** — applies a function to each element of a list

V1.4.1

**Call:**

```c
net::map(list, func)
```

```c
net::map(list, func, expr_1, ...)
```

**Parameter:**

- `list` — list of expressions
- `func` — function or any object that can be applied like a function
- `expr_1` — expression

**Synopsis:**

`net::map` applies the function `func` to each element of the list `list`. When `func` is executed, the expressions `expr_1`, ... are used as additional arguments.

`net::map` works much like the map kernel function, but the function calls are distributed over the macro parallel network. The function `net::map` may only be invoked by the master. Expressions like `net::map([a1, ..., an], f)` are evaluated to `[f(a1), ..., f(an)]` where the `n` calls to `f` are distributed among the slaves. If the number of slaves exceeds the number of list elements, some slaves remain idle; if the number of list elements exceeds the number of slaves, some or all slaves are given more than one function call to evaluate.

If you want to evaluate user-defined functions via `net::map`, note that only the function calls `func(...)` are distributed to the slaves automatically, but not
the definition of function \texttt{func} itself. Second-level functions must be distributed to the slaves explicitly.

Examples:

```plaintext
>> # distribute second-level function definition to slaves #
  for s from 2 to topology() do
    writequeue("work", s, hold((
      g := proc(n)
        begin
          [ n, sin(float(n)), topology(Cluster)];
          end_proc;
        )
    ));
  end_for:

# define first-level function on master #########################
f := proc(n)
  begin
    g(n)
  end_proc:

# distribute function calls ####################################
1 := [1, 2, 3, 4, 5, 6, 7, 8]:
net::map(1, f);

[[1, 0.8414709848, 2],[2, 0.9092974268, 3],
 [3, 0.1411200080, 3],[4,-0.7568024953, 3],
 [5,-0.9589242746, 3],[6,-0.2794154980, 3],
 [7, 0.6569865987, 3],[8, 0.9893582466, 3]]

>> net::map(1, _plus,12);

[13, 14, 15, 16, 17, 18, 19, 20]

See also:

map
net::master — starts up a macro parallel network
V1.4.1

Call:

net::master()

Synopsis:

net::master reads the network configuration specified with NETCONF and starts up the macro parallel network accordingly. The function returns TRUE if the network has been started.

Note that a network is started even if some of the hosts respectively clusters specified in NETCONF could not be added to it. Use function topology to check if all clusters were started up.

NETCONF must be a list of equations with the left sides are strings naming hosts and the right sides are positive integers specifying the number of MuPAD clusters to start on this host.

Starting up the network may need some seconds, depending on your network configuration and your hardware and software. The network can be shutdown and reconfigured during the session.

Note, that the program ssh (secure shell client) is used by the underlying network service to startup remote MuPAD clusters.

Examples:

```matlab
>> NETCONF := [ "localhost" = 2 ];
   net::master();

   TRUE
```

See also:

net::halt, net::ismaster, net::shutdown, net::topology
net::mprint – prints to the master’s console
V1.4.1

Call:

net::mprint(expr)

Parameter:

expr — expression, value to be displayed

Synopsis:

net::mprint(expr) displays the object expr on the master's console. This function can be used to let slave clusters print protocol or debug messages.

Examples:

>> net::mprint("test");

"test"

See also:

net::ismaster, net::master, net::topology

net::pref – returns/sets network preferences
V1.4.1

Call:

net::pref(option)
net::pref(option=value)

Parameter:

option — identifier, the name of an option
value — value, depends on the option
Synopsis:

net::pref returns respectively sets a network preference of macro parallelism. The following options are supported at present:

**LogError**: Display of error messages is FALSE by default. If the network is started, this is changed to TRUE on the master.

*Default Value: FALSE*
*Possible Values: TRUE, FALSE*

**GarCol**: Switches the garbage collector on and off. The garbage collector is used to free unused memory cells.

*Default Value: TRUE*
*Possible Values: TRUE, FALSE*

**ComType**: Selects the encoding for sending MuPAD data: "MCODE" is a MuPAD specific binary format. Up to the ALPHA release of MuPAD 1.4.1 (Aug.'98) it does not work in heterogeneous networks. This is fixed in later versions. "ASCII" supports heterogeneous networks but does not allow to send data which have no text representation in MuPAD (e.g. funs, refer to function text2expr and expr2text).

Note: this preference must be set before the macro parallel network is started up. Thus, heterogeneous networks currently cannot be started using the MuPAD option `-U master`.

*Default Value: "MCODE"*
*Possible Values: "ASCII", "MCODE"*

Examples:

```plaintext
>> net::pref(ComType); net::pref(LogError);
net::pref(LogError=FALSE): net::pref(LogError);

"MCODE"
TRUE
FALSE
```

See also:

Pref
net::readpipe - reads from a network pipe
V1.4.1

Call:

net::readpipe(expr, integer)
net::readpipe(expr, integer, Block)

Parameter:

expr — expression, name of the pipe
integer — positive integer, number of a cluster
Block — identifier, option

Synopsis:

net::readpipe(expr, integer) reads from the pipe expr which is connected with the cluster named integer. If the pipe is empty, the function returns the value of type DOM.NULL.

If option Block is used and the pipe is empty, execution is blocked until a value has arrived.

Examples:

>> writepipe("a", 1, hello): writepipe("a", 1, world):
   readpipe("a", 1), readpipe("a", 1);

   hello, world

>> writequeue("work", 2, hold(writepipe("a", 1,"macro"))):
   writequeue("work", 3, hold(writepipe("a", 1,"parallelism"))):
   readpipe("a", 2, Block), readpipe("a", 3, Block);

   macro, parallelism

See also:

net::readqueue, net::writepipe, net::writequeue
net::readqueue – reads from a network queue
V1.4.1

Call:

net::readqueue(expr)
net::readqueue(expr, Block)

Parameter:

expr — expression, name of the queue
Block — identifier, option

Synopsis:

net::readqueue(expr) reads from the queue expr. If the queue is empty, it returns the value of type DOM_NULL.

If option Block is used and the queue is empty, execution is blocked until a value has arrived.

Examples:

```>
writequeue("a", 1, hello): writequeue("a", 1, world):
readqueue ("a"), readqueue ("a");
```

hello, world

```>
writequeue("work", 2,
    hold(writequeue("c", 1, readqueue("b", Block))))):
writequeue("b", 2, "traveler"): readqueue ("c", Block);
```

"traveler"

See also:

net::readpipe net::writepipe, net::writequeue
net::setlock – sets a network lock

V1.4.1

Call:

net::setlock(ident)

Parameter:

ident — identifier, name of a network variable

Synopsis:

net::setlock(ident) sets the network lock associated with the identifier ident. If the lock was set before, the function returns TRUE, meaning that the protected resource is already in use. Otherwise, it returns FALSE, meaning that the protected resource can be used. net::setlock is non-blocking.

If FALSE is returned, net::unsetlock must be used later to indicate that the protected resource is not needed any longer. Network locks must be initialized via net::initlock before use.

Examples:

```c
>> net::initlock(c):
   net::setlock(c);

   FALSE

>> writequeue("work",2,hold((b=net::setlock(c);global(L,b)))):
   global(L), net::unsetlock(c);

   TRUE, TRUE
```

See also:

net::acquiresem, net::global, net::initlock, net::spinlock, net::unsetlock
net::shutdown – shuts down the macro parallel network

V1.4.1

Call:

net::shutdown()

Synopsis:

net::shutdown terminates all slaves of the macro parallel network. It returns the value TRUE if the network was shutdown successively and returns FALSE otherwise.

In contrast to function net::halt, here the master of the macro parallel network is not terminated but still running. A new network may be configured in NETCONF and started with function master.

Examples:

>> net::shutdown();

TRUE

See also:

net::halt, net::ismaster, net::master, net::topology

net::spinlock – sets a network lock, blocking

V1.4.1

Call:

net::spinlock(ident)

Parameter:

ident — identifier, name of a network variable
Synopsis:

`net::spinlock(ident)` waits until the network lock associated with the identifier `ident` is free and then sets it.

`net::unsetlock` must be used later to indicate that the protected resource is not needed any longer. Network locks must be initialized via `net::initlock` before use.

Examples:

```plaintext
>> net::initlock(c): global(L,0): net::setlock(c);
    FALSE

>> writequeue("work", 2, hold((
       net::spinlock(c): global(L,42): net::unsetlock(c ))));

>> global(L), net::unsetlock(c);
    0, TRUE

>> global(L);
    42
```

See also:

`net::global, net::initlock, net::setlock, net::spinsem, net::unsetlock`

---

`net::spinsem` – acquires a network semaphore, blocking

V1.4.1

Call:

`net::spinsem(ident)`

Parameter:

`ident` — identifier, name of a network variable
Synopsis:

net::spinsem(ident) waits until the network semaphore associated with the identifier ident is greater than zero and then decrements it, meaning that one more instance of the protected resource can be used.

net::freesem must be used later to indicate that the protected resource is not needed any longer. Network semaphores must be initialized via net::initsem before use.

Examples:

```plaintext
>> net::initsem(s,2):
  net::acquiresem(s);
  FALSE

>> global(L,0);
  global(K,0);
  writequeue("work",2,hold((b:=net::acquiresem(s);global(L,b))));
  writequeue("work",3,hold((net::spinsem(s);global(K,42))));

>> global(L);
  global(K);
  net::freesem(s);
  FALSE
  0
  0

>> global(K);

  42

See also:

net::acquiresem, net::freesem, net::global, net::initsem, net::spinlock
net::topology – informs about the network configuration
V1.4.1

Call:

net::topology()
net::topology(Cluster)

Parameter:

Cluster — identifier, option

Synopsis:

net::topology() returns the number of clusters the current macro parallel network consists of.
net::topology(Cluster) returns the identification number of the cluster on which it is executed. The master of the macro parallelism has always the identification number 1.

Examples:

>> net::topology();

3

>> net::topology(Cluster);

1

See also:

net::gcluster, net::gfree, net::gmaster, net::ismaster
net::unsetlock – unsets a network lock
V1.4.1

Call:

net::unsetlock(ident)

Parameter:

ident — identifier, name of a network variable

Synopsis:

net::unsetlock(ident) unsets the network lock associated with the identifier ident and returns the previous status of ident.

Examples:

```plaintext
>> net::initlock(c):
   net::setlock(c);

   FALSE

>> writequeue("work",2,hold((b:=net::setlock(c);global(L,b)));
   global(L), net::unsetlock(c);

   TRUE, TRUE
```

See also:

net::freesem, net::global, net::initlock, net::setlock, net::spinlock
net::writepipe—writes into a network pipe

V1.4.1

Call:

net::writepipe(expr_1, integer, expr_2)

Parameter:

- expr_1: expression, name of the pipe
- integer: positive integer, number of a cluster
- expr_2: expression, the data written into the pipe

Synopsis:

net::writepipe(expr_1, integer, expr_2) writes expression expr_2 into the pipe expr_1 on the cluster with the identification number integer.

Examples:

```
>> writepipe("a", 1, hello): writepipe("a", 1, world):
    readpipe("a", 1),       readpipe("a", 1);

    hello, world

>> writequeue("work", 2, hold(writepipe("a",1,"macro"))):
    writequeue("work", 3, hold(writepipe("a",1,"parallelism"))):
    readpipe("a", 2, Block), readpipe("a", 3, Block);

    macro, parallelism
```

See also:

net::global, net::readpipe, net::readqueue, net::writequeue
net::writequeue - writes into a network queue

V1.4.1

Call:

net::writequeue(expr_1, integer, expr_2)

Parameter:

expr_1  —  expression, the name of the queue
integer —  positive integer, the number of a cluster
expr_2  —  expression, the data written into the queue

Synopsis:

net::writequeue(expr_1, integer, expr_2) writes expression expr_2 into
the queue expr_1 on the cluster with the identification number integer.
The queue with the reserved name work is used as input device on slave clusters. Thus writing an expression to the work queue instructs a slave to compute it.

Examples:

>> writequeue("a", 1, hello): writequeue("a", 1, world):
    readqueue ("a"),
    readqueue ("a");

hello, world

>> writequeue("work", 2,
    hold(writequeue("c", 1, readqueue("b", Block))))):
    writequeue("b", 2, "traveler"):
    readqueue ("c", Block);

"traveler"

See also:

net::readpipe, net::readqueue net::writepipe
Chapter 3

Examples of Parallel Algorithms

This chapter demonstrates some implementations of macro parallel algorithms in MuPAD. We start with an example for search parallelism and proceed with some mathematical problems: a parallel implementation for computing Fibonacci numbers, parallel algorithms for factoring integers, a modular algorithm for computing greatest common divisors over algebraic number fields and Karatsuba's method of multiplying polynomials. Also refer to Chapter 8.

In the following examples, we assume that a macro parallel network has been properly started up, with at least one slave being available (see Section 2.1.1 how to do this). If too many clusters are started on one host, memory problems are likely to occur.

The examples are installed in the MuPAD directory share/examples/net. Include this path in your READ_PATH with the following commands:

```plaintext
>> module(net): MP:=net::getenv("MuPAD_ROOT_PATH"): MP:=MP."/share/examples/net":
   if type(READ_PATH)=DOM_IDENT then
      READ_PATH:=MP
   elif map({READ_PATH}, bool@equal, MP)={FALSE} then
      READ_PATH:=READ_PATH, MP
   end_if:
   unassign(MP):
```

To get protocol messages during the execution of the following examples, use the command setuserinfo to increase the user information level, e.g.:

```plaintext
>> setuserinfo(Any, 10):
```
It might also be interesting to get user information from slave clusters; refer to Section 4.4 to read how to achieve this. Note that each cluster’s user information level must be set explicitly (e.g. using net::compute); different information levels for different clusters are allowed.

3.1 A very simple example: search parallelism

Suppose we are given a large, unsorted list and want to find one list element that meets some condition; for example, let 1 be a list of random numbers. Such a list can be produced by the following command:

```
>> l:=[random() $i=1..1000]:
```

For example, we could try to find an element smaller than $10^9$. The simple linear search

```
>> for i from 1 to 1000 do
   if l[i]<10^9 then break end_if
   end_for:
   i;
```

can be parallelized by subdividing the interval.

A possible way to do this is shown in Figure 3.1. The searching is done inside the procedure searchlist: the global variable found is used to store the index position of a list element that satisfies the condition; its value is FAIL until such index position is found. Since read accesses to global variables are cheap, every cluster can check in every iteration whether some other cluster has found something. In the main program parsearch, the original list is split into smaller sublists; the sublists are distributed to the slaves, with the last one being taken by the master itself. A global variable ready is used to store the number of tasks finished so far.

This example demonstrates some programming techniques: use of hold to prevent unintended evaluation (that is: execution) of a task by the master itself, use of hold(fp::apply) to evaluate both a function and its arguments, but not the function call.

Some improvements could still be considered. For example, it will be desirable to split large lists into more sublists than there are clusters: whenever a cluster has finished a task, it gets a new one, as it is done in net::map. This is better if some machines in the network are much slower than others, or if list elements are quite likely to satisfy the given condition.
searchlist:=proc(l, f, offset)
local i;
    # l : a list
    # f : a function mapping list elements to TRUE or FALSE
    # offset: an index where to start the search in list l
    # returns i such that f(l[i])=TRUE, or FAIL if no such index exists

begin
    for i from 1 to nops(l) do
        if net::global(found)<>FAIL then
            break
        end_if;
        if bool(f(l[i]))=TRUE then
            net::global(found, i+offset);
            break
        end_if
    end_for;
    net::global(ready, net::global(ready)+1)
end_proc:

parsearch:= proc(l, f)
local i, tasksize;
    # l: a list
    # f: a function mapping list elements to TRUE or FALSE
    # returns i such that f(l[i])=TRUE, or FAIL if no such index exists

begin
    tasksize:=nops(l) div topology();
    net::global(found, FAIL);
    net::global(ready, 0);

    for i from 2 to topology() do
        writequeue("work", i, hold(fp::apply)(searchlist,
            [op(l, (i-2)*tasksize+1 .. (i-1)*tasksize)], f,
            (i-2)*tasksize))
    end_for;

    searchlist([op(l, (topology()-1)*tasksize+1 .. nops(l))],
        f, (topology()-1)*tasksize);

    while net::global(ready) < topology() do #nothing# end_while;
    net::global(found);
end_proc:

Figure 3.1: A Parallel Search Method
A call to the parallel search looks like this:

```plaintext
>> unassign(i):
  l:=[random() $i=1..1000]:
  read("parsearch.mu"): 
  parsearch(l, fun(args(1) < 10^9));
```

It should be finally remarked that, for practical purposes, the use of kernel functions like select or map should be preferred over the use of loops. In our example,

```plaintext
>> ll:=map(l, fun(bool(args(1)<10^9))):
    contains(ll, TRUE);
```

is faster although all list elements have to be compared with $10^9$ in this case.

### 3.2 Factoring Integers Using Elliptic Curves

Another example of a search-parallel algorithm is the elliptic curve method for factoring integers (e.g. see [17]) which is described in this section.

If $n$ is the integer to factor, the computation consists — briefly said — in computing multiples of random points on some random elliptic curves modulo $n$. By the Chinese Remainder Theorem, each curve is isomorphic to a direct product of curves modulo the prime (power) factors of $n$; the computation ends when, on some factor of some curve, a multiple of the order of the chosen point is reached. (At that moment, a division modulo $n$ fails due to the occurrence of a zero divisor — which immediately gives rise to a factorization of $n$.)

The running time of this algorithm greatly depends on a lucky choice of curves. It is important not to stick to "bad" curves too long; hence our algorithm abandons the computation on a given curve after a while. The strategy for this is given by the (optional) second and third argument: after timeout seconds, another curve is tried. Each time a computation is timed out, the value of timeout is increased by increment (it depends on the smallest factor of $n$ how "good" curves can be expected to be, and this factor is unknown.)
parifactor:=proc(n, timeout, increment)
local i;
begin
# Load PARI random generator on each cluster of the network ----------#
net::compute(All, hold(module(parirand)));

# Initialize different start values on each cluster ------------------#
net::compute(All, hold(parirand::setseed(parirand::getseed()+topology(Cluster))));

# Distribute local functions and variables in the network ------------#
net::define(All, hold(timeout), timeout);
net::define(All, hold(increment), increment);
net::define(All, hold(n), n);

# Initialize global variable for result and return condition ---------#
net::global(found, FAIL);

# Start computing 'ifactor' on each cluster of the network ----------#
net::compute(All, hold(( while net::global(found)=FAIL do
    userinfo(3, "Trying curve for ", expr2text(timeout), " seconds");
    if traperror(l:=ifactor(n).timeout)=0
    then net::global(found, l)
    else timeout:=timeout+increment
    end_if
end_while ));

# Wait for the first answer available by any cluster ------------------#
while net::global(found)=FAIL do # nothing # end_while;

# Return the result --------------------------------------------------#
net::global(found);
end_proc:

Figure 3.2: Integer Factorization by Elliptic Curves

The implementation of this algorithm in MuPAD is very easy, since the MuPAD kernel function ifactor already uses the elliptic curve method.

The random curves used by this MuPAD function -respectively the underlying PARI routine- are generated locally on the clusters of the macro parallel network. In order to get different curves on each cluster, we have to manipulate the random curve generator, which means we must initialize the corresponding

---

1This MuPAD function uses a corresponding PARI [3] routine to factor a given integer. Parts of the PARI arbitrary precision arithmetic are integrated in the MuPAD kernel.
random seed variable with a different start value on each cluster. This can be done using a simple dynamic module (refer to the *Dynamic Modules Manual* [29]) which interferes into the MuPAD kernel. Figure 3.3 lists the C/C++ source code of the dynamic module parirand.

![Figure 3.3: Module Source Code for Setting the PARI seed Variable](image)

This example shows an algorithm with the least possible amount of communication. Figure 3.2 lists the complete MuPAD source code of the macro parallel algorithm.

### 3.3 Factoring Integers by Means of a Quadratic Sieve

We proceed with an example for a modular algorithm. Unlike in the previous examples, all slaves' results have to be waited for.

For integers which have no prime factor of less than about 15 decimal digits, the elliptic curve method becomes very slow. The fastest known method in the range up to about 120 digits is the so-called multi-polynomial quadratic sieve.

The key step of the mpqs algorithm consists in finding out which values of certain polynomials are smooth, i.e. factor completely into small primes. Each polynomial is of the form \( ax^2 + bx + c \), with the coefficients \( a, b, c \) depending on a certain integer \( d \). In order to sieve such a polynomial, the MuPAD procedure sievepoly has to be called with this \( d \); some parameters of the algorithm have to be made known in advance to all clusters.

Since the polynomials can be investigated independently from each other, writing a parallel version of the sieve is trivial. With \( d \text{list} \) being a list of \( d \) generated by the master, the MuPAD syntax of the sieve step could be simply
provided that the definition of sievepoly has been distributed by
net::define(hold(sievepoly), sievepoly)
before\textsuperscript{2}, and the number of polynomials to be sieved is known in advance.

Actually, our implementation does a bit more: statistics are collected etc (refer to Section 8.4 for additional information). Thus, the source code looks more like that one of the gcd algorithm in Section 3.4.

To improve the running time, some parts of the algorithm are implemented using dynamic modules. They can be invoked (instead of equivalent code written in MuPAD) by passing additional option(s) to mpqs.

The syntax of a call to mpqs is simply mpqs(n, options), where options can include any of the following:

\textbf{MaxInFactorbase=}fmax Use the list of all primes p smaller than \texttt{fmax} modulo which n is a quadratic residue as a factor base.

\textbf{Factorbase=} fb Use \texttt{fb} as the factor base. \texttt{fb} must be a list of primes.

\textbf{SieveArrayLimit=} M For every polynomial \(f\), the values \(f(-M) \ldots f(M)\) are investigated. \(M\) must be a positive integer.

\textbf{NumberOfPolynomials=} k The number of polynomials the values of which are sieved; default is 1. \(k\) must be a positive integer.

\textbf{Tolerance=} T Factor all polynomial values that seem to have no prime factors larger than \(p^T\), where \(p\) is the largest prime in the factor base. Default=2.0.

\textbf{LargeFactorBound = B} After factoring the polynomial values, throw away those that have a factor larger than \(B\). Default is \(p^T\) (\(p, T\) as above).\textsuperscript{3}

\textbf{ExitWithMissing = m} Stop the sieve step as soon as the sieve reports of at most \(m\) polynomials are still missing (such that the number of polynomials actually processed is only \(k - m\). Setting this to a high value allows continuing even if some (at most \(m\)) slaves break down during the sieve step, and it avoids waiting for very slow slaves.

\textsuperscript{2}Note the difference between the identifier \texttt{sievepoly} (the first argument, protected from unintended local evaluation using the function \texttt{hold}) and its value (the second argument).

\textsuperscript{3}We do not try to factor into any primes outside our factor base, therefore \(T\) should not exceed 2 by too much. (so-called single large prime variation of the \texttt{mpqs} algorithm).
Examples of Parallel Algorithms

Carray Use the dynamic module carray. This module does the key part of the sieve step (marking off – for every prime in the factor base – those polynomial values that are multiples of a certain prime). Because an array of int in C/C++ takes considerably less memory than a list of DOM_INT in MuPAD, this allows to handle larger sieve arrays, in addition to a major speed up. Requires carray.mdm.

InteractiveInput Prompt the user for the values of the parameters.

NetModule Use the parallel version of the sieve. Requires net.mdm.

CollectInformation Return a list containing information on the course of the computation.

SieveOnly Do not solve the linear system at all, return a list of all sieve reports instead.

Gauss Simply use gaussian elimination for the linear system. If this option is not given, a simple form of Wiedemann’s algorithm is used; this one uses the modules sparsemult and gf2fac if sparsemult.mdm and gf2fac.mdm are available, but it works otherwise, too.

ListRep Represent the rows of the coefficient matrix as lists. Without this option, the rows are represented as polynomials. Works only with option Gauss.

To demonstrate examples, we first have to load the MuPAD algorithm:

```latex
>>> read("mpqs.mu"): \\

>>> n:=nextprime(10\textsuperscript{10})*nextprime(10\textsuperscript{12}); \\

1000000001939000000741
```

This example is small enough for the sequential algorithm (and small enough for Gaussian elimination):

```latex
>>> mpqs(n, MaxInFactorbase=800, SieveArrayLimit=50000, NumberOfPolynomials=2, Carray, Gauss); \\

1000000000039
```

The output could be any nontrivial factor, or FAIL if no one is found.
The next example is slightly larger, and distributing the polynomials will pay off here. However, computing a factor may need some minutes.

\[
\text{» } n := 2^8(2^7)+1;
\]

\[
34028236920938463463374607431768211457
\]

\[
\text{» mpqs}(n, \text{MaxInFactorbase}=2100, \text{SieveArrayLimit}=500000, \text{NumberOfPolynomials}=200, \text{Tolerance}=2.2, \text{Carray}, \text{NetModule});
\]

\[
59649589127497217
\]

3.4 Gcd over Algebraic Number Fields

In [11], Langemyr and McCallum propose a modular algorithm for computing the greatest common divisor of two polynomials \( f, g \in (\mathbb{Q}[x]/(r))[y], r \in \mathbb{Q}[x] \). The basic idea is to compute many gcd's of homomorphic images of \( f \) and \( g \) in residue class rings \( \mathbb{F}_p[x]/(\bar{r}) \), where \( \bar{r} \in \mathbb{F}_p[x] \) is obtained from \( r \) by reducing all coefficients modulo \( p \). (This requires replacing \( r \) with a polynomial over \( \mathbb{Z} \) first, as well as removing all denominators from \( f \) and \( g \).) Each time after processing a prime, the results are put together by means of the Chinese remainder algorithm.

Unlike in the case of the sieve algorithm (Section 3.3), it is not determined in advance how many \( p \) will be necessary (the theoretical upper bound is too conservative). The algorithm proceeds until a trial division of both inputs by the result succeeds; such trial division is attempted whenever considering a new prime does not change the previously obtained result.

The implementation uses a procedure \text{gcdmodp} which computes the modular gcd of \( f \) and \( g \) for a given input \( p \). The huge polynomials \( f \) and \( g \) as well as \( r \) are not passed as parameters, but stored in global variables, which means that they have to be broadcast over the net only once. An excerpt of the MuPAD source code of the Langemyr algorithm is listed in Figure 3.4 and Figure 3.5.

In order to create an example, we first have to generate polynomials over an algebraic number field. We use the field \( \mathbb{Q}(i) \) here\(^4\). Note that the parallel algorithm will not be faster than the sequential one for input polynomials that

\(^4\)Note that we have to specify the minimal polynomial of \( i \), because the identifier \( i \) has, by default, nothing to do with the imaginary unit (denoted by \( I \) in MuPAD).
are relatively prime (because this is likely to be seen from the first homomorphic image already).

As an optional third argument, a positive integer can be given; modular gcd's are computed modulo the next primes following that integer. The optimal choice for this parameter depends on the degree and coefficient size of the input and on the number of slaves available.

To demonstrate an example, we have to load the MuPAD algorithm first:
Computing Fibonacci Numbers in Parallel

This example demonstrates how to solve highly recursive problems using the programming paradigm of work groups. It computes Fibonacci numbers using their recursive definition, thus creating a binary tree to place intermediate results in it. Of course, the well-known iterative algorithm is much better. But the intention here is to present how to solve problems which cannot be reduced to effective iterative algorithms.

We begin with the initialization of the group master and its group slaves as well as the definition of the procedure fib which spawns two recursive group jobs and while waiting for their results successively processes other group jobs. To ensure that all group slaves are ready, a synchronization takes place at the end.

3.5 Computing Fibonacci Numbers in Parallel
Below, you can see how to use the procedure fib. To compute the Fibonacci number of 14, the procedure call fib(14) is passed to the group master. Once the return value is available, it is assigned to the variable r.

```plaintext
>> h := net::gjob(1, hold(fib)(14)):    # start first group job #
   while( net::gstatus(1,h)=0 ) do    # nothing #
      end_while:
      r := net::gresult(1, h);        # wait for result #
      # collect result #

610
```

Finally, we have to close the work group:

```plaintext
>> net::gmasterquit();
```

### 3.6 Karatsuba’s Method for Multiplying Polynomials

Using a well-known trick of Karatsuba [9], it is possible to reduce one polynomial multiplication to three multiplications of polynomials of half of the degree of the original polynomials (plus some additions). This is also an example for an algorithm that involves a lot of recursive calls. The implementation works as follows: all clusters form one work group, with the master being the group master, too. Whenever a polynomial is split by a process, two of the multiplications to be done become jobs of the work group (the third one is done by the process that did the split). While waiting for a result, every cluster (except for the
master) takes jobs. Since taking a task means redistributing smaller tasks and taking further ones while waiting for the result, the length of the polynomials in the problem list decreases quickly, while any cluster is waiting for a cascade of results. Polynomials of degree smaller than a certain limit are not distributed further, but handled by a sequential version of the Karatsuba algorithm. The code of the main program karazmult is listed in Figure 3.6 and the parallel multiplication algorithm karazuba itself reads as shown in Figure 3.7.

```maple
carazmult:=proc(f,g)
local i, res;
begin
for i from 2 to topology() do
    // hire some cheap workers
    writequeue("work", i, hold(net::gslaveinit(i)))
    // distribute the procedure 'karazuba' to all workers
    writequeue("work", i, hold(_assign)(hold(karazuba),karazuba));
    // display a message to the user
    userinfo(2, "Getting cluster ".expr2text(i)." to work");
end_for;
res:=karazuba([coeff(f,i) hold(i)=0 .. degree(f)],
    [coeff(g,i) hold(i)=0 .. degree(g)]);
poly( zip(res,[0 .. nops(res)-1],DOM_LIST), op(f,2 .. 3) );
end_proc:
```

Figure 3.6: Karatsuba’s Method for Multiplying Polynomials I

We used BOUND1 = 40 and BOUND2 = 400; depending on the configuration, different parameters may be optimal.

We first load the MuPAD algorithm. The algorithm can then be called as follows:

```maple
>> read("karazuba.mu");
```

```maple
>> f:=randpoly([x], Degree=2000, Terms=infinity):
g:=randpoly([x], Degree=2000, Terms=infinity):
karazmult(f,g); # this may need some minutes #
```

4000 3999 3998

poly(155840 x + (-269747) x + 170705 x + ...
karazuba:=proc(f, g)  
begin  
resqu:=genident("result");  
if nops(f)<BOUND1 or nops(g)<BOUND1 then  
[...]
else  
// parallel Karazuba  
n:=max(nops(f), nops(g)) \text{ div } 2;  
if nops(f)<=n then [...]
else  
a:=[op(f,n+1..nops(f))]; b:=[op(f,1..n)];  
c:=[op(g,n+1..nops(g))]; d:=[op(g,1..n)];  
aplusb:=zip(a,b,_plus,0); cplusd:=zip(c,d,_plus,0);  
// distribute the first two multiplications  
handle1:=net::gjob(1, hold(karazuba)(aplusb, cplusd));  
handle2:=net::gjob(1, hold(karazuba)(a, c));  
// Compute the third multiplication locally  
p3:=karazuba(b,d);  
// process further group jobs while waiting for both results  
while bool(net::gstatus(1,handle1)=0) do  
  if topology(Cluster)<>1 then net::gprocess() end_if;  
end_while;  
while bool(net::gstatus(1,handle2)=0) do  
  if topology(Cluster)<>1 then net::gprocess() end_if;  
end_while;  
// Fetch and use the results of the first two multiplications  
p1:=net::gresult(1, handle1); p2:=net::gresult(1, handle2);  
// Do some additional computation (carried out locally)  
[...]
end_if
end_if
end_proc:

Figure 3.7: Karatsuba's Method for Multiplying Polynomials II
Chapter 4

Programming Guidelines

Writing macro parallel programs is rather easy and allows quick implementation of algorithms. However, in this chapter we like to present some aspects to be taken into account when programming macro parallel MuPAD programs. Also refer to Section 6.1 to read about common errors in writing parallel algorithms.

4.1 Some Efficiency Considerations

Reconfiguring the Network

Although starting up and shutting down the network repeatedly at run-time is possible, it should rather be tolerated that the slaves remain idle during some sequential parts of an algorithm. The reason for this is, that starting up the network of MuPAD clusters is done by use of PVM [1] and the remote shell ssh\textsuperscript{1} which both need a significant amount of time.

If the remote shell rsh\textsuperscript{2} can be used within your local area network, you may want to use this instead of ssh because establishing connections is much faster (but less secure) with it. Refer to Section B.3 for detailed information.

Message Passing

Due to the network overhead of encoding, transferring and decoding objects, tasks that will take less than about 1-3sec (depending on your local configura-

\textsuperscript{1} ssh - secure shell client (remote login program), refer to the UNIX manual ssh(1).
\textsuperscript{2} rsh - remote shell, refer to the UNIX manual rsh(1).
tion) should not be distributed to be computed elsewhere in the network. It is often more efficient to compute them locally. Concerning message passing, also refer to the glossary.

Global Variables

Distributing and exchanging global information in a network is quite expensive. Therefore, write access to global variables should be used with great care unless the information is really needed by every cluster.

Another reason for low performance when using global variables extensively, is the currently used straight forward implementation (see Section 7.2). This aspect may be less relevant with future versions of macro parallelism.

Work Groups

Similar to tasks, discussed before in section Message Passing, also group jobs should not be too small. A run-time of only 1-3sec (depending on your local configuration) for a group job may lead to low performance, because of the network- and administration overhead which is needed.

When implementing highly recursive algorithms using the work group paradigm, each job should be instructed to process other group jobs (see net::gprocess) while it is waiting for results after using the function net::gjob to spawn new sub-tasks. This prevents possible dead locks\(^3\) and leads to faster processing the group's job queue. However, the program stack of the MuPAD kernel processes may grow rapidly. Refer to page 77 (Stack Overflow) for possible problems concerning memory limitations on your operating system.

Never Distribute Known Data

It is more efficient to make a cluster store an object in a variable than to send it several times. Especially library functions need not be sent to slaves because every cluster can access its (local) MuPAD library.

This is obvious, of course, but it can easily overlooked: since the functions writequeue, writepipe, and net::map evaluate their arguments, the function hold must be used when necessary. For example, the command net::map([2.0, 3.0], sin) makes the master distribute the whole source code of sin to the

\(^3\)Dead lock: all group slaves may wait on results and no one is working.
slaves, while \texttt{net::map([2.0, 3.0], \texttt{hold} (\texttt{sin}))} causes only the \texttt{identifier sin} to be distributed and evaluated by the slaves. Knowing this, it is easy to reduce network communication drastically.

### 4.2 Writing Good MuPAD Code

#### Naming Conventions

In MuPAD pipes and queue may the named using nearly any MuPAD object. However, it is a good style to use character strings (DOM\_STRING) as names. This makes it easier to read and understand the source code and cannot result it unintended evaluations.\footnote{When using the expression \texttt{1+1} as a name of a pipe or queue, the function \texttt{hold} must be used to prevent this expression from being simplified to \texttt{2} by the MuPAD evaluator.} Also refer to Section 6.1.

#### Length of Code Lines

The length of MuPAD code lines should not be too long. A length of 60-80 characters is suitable. Like all source level debuggers, also the MuPAD debugger is line-oriented, meaning each line performs a single debugger step. Putting too much code into one line makes debugging ugly because the steps may not be fine grained enough. Refer to Section 5.1 for detailed information about debugging.

### 4.3 Illegal Functions and Commands

When using macro parallelism, some MuPAD functions must not be used because they may conflict the integrity of the macro parallel network:

\texttt{reset}

Due to technical reasons, currently the MuPAD built-in function \texttt{reset} must not be used on the master of macro parallelism respectively sent to any cluster, while a macro parallel network is running. Otherwise, the network breaks down and must be halted with the shell script \texttt{netstop} (see Section B.4).
quit

Do not sent the MuPAD command quit to any slave cluster of a macro parallel network. The network may break down and then has to be halted with the shell script netstop (see Section B.4).

Use the function net::shutdown to terminate clusters. Currently it is not possible to add or remove hosts to/from a running macro parallel network.

traperror

Do not use the MuPAD command traperror with a second (time) argument on the master of macro parallelism, because it may effect the termination of the MuPAD kernel.

Use the command Pref::maxTime (refer to the MuPAD Online Manual) instead. In contrast to the function traperror, Pref::maxTime effects all following computations until it is reset explicit. It also does not guarantee real time handling.

4.4 Tips and Tricks

Printing Log Messages

Using the function net::mprint, slave clusters can display information on the master's console. In order to see what happens when a macro parallel algorithm is running, it is often also desired that any print and userinfo command executed by a slave cluster is displayed on the master's console. Figure 4.1 shows how this can be achieved by re-defining the kernel print method on all slave clusters of a macro parallel network.

Using Function fp::apply

Suppose that a slave is to execute a procedure f with input x, how can the task f(x) be sent such that both f and x are evaluated by the master, but the actual call to f is not? One way to do this is to use net::define to make f known to the slave first; another is to send the following task: hold(fp::apply)(f,x). This works also for more than one argument. If some (or even all) arguments are known to the slave, their evaluation can also be prevented by using hold:
netprint := proc()
local argv, i;
begin
    argv := [args()];
    if (argv[1] = Unquoted) then unassign(argv[1]); end_if;
    for i from 1 to nops(argv) do
        if (type(argv[i]) <> DOM_STRING) then argv[i] := expr2text(argv[i]);
        end_if;
    end_for;
    net::mprint("Cluster " . expr2text(topology(Cluster)).": "._concat(op(argv)));
end_proc;
net::compute(All, hold(sysassign)(hold(print), netprint));

Figure 4.1: How to Replace the Default print Method on Slaves

e.g. hold(fp::apply)(f, x, hold(y), z) is the task to use the local value of y and the sent values of f, x, and y to evaluate f(x,y,z).
Chapter 5

Additional Aspects

This chapter discusses some additional aspects of developing and writing macro parallel algorithms as well as using MuPAD as an open parallel problem solving environment for mathematical applications.

5.1 Debugging Parallel Algorithms

The current version of macro parallelism does not support debugging of parallel algorithms on slave clusters but only on the master of the network. Future versions will be improved to enable parallel debugging of a whole macro parallel network simultaneously. Also refer to Section 4.2.

For detailed information about debugging of MuPAD programs refer to the *MuPAD User’s Manual* [31], Chapter 3.

When using dynamic modules within parallel algorithms, also debugging of module functions on a C/C++ source code level may be useful. Refer to the *Dynamic Modules – User’s Manual and Programming Guide* [29], Section 8.7 and 10.1.7 for detailed information.

5.2 Profiling Parallel Algorithms

Since macro parallel algorithms are based on distributed programs running on sequential MuPAD kernels in a network, profiling on each kernel can be done as usual. Refer to the *MuPAD User’s Manual* [31] (see function profile) for detailed information.
Tools for merging the profile protocols from each cluster into one protocol describing the whole network are not provided at present.

5.3 Software Integration

The MuPAD concepts of domains\(^1\) (manual [31], paper [6]) and dynamic modules (manual [29]) allow users to integrate C/C++ algorithms and data structures as well as complete software packages as native elements of the MuPAD language.

Following the idea of hiding technical aspects from the user, it makes MuPAD to a *non-monolithic* CAS which performs an *open* (extensible) mathematical working environment. The concept of dynamic modules can also be used to integrate interprocess communication protocols (IPC) like MP [7][2] or *OpenMath*\(^2\) to interface special purpose systems via the network.

Many special purpose systems and algorithms written in C/C++ are available via the Internet from research groups all over the world (as well as from commercial companies) and can be used within parallel MuPAD algorithms. This includes user algorithms based on numeric libraries like the *NAG-C*\(^3\) or *IMSL CNL*\(^4\) library or arbitrary precision arithmetic packages like *GMP*\(^5\) or *NTL*\(^6\) and others.

Using existing software written by specialists, and thus utilizing their know-how, may drastically reduce the time and costs of developing algorithms for special applications. For additional information and examples read the *Dynamic Modules* user's manual (refer to http://www.mupad.de/BIB/ONLINE/SORGATZ98/).

As in sequential MuPAD programs, also within macro parallel programs it is often useful to integrate C/C++ algorithms for hotspots. On the other hand, macro parallelism can be used to schedule user algorithms written in C/C++ and implemented as dynamic modules. It supports the user in placing these tasks on various hosts of a heterogeneous computer network and in organizing distributed computations.

---

\(^1\) An approach of object-oriented programming in MuPAD. It enables users to define their own data types and structures and to write polymorphic algorithms very easily. Also refer to http://www.sciface.com/support/papers/.

\(^2\) *OpenMath*, refer to http://www.openmath.org/

\(^3\) The Numerical Algorithms Group Ltd., http://www.nag.co.uk/numeric/CL.html.


\(^5\) GNU MP, A Library for Arbitrary Precision Arithmetic.

5.4 A Math Problem Solving Environment

To solve so-called real world applications, meaning large physical systems and applications in industry, e.g. from robotics, it is often necessary to perform extremely fast symbolic as well as numerical computations. Moreover sometimes hybrid calculations are very useful, continually switching between numerical and symbolic computation.

General purpose computer algebra systems are very good in defining mathematical descriptions of these kind of problems, to transform them and also to display their solutions graphically. But alas, they are often not efficient enough to solve e.g. huge polynomial systems in a reasonable amount of time.

One approach to solve this discrepancy is to develop an open and parallel interactive general purpose CAS,

- which provides a convenient and object-oriented programming language (like Domains [6]) to handle complex (mathematical) data in an easy way and
- which is able to integrate efficient special purpose algorithms in an efficient and flexible way and to use them like native elements of the CAS programming language (like Dynamic Modules, see Section 5.3)
- in a parallel working environment for distributed computations in heterogeneous computer networks (like Macro Parallelism).

This was one reason for developing macro parallelism in MuPAD. Even today, MuPAD provides all features needed (see Figure 5.1) to be used as a flexible open and parallel problem solving environment for mathematical applications.

<table>
<thead>
<tr>
<th>Macro Parallelism</th>
<th>- distributed computing (network == computer)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- job placing and scheduling, information sharing</td>
</tr>
<tr>
<td>Dynamic Modules</td>
<td>- software integration: C/C++ algorithms</td>
</tr>
<tr>
<td></td>
<td>- C/C++ data structures as native language elements</td>
</tr>
<tr>
<td>Domains</td>
<td>- easy construction of mathematical data structures</td>
</tr>
<tr>
<td></td>
<td>- object-oriented and polymorphic programming</td>
</tr>
<tr>
<td>Object-oriented High-level Programming Language</td>
<td>Graphics Library</td>
</tr>
<tr>
<td></td>
<td>Graphical Source Code Debugger and Profiler</td>
</tr>
</tbody>
</table>

Figure 5.1: Components of the MuPAD Working Environment
Chapter 6

Trouble Shooting

This chapter lists some problems which may occur when installing and/or using macro parallelism in MuPAD and describes how to work around them.
Also refer to Chapter 4 for guidelines for macro parallel programming in MuPAD and to Section 5.1 for information concerning debugging parallel algorithms.

6.1 Common Errors in Writing Algorithms

The following user errors are likely to occur in implementations of macro parallel MuPAD programs. Also refer to Chapter 4 for programming guidelines.

Missing Definements

User-defined procedures and variables which are to be used by (all) slave clusters of a macro parallel network must be made explicitly known to all of them. This can either be done by distributing their definitions using the command \texttt{net::define} or by reading their definitions from a file locally on each cluster.

Unintended Evaluation by the Master

Since MuPAD functions like \texttt{writequeue} evaluate their arguments, the tasks to be distributed to slave clusters are executed by the master itself unless this is prevented by using the function \texttt{hold}. For example,

\texttt{writequeue("work", 5, (a:=7))}:
causes an assignment of 7 to the identifier a on the master. The slave cluster with the identification number 5 will only receive the integer 7 (which is the result of this assignment).

The correct way to send this instruction to a slave cluster would have been:

\[
\text{writequeue( "work", 5, hold((a:=7)) );}
\]

or if the assigned value, here 7, shall be evaluated locally.

\[
\text{writequeue( "work", 5, hold(_assign)(hold(a),7) );}
\]

**Confusion by Late Results**

Momentarily, it is not possible to reset/empty queues and pipes (including work queues) in remote or asynchronously. Therefore, a queue that has been used to collect results should not be re-used for another purpose until it is sure that all slaves have finished their tasks and will not write further results to it.

To synchronize algorithms within a macro parallel network, global variables as well as locks and semaphores can be used. Refer to Section 2.1.4 as well as Page 30 and Page 31 for additional information.

### 6.2 Problems Concerning Module net

**Macro Parallelism Hangs when Starting Up**

Before the module net can be used within a network, it must be made available to all network hosts which participate on macro parallelism. This includes its installation as described in appendix B.2 as well as its configuration which is described in appendix B.3.

Typical reasons for problems that occur when starting up macro parallelism are an undefined UNIX environment variable `MuPAD_ROOT_PATH` and the missing command line module `module(net)`: in the user's personal file `~/.mupadinit`.

**Problems With Using Dynamic Modules**

Refer to the *Dynamic Modules* manual [29], section 9 *Trouble Shooting* to read how to work around problems which may occur when using dynamic modules within MuPAD.
6.3 Problems Concerning PVM

The PVM library [1] is used as underlying network service. While running a macro parallel network, PVM starts daemon processes on each host which participates on macro parallelism in order to manage the MuPAD clusters. This section describes problems which concerns to this fact.

Problems after MuPAD was Aborted

When MuPAD was aborted instead of being terminated correctly, the PVM communication network might not be shutdown. In this case MuPAD is not able to start a new macro parallel network until all PVM daemons are terminated.

Use the shell script netstatus which is distributed with the dynamic module net to check the current status of PVM. It must be started on the master of macro parallelism or one of the hosts which participated on the macro parallel system. If PVM was terminated correctly, the following message is displayed:

```
tom> netstatus
There is no MuPAD Macroparallel System running on this host
```

Otherwise, the script displays a report about hosts and running processes:

```
tom> netstatus

Hosts of the existing MuPAD Macroparallel System
==================================================================
diophant
planck

Running jobs of the MuPAD Macroparallel System
==================================================================

HOST     JOB      ID
---------- ------ ----
diophant  -     40001
planck   mupad  80001
planck   mupad  80002
```

To terminate such a network in case of a fatal error, use the shell script netstop as shown below:

```
tom> netstop
The MuPAD Macroparallel System is stopped
```
Trouble Shooting

In some rare cases netstatus does not find any running PVM process, but MuPAD is still not able to restart macro parallelism (this may happen if the PVM master daemon has been vanished unexpectedly due to network problems or other reasons).

To "clean up the mess" the following two strategies can be used:

1. Call netstop host1 host2 .... to explicitly kill the PVM daemons and MuPAD clusters running on the specified hosts.

2. Call netstop -m. A MuPAD kernel is started and reads out the variable NETCONF, that is expected to be defined in your personal file ~/.mupadinit. Then netstop cluster is called for each of the listed clusters.

See Appendix B.4 for more detailed information about the shell script netstop.

6.4 Other Technical Problems

Problems Concerning MCODE-Encoding

MCODE is a MuPAD specific binary format for encoding MuPAD objects when writing them to files or transferring them over a network etc. The MCODE up to the ALPHA release of MuPAD 1.4.1 (Aug. 1998) could not be used within heterogeneous networks, because the representation of MuPAD objects depended on the architecture of the computer on which an object was encoded.\(^1\) This was fixed for later versions, where the MCODE can now also be used in heterogeneous environments.\(^2\) Also refer to Section Problems Concerning ASCII-Encoding.

Problems Concerning ASCII-Encoding

ASCII encoding of MuPAD objects, when transferring them in a network, is suitable for heterogeneous network environments. It has the disadvantage, that some objects have no ASCII representation in MuPAD and thus cannot be ASCII encoded. For example, funs and MuPAD Domains (DOM.DOMAIN) cannot be encoded in this way.

\(^1\)This concerns the size of the C/C++ data types long, float and double as well as the conflict between the little endian and big endian format.

\(^2\)The accompanying net module contains the corresponding MCODE patch.
To work around this problem, either use MCODE encoding or, if this is not possible due to a heterogeneous network environment, convert these objects into other MuPAD data types in order to pass them over the network. Also refer to Section Problems Concerning MCODE-Encoding.

**Stack Overflow**

Computing huge and highly recursive problems, e.g. when using *work groups* for such kinds of tasks, the operating system may abort the MuPAD program and display an error message like stack overflow.

This problem occurs when too few memory is available for the program stack or the program stack size is limited by the operating system.\(^3\)

Refer to the UNIX command `limit` to display and enlarge the current stack size. For example, on Linux 2.0 and Solaris 2.5 operating systems, this can be done as follows:

```plaintext
andi> limit
...
stacksize  8192 kbytes
...
andi> limit stacksize 32768 kbytes ; limit
...
stacksize  32768 kbytes
...
```

### 6.5 Questions and Answers

Starting the network fails. MuPAD hangs:

1. Check if the dynamic module `net` is installed on all hosts which shall participate on the macro parallel network. Refer to Appendix B.2 how to install it.

2. Check if the file `~/mupadinit` is available on these hosts and contains the command `module(net)` to load the dynamic module `net`.

\(^3\)On some UNIX systems the process stack size is limited to 8192 kbytes by default.
3. Check if ssh respectively rsh (see Section B.3) works properly by executing the command: ssh hostname pwd, which executes the pwd command on host hostname. If there are any problems executing this command, please ask your system administrator for help.

4. Quit MuPAD and execute the shell script netstatus to check if any network is still running. This can be stopped using the shell script netstop. See Section 6.3 and Appendix B.4 for more detailed information.

Some clusters cannot be allocated:
Starting up a macro parallel network, MuPAD may display the error message It was only possible to add xx hosts:
The reason for this problem may be a misspelled hostname, the fact that a host cannot be reached in the network or the fact that there is still a MuPAD cluster running on this host.
Use the shell script netstop host to explicitly kill this cluster. If this does not help, check the solutions described in Starting the network fails.

Transferring complex MuPAD data fails:
Refer to Section 6.4 to read about problems concerning MCODE-Encoding and ASCII-Encoding.

MuPAD was terminated by interrupting a computation using CTRL-C and then choosing quit or abort:
In this case the macro parallel network cannot be terminated correctly and restarting it will not work. Use the shell script netstop to shut down the network (see Section 6.3 and Appendix B.4 for details).

Can I interrupt a computation?
Unfortunately, the answer is no. Interrupting the master will not prevent the slaves from doing all of the jobs in their respective work queues.
Furthermore, using CTRL-C or the INTERRUPT button of MuPAD should be avoided, because it may conflict the integrity of the macro parallel network. On problems, refer to the solutions given above.

The log file /tmp/pvml.uid grows and grows:
The PVM system creates a log file named /tmp/pvml.uid and every output of PVM clients is logged it for debugging.
Unfortunately we cannot avoid this in the current version of the dynamic module net. Thus, in some rare cases, when network errors etc occur, the log file may grow rapidly.
Use the shell script `netstatus` to display the status of the current macro parallel network and examine the corresponding log file to search for the cause of the problem.

The output of `netstatus` contains *host ids*, like `t80001`, which can also be found in the log files, e.g.:

```
[t80040000] 07/21 14:31:14 [t80001] Starting slave of macro parallelism
```

and allow to determine corresponding hosts and processes.
Chapter 7
Design and Implementation

The current implementation of macro parallelism is based on the MuPAD release 1.4.1 for SunSPARC workstations running Solaris 2.5 and Intel PCs running Linux 2.0. However, there are no restrictions in using respectively porting it to other UNIX platforms.

Since the binary code of macro parallelism is relatively large (~400Kb), it was not statically integrated into the MuPAD kernel, but implemented as a so-called dynamic module (see Manual Dynamic Modules [29], Paper [27]) with the advantage of loading it at run-time on demand only and maintaining it independently from the MuPAD kernel. Figure 7.1 gives a first impression how this is realized.

![Figure 7.1: Layer Scheme of the Dynamic Module net](image)

Macro parallelism is now fully provided by the dynamic module net which contains the code for the master as well as for slaves. As network transportation layer the \textit{PVM} [1] system is used. This is a portable C library which is easy to use. It supports us in managing an arbitrary number of MuPAD clusters in a heterogeneous network and to send and receive network messages (see
Section 7.5). Figure 7.2 shows a simplified scheme of the internal design of the dynamic module net.

The main components of the dynamic module net are:

- **Interface**: provides the user interface to access the features of macro parallelism from within MuPAD.

- **InitMaster/InitSlave**: contains the code for initializing data structures and starting up a macro parallel network using PVM primitives.

- **Polling/Events**: contains event handlers, callbacks and other code which is needed to realize asynchronous actions like receiving messages, filling queues, updating global variables etc.

- **Pipes/Queues**: contains the code for managing all kind of pipes and queues available with macro parallelism.

- **Globals**: contains the code for managing network variables which are also used for network locks and semaphores.

- **Work Groups**: contains the code for managing work groups for automatic job scheduling in a (specified part of a) network.

- **MCODE/ASCII**: contains the code for encoding and decoding MuPAD objects before/after transferring them in a network using PVM primitives.

- **PVM**: the PVM library, used as transportation layer and to manage an arbitrary number of MuPAD clusters in a heterogeneous network.
7.1 How does it Work Inside

By use of the MuPAD command `module(net);`, the dynamic module `net` is dynamically linked to the MuPAD kernel at run-time. When it is loaded, the MuPAD kernel is called a cluster.

Starting up macro parallelism is originated by the first cluster, which is then called the master. For this, the module reads out the macro parallel network configuration (NETCONF) and instructs `PVM` to establish the network of clusters by starting remote kernels – the so-called slaves – on corresponding hosts of a heterogeneous computer network. Each cluster either executes the `InitMaster` or `InitSlave` procedure to install network event handlers, to share the network topology information, to initialize data structures for queues, pipes, global variables and work groups and so on. While the master – which is connected with a terminal – goes into interactive mode, slaves read the queue with the reserved name `work` to get their input.

All communication between clusters is channelled through the `PVM` networking interface. MuPAD expressions are encoded into MCODE messages (also ASCII encoding is supported, see below) which are sent as so-called `PVM` tagged messages. Reading incoming messages is done via a mixed polling and event driven strategy using the UNIX alarm timer and signals. After reading a `PVM` message, the embedded object is decoded and – according to the `PVM` message tag – inserted into a corresponding pipe or queue (including the work queue of slave clusters and the job queue of work groups) or assigned to a global variable.

The implementation of named queues and pipes is based on an internal data structure `queue` which is defined in the MuPAD kernel and is accessible within dynamic modules. Since names of queues and pipes may be any arbitrary MuPAD object, hash tables (the MuPAD data type `DOM_TABLE`) are used to associate a name with its corresponding internal queue.

With the current version of the module `net`, MuPAD objects can be transferred either ASCII encoded or binary encoded using a MuPAD specific binary format called MCODE (used by default). The user can switch the encoding because in older MuPAD versions (up to the ALPHA release of MuPAD 1.4.1, Aug.'98) MCODE could not be used in a heterogeneous environment. On the other hand, MuPAD objects like `funs` have no text representation in MuPAD and thus cannot directly be sent ASCII encoded. Having this in mind, also older versions of MuPAD macro parallelism can be used in heterogeneous networks of Sun-SPARC workstations running Solaris and Intel PCs running Linux. The new MCODE version is architecture independent to avoid these problems. Also refer to Section 6.4.
7.2 Concerning Global Variables

Global variables are represented as entries of a hash table (DOM_TABLE). The current values of all global variables are held on each cluster, so that reading can be done locally, which is very fast. Writing to a global variable is realized by a network broadcast using PVM messages, which updates the corresponding hash table entry on each cluster of the network.

It is guaranteed that assigning a value to a global variable is an atomic operation, in the sense that it cannot be spoiled by any other write access. More precisely, it is a blocking operation which returns after all clusters have updated their local value and which prevents any other write access to the same global variable requested elsewhere from execution. Thus, concurrent calls like global(v,global(v)+1) are safe operations.

The communication protocol for global variables requires great care: network communication must be reduced to a minimum and global writes must not block any other operation but global writes to the same variable. Currently, the master is used as permit server for write permissions to globals: it notes all write requests in queues, processing them using the strategy first come first serve. While a slave waits for a permit, it processes incoming messages and may write to other global variables to avoid deadlocks. Once the permit has arrived, the slave broadcasts the new value and waits for corresponding acknowledgements. After all acknowledgements arrived, the permit is given back to the master.

We chose this straightforward strategy to realize global variables because we did not want to spend our time adapting or re-implementing algorithms of current research. We primarily wanted to gain first experiences with the macro parallelism and write parallel algorithms at a very early state of the development. But even this rather naive approach turned out to require a lot of fine-tuning and little variations of the communication protocol had drastic consequences. The performance is reasonable, if globals are used deliberately. But globals may become a terrible bottle-neck when used extensively. More efficient solutions are needed for the future.

7.3 Concerning Locks and Semaphores

Network locks and semaphores can be seen as special network variables with exclusive write access (atomic operation, see Section 7.2). Thus in macro parallelism, they were implemented directly based on global variables. The source


Concerning Work Groups

The code just consists of a user interface written in the MuPAD programming language, implemented as so-called module procedures (see Dynamic Module [29], Section 7.2.1). Refer to the source file net.c for additional information.

7.4 Concerning Work Groups

Work groups are implemented as an extension to the processing of the work queues. The group master holds a table (DOM_TABLE) of all active group jobs between spawning and result fetching, an internal queue for the numbers of all waiting jobs and a set (DOM_SET) of its currently idle group slaves.

The group slaves basically hold the information that they are slaves of a specific group master and the last working status information that has been transferred to the group master. Whenever the status of a group slave changes, this is reported to the group master.

New jobs are distributed to the group slaves for processing in two cases: First, if a new job is added to the table of active jobs and there are idle clusters available, second, if the group master receives a message that a group slave has become idle and there are waiting jobs. Only if the work queue of a group slave has become empty, the group slave reports that its status has changed to idle. Therefore the priority of work queue jobs is higher than that of group jobs in general.

This approach was again designed to reduce communication inbetween work groups to a minimum and to make the ties between group members as weak as possible. Thus, reconfiguring a work group (assigning new group slaves to it or deleting old ones from it) requires very little overhead. Our primary intention here was to offer an easy, powerful and fully automatic job scheduling mechanism.

The guideline that jobs (tasks), distributed in the macro parallel network, should not be too small (see Section 4.1) also applies here. Note, that a job may have to be transferred twice before it is evaluated: first to the group master, and second to the group worker.

7.5 Concerning the PVM Library

Macro parallelism is currently based on PVM [1] which is used as transportation layer and for the process management. Due to network security reasons, the
UNIX program ssh is used for spawning PVM remote tasks. To use PVM in the context of a dynamic module, some slight modifications to the standard installation of PVM are necessary. Refer to Page 87 for details.

To hide PVM to the normal user of macro parallelism, we utilize the possibility of a global configuration file to specify the PVM run-time environment and to tell all PVM daemons where MuPAD is installed. With this, normal users have the view to a macro parallel MuPAD system without being tangled with other software components. This run-time environment is delivered and installed with the module net. However, users with knowledge of PVM can surely use their own configuration file and thus can utilize all extended configuration features PVM offers. No limitations are made by the MuPAD net module.


7.6 Technical Information and Discussion

In this section details about the development and implementation of macro parallelism, in form of the dynamic module net, are discussed. This is intended as an introduction for those users who want to port the module to other operating system platforms as well as for those who want to extend the implementation with new features.

Extensions and improvements of the implementation discussed here can be sent to the authors in order to be included into the next official release of macro parallelism.1

7.6.1 Technical Concepts and Details

7.6.1.1 Using the Concept of Dynamic Modules

By default, dynamic modules are handled as temporary machine code resources of the MuPAD kernel, with the advantage to link and unlink them into respectively from the kernel on demand, automatically and transparently to the user. For details refer to the Dynamic Modules Manual [29].

1Use one of the email addresses {andi,bugs,tom}@mupad.de
Displacing cannot be used for module code which contains interrupt handlers and/or stores a status like an open file or socket handle, as it is needed by the macro parallelism. For these cases a dynamic module can be declared as static respectively as temporary static (this can be changed at run-time), meaning its machine code cannot be unlinked automatically by the displacing strategies of the MuPAD module manager.

At present the dynamic module net is declared as permanent static, which means that it can not be unloaded within the MuPAD session. For this, the module source code is compiled with the module generator mmg using the option -a static respectively by defining the module function net::initmod as static using the flag MCstatic (refer to the source file net.C).

### 7.6.1.2 Using the PVM Library

For the current implementation of the module net, the PVM [1] (see Section 7.5) version 3.4beta6 is used. The source code of PVM is available via the World-Wide-Web at: [http://www.netlib.org/pvm3/index.html](http://www.netlib.org/pvm3/index.html) as well as on the accompanying CD-ROM in the directory /cdrom/demo/NET/pvm3/.

To use PVM in the context of a dynamic module, it was compiled with the GNU g++ compiler using option -fpic to create position independent object code:

- g++ 2.7.2 or later must be used to compile PVM.
- the g++ compiler option -fpic must be used to compile PVM to position independent code.
- g++ 2.7.2 or later must be installed when creating the dynamic module binary using the MuPAD module generator mmg.

For this, the PVM file pvm3/Makefile.aimk has to be changed as listed below:

```plaintext
CC = gcc
CFLOPTS = -fpic
PVMOPTS = CC=$(CC)
F77 = $(F77)
CFLOPTS = $(CFLOPTS)
```

Due to network security reasons, the UNIX program ssh\(^2\) instead of rsh\(^3\) (default) is used for spawning PVM remote tasks in a network. ssh is searched for

---

\(^2\)ssh - secure shell client (remote login program), refer to the UNIX manual ssh(1).

\(^3\)rsh - remote shell, refer to the UNIX manual rsh(1).
according to the user's UNIX environment variable PATH. For this, the corresponding PVM configuration file pvm3/conf/$PVM.ARCH.def has to be changed as listed below:

```
-DRSHCOMMAND="ssh"
```

Note: PVM searches executables in the directories $PVM_ROOT/bin/$PVM.ARCH and $HOME/pvm3/bin/$PVM.ARCH as well as in the directories specified by the environment variable PVM_PATH. This variable must have been set before the PVM system is started.

### 7.6.1.3 Starting up the Network

A macro parallel network can be started both by command line option when a MuPAD session is started and by executing the function net::master within a running MuPAD session. To start it by command line option, the user's file `~/.mupadinit` must include the network configuration defined in the MuPAD variable NETCONF and the MuPAD command module(net): for loading the dynamic module net and starting up the network.

When a dynamic module is loaded, its module function initmod is executed automatically if available [29]. Here, this function is used to test if the user option '-U master' or '-U slave' was given in the command line and then handle these cases accordingly:

<table>
<thead>
<tr>
<th>Master Code</th>
<th>Slave Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize, configure and start up the PVM system library</td>
<td>Install signal handler and callbacks for processing PVM messages</td>
</tr>
<tr>
<td>Install the alarm timer for SIGALRM</td>
<td>Wait for topology information</td>
</tr>
<tr>
<td>Read configuration and start clusters</td>
<td>Still wait on the information</td>
</tr>
<tr>
<td>Distribute the topology information</td>
<td>Read the topology information</td>
</tr>
<tr>
<td>Change into the interactive mode</td>
<td>Start work queue polling loop</td>
</tr>
<tr>
<td>Repeat</td>
<td>Repeat</td>
</tr>
<tr>
<td>Read user input from console</td>
<td>Read job from work queue</td>
</tr>
<tr>
<td>Evaluate user input</td>
<td>Evaluate job</td>
</tr>
<tr>
<td>Until shutdown or halt</td>
<td>Until shutdown or halt</td>
</tr>
<tr>
<td>If halt then terminate the kernel instantly</td>
<td></td>
</tr>
<tr>
<td>Simulate network of topology 1</td>
<td>Terminate the kernel instantly</td>
</tr>
</tbody>
</table>

Table 7.1: Starting up and Running the Network
7.6.1.4 Network Variables

The master maintains a table (MAPV.GLockTab) in which all network variables are referenced whose permit is currently used and which clusters are currently waiting for it. The communication protocol for global writes is as follows:

If a slave wants to issue a global write, it sends a permit request to the master as a MAPT:DPERM message with name of the global variable. If the permit for the requested variable is already in use, the master stores the request in an internal queue. If the master has the permit for the requested variable, it marks the permit as used and passes it to the slave with a MAPT:PERMACK message. If the slave receives this message, it broadcasts the new global value with a MAPT:DWGLOBAL message to all other clusters and waits for their acknowledgement messages of type MAPT:DWGLOBALACK. The number of incoming messages is counted within the MAPcatchEvents function. When all acknowledgements are received, the permit is sent back to the master with a message of type MAPT:PERMACK including the name of the variable and the slave continues to execute the next commands within its work queue.

If the master receives the MAPT:PERMACK message and there is at least one cluster waiting for the permit of the associated variable, the permit is immediately passed on to the next waiting cluster via MAPT:DPERM message. But if there are no clusters waiting for the permit of the incoming MAPT:PERMACK message, the permit is marked as free. If the master wants to issue a global write, it checks in its local table (MAPV.GLockTab) if the permit is present. If the permit is already in use, it stores its own request in the variable’s permit queue. But if the permit is free, the master marks it as used, broadcasts the new value and waits for acknowledgements absolutely similar to the slaves. Then the master marks the permit as free. Both the master and the slaves process all incoming messages at all times if they wait for a certain message to avoid deadlocks.

The global writes are atomic only in the sense that all global writes to one variable are sequentialized, but there is a time interval in which the distribution of new values takes place and all clusters operate as usual except that no other global write on this variable can be initiated. The local value of the global variable on each cluster can be the old one or the new one in this special time interval, but which one you get is not defined. Therefore it can happen that two different clusters get different return values when they read the value of one global value at exactly the same time. However, after the global write is completed, i.e. the global function returns, all clusters do have the new global value.
7.6.1.5 Work Groups

Each cluster of the macro parallelism contains the information whether it is a group master, a group slave or are not assigned to any work group. This is stored in the variable MAPV.GroupMember. For group members, the variable MAPV.GroupMaster specifies the group master.

Each work group master maintains three additional data structures. First, there is a table of all active group jobs (MAPV.GroupJobTable) with entries for the job itself, its number, status and result. Second, there is a queue for the numbers of all waiting group jobs (MAPV.GroupJobQueue). Third, there is a set (MAPV.GroupIdleSet) of all group slaves that are idle.

Each work group slave maintains the additional variable MAPV.GroupIdle which contains the last working status information that has been passed to its work group master. When a group slave detects (while polling its work queue) that its status has changed, this is reported to the group master via MAPT:DGBUSY or MAPT:DGIDLE messages. There are two subtypes of MAPT:DGIDLE messages, the first with name field set to zero is the silent type, and the second with name field set to one is the verbose type. The work queue polling algorithm issues the silent type only (see below).

New work group jobs can be created by the net:gjob function. If the job is created on the master itself, no network communication is needed for simply entering the new job in the job queue (local case). But if the job is created elsewhere (remote case), the new job is sent to the group master embedded in a MAPT:DGJORD message. The group master then returns the new job’s handle number embedded in a MAPT:DGJORDACK message.

The remote fetching of a group job’s status via net:gstatus is done through MAPT:DGJSTS and MAPT:DGJSTSACK messages, the remote fetching of a finished job’s result via net:gresu1t is done by messages of type MAPT:DGJRES and MAPT:DGJRESACK similarly to the job creation via net:gjob.

Whenever a group master receives a new job from a remote cluster or spawns it itself, it creates a new job record for it and then looks if there is an idle cluster available (by a lookup in its idle cluster set). If there is an idle cluster, it immediately sends the new job to it via MAPT:DGJOB message. If there is no idle cluster, it stores the new job’s handle number in the waiting job queue.

Whenever a group master receives a (silent or verbose) MAPT:DGIDLE message from a group slave, it checks if there are waiting jobs. If there is a waiting job, it immediately sends the next job to the idle group slave via MAPT:DGJOB message, and the processing of this idle notification message is completed. If there
are no waiting jobs, the idle group slave's cluster number is stored in the idle cluster set, but then the distinction between silent and verbose MAPT: :DGIDLE messages becomes significant: if and only if the idle notification was verbose, a MAPT: :DGNOJOB message is returned to the idle slave.

Calls to net: : gprocess generate verbose MAPT: :DGIDLE messages: these calls are blocking until the group master has either sent back a new job or a notification that there is no other job to process.

7.6.1.6 Error Handling

For obvious reasons, we wanted to implement an elementary error handling for macro parallel algorithms in MuPAD at least. Errors on the master do not pose a problem, they are echoed as usual. Since the slaves are not interactive and normally do not have any visible output, their errors must be indicated to the master and/or to the cluster which originated the job which failed.

At the moment errors on slaves are handled as follows: whenever an error occurs, the slave sends a message of type MAPT: :DERRORINFO with the error number to the master. As soon as it processes the incoming message, the master displays a corresponding error message on its console. This can be prevented by the user by altering the preference LogError with the function net: :pref.

Extending the error handling in the sense that also the originators of failed jobs are notified will be fairly easy but is not yet implemented.

7.6.1.7 Pseudo Asynchronous Event Handling

At present, fully asynchronous event handling is not possible due to the fact that the current version of the MAMMUT [20] memory manager used by MuPAD 1.4.1 is not reentrant. This means that internal data structures of MuPAD, like queues and tables (DOM_TABLE), can be accessed in event handlers only with serious limitations. Thus, since the MuPAD kernel is not fully reentrant and thread-safe, the following pseudo asynchronous way of handling events is used:

All events are encoded as PVM messages, processed by an internal callback routine bound to an interrupt handler which is linked to the SIGALRM signal. The callback routine is called periodically from a central and safe code segment within the evaluator of MuPAD and enables processing incoming PVM messages quasi in background mode when clusters are busy. On the other hand, the callback routine can be invoked directly by the alarm handler, when this knows for sure, that the memory manager can be called without causing conflicts.
When clusters are idle, processing of incoming messages is explicit enforced: On a slave, the message processing routine is called whenever its work queue is empty and it is waiting for a new task to evaluate. On the master, which does not use the work queue but is connected to a terminal, being idle means that the kernel goes into the interactive mode of the line editor and waits for user input. This is known as a safety condition and enables the alarm handler to directly invoke the message processing routine. Finally the alarm timer causes the interrupt handler to be executed periodically.

### 7.6.2 About Module Source Code and Development

#### 7.6.2.1 Modular Division of the Source Code

So far the implementation of macro parallelism as the dynamic module net consists of three separated layers:

1. The lowest layer implements the access to \textit{PVM} functions and handles the conversion of MuPAD objects into/from \textit{PVM} messages. This is done in the source files MAP_iostream.[CH] and MAP_net.[CH].

2. The second and middle layer implements the functionality of queues, pipes, net variables and work groups as well as routines for processing incoming messages. This code can be found in the files MAP_err.[CH] and MAP_mup.[CH].

3. The third and top layer exports the internal functionality to the dynamic module user interface. This is done in file net.[CH].

Table 7.2 describes the contents of each C/C++ source code file in brief:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP_err.[CH]</td>
<td>Code for handling MuPAD kernel error codes</td>
</tr>
<tr>
<td>MAP_iostream.[CH]</td>
<td>Byte stream interface for using the MCODE</td>
</tr>
<tr>
<td>MAP_locks.[Hs]</td>
<td>Locks for mutual exclusion with signal handlers</td>
</tr>
<tr>
<td>MAP_mup.[CH]</td>
<td>Internal functionality above the \textit{PVM} level</td>
</tr>
<tr>
<td>MAP_net.[CH]</td>
<td>Layer for \textit{PVM} access and routines for encoding and decoding MuPAD objects</td>
</tr>
<tr>
<td>net.[CH]</td>
<td>Dynamic module: user interface functions</td>
</tr>
</tbody>
</table>

Table 7.2: Source Files of Module net
7.6.2.2 Compiling the Dynamic Module

The C/C++ source code of the dynamic module net is available with the accompanying CD-ROM in directory demo/NET/. To compile it:

1. MuPAD must be installed and configured. See Appendix A.2 and A.2.1.

2. the directory demo/NET must be copied to the hard disk, because temporary files and the binaries will be stored at this location.

3. the PVM library must be installed and compiled according to the hints given on Section 7.6.1.2. It is expected to be placed in demo/NET/pvm3 (this is available on the CD-ROM, compiled for SunSPARC Solaris 2.5 and PC Linux 2.0 (for libc.so.5 versions).

The following environment variables must be configured in file ~/.cshrc when not using the PVM version distributed with this CD-ROM:

setenv PVM_ROOT /your_pvm_directory
setenv PVM_ARCH `$(PVM_ROOT)/lib/pvmgetarch`
setenv MANPATH $(PVM_ROOT)/man
set path=($path $(PVM_ROOT)/lib $(PVM_ROOT)/bin/$(PVM_ARCH))

4. the g++ compiler version 2.7.2 or later must be installed.

5. also refer demo/NET/Makefile and demo/NET/Makefile.net

6. execute make to create the module binary net.mdm for the current operating system as well as the MuPAD code file net.mdg.

Refer to the Dynamic Modules Manual [29] for detailed information about compiling dynamic modules.

7. exchange both file in your MuPAD installation distributed with this CD-ROM in directory $MuPAD_ROOT_PATH/`sysinfo`/modules.

7.6.2.3 Functionality and Performance Tests

The files listed in Table 7.3 are available with the accompanying CD-ROM in directory demo/NET/ and can be used to check the functionality as well as the performance of the dynamic module net. Also refer to the mathematical benchmarks listed in Chapter 8. For information about debugging the module refer to Section 7.6.2.4.
Table 7.3: Files of the Test Suite

<table>
<thead>
<tr>
<th>Filename</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfGlobals.mu</td>
<td>Test performance of global variables</td>
</tr>
<tr>
<td>testErrors.mu</td>
<td>Test report of remote errors on master</td>
</tr>
<tr>
<td>testGarbage.mu</td>
<td>Test garbage collector on slaves</td>
</tr>
<tr>
<td>testGlobals.mu</td>
<td>Test (concurrent) writing global variables</td>
</tr>
<tr>
<td>testGlobals2.mu</td>
<td>Test atomicity of writing global variables</td>
</tr>
<tr>
<td>testGroupProcess.mu</td>
<td>Test net::gprocess function</td>
</tr>
<tr>
<td>testKarazuba.mu</td>
<td>Karatsuba algorithm, uses whole function set</td>
</tr>
<tr>
<td>testMap.mu</td>
<td>Test net::map function</td>
</tr>
<tr>
<td>testMisc.mu</td>
<td>Test miscellaneous functions</td>
</tr>
<tr>
<td>testMprint.mu</td>
<td>Test net::mprint function</td>
</tr>
<tr>
<td>testPipes.mu</td>
<td>Test net::readpipe and net::writepipe</td>
</tr>
<tr>
<td>testPref.mu</td>
<td>Test net::pref function</td>
</tr>
<tr>
<td>testQueues.mu</td>
<td>Test net::readqueue and net::writequeue</td>
</tr>
<tr>
<td>testStandalone.mu</td>
<td>Test network with a master without slaves</td>
</tr>
<tr>
<td>testTopology.mu</td>
<td>Test net::topology</td>
</tr>
<tr>
<td>testWorkGroups.mu</td>
<td>Test work group functions 1</td>
</tr>
<tr>
<td>testWorkGroups2.mu</td>
<td>Test work group functions 2</td>
</tr>
<tr>
<td>testWorkGroups3.mu</td>
<td>Test work group functions 3</td>
</tr>
</tbody>
</table>

Note, that when starting the macro parallel network most time is needed by ssh and PVM to start the remote PVM daemons and PVM clients (MuPAD kernels).

Note also, that the communication time (and thus performance) is influenced by the time needed for encoding and decoding MuPAD objects (relative high), the time for transferring the encoded data in a physical network (small) and the time needed for processing the network messages pseudo asynchronously (low up to high: depends on the concrete application, the current processor load on the local system and the time intervals configured for the alarm timer).

7.6.2.4 Debugging the Dynamic Module

To debug the dynamic module net, this must be compiled with debug information first. For this, change the file demo/NET/Makefile as follows:

```
MMGO = -g# # Compile with debug information
```

For additional information about debugging dynamic modules, please refer to the Dynamic Modules Manual [29].
It is recommended to compile the module setting the definition MAPNETDEBUG which instructs the net module to print some protocol messages at run-time.

Furthermore, the following lines should be inserted into the user's personal file \texttt{~/.mupadinit}, before starting MuPAD and loading the net module. Adapt the example below according to your local configuration:

\begin{verbatim}
NETSRCPATH:= "/user/cube/MuPAD/demo/NET/": #path of source code#
NETLOGPATH:= NETSRCPATH."/debug.".module(slave,exec)("hostname"): # create a log file#
NETDISPLAY:= "wiwianka:0.0": # console of master #
protocol(NETLOGPATH);
\end{verbatim}

The NETSRCPATH variable specifies where the source and object files of the module net are located for later use with the C/C++ source code debugger xgdb.

NETLOGPATH specifies the name of the files to which the output of each cluster will be written. Here, the directory of the source files is used, where log files with the names debug.	exttt{hostname} are created.

NETDISPLAY specifies the X11 display (typically the master's console) which is used when xgdb is invoked.

The function protocol instructs MuPAD to log all output in a log file. It is described in detail in the *MuPAD User's Manual* [31].

### 7.6.3 Open Problems and Missing Features

Apart from the fact that the current implementation is usable and relatively stable and can be used properly for many applications, a few technical problems remain to be solved:

**Ctrl-C:** It may cause many problems to stop the master via Ctrl-C. First, if you choose to quit after Ctrl-C, the macro parallel network will not be shut down correctly because currently the dynamic module net cannot trace these kinds of interruptions.

Thus you must use the shell scripts netstatus and netstop (delivered with the dynamic module net, see appendix B.4) to shut down the *PVM* system explicit.

Furthermore, if a write access to a global variables is interrupted and aborted via Ctrl-C there is a good chance that some internal switches are not reset. This means that globals cannot be used any longer and
everytime you try to write to a net variable you get the error Illegal cascaded write global. The only thing you can do then is to shut down the system completely and restart it.

A way out: either to mask the Ctrl-C interrupt during critical sections or to trace them all. However, the Ctrl-C interrupt is meant as an emergency stop in MuPAD and should not be inhibited.

Irregular termination of the system: If the system terminates irregular it is possible that there are still active MuPAD processes and PVM daemons on some hosts. The same can happen if there are network problems so that the master can't reach a slave.

No way out: this is a general problem of computer networks.

PVM confusion: In some rare cases it may happen that PVM becomes confused and PVM daemons cannot be killed completely by use of the shell script netstop (appendix B.4).

In this extremely rare cases one has to login on all hosts which participated on the macro parallel network and must to remove remaining temporary PVM files in the directory /tmp.

No way out: since PVM may be aborted via an emergency stop.

Global variables: As discussed in Section 7.2, due to the currently used straight forward strategy, the write access to network variables is not as efficient as desired.

A way out: sometimes global-messages which tell the clusters to update a global variable are not processed quickly enough. Even the use of signals and/or alarm timers does not solve this problem properly. More intelligent caching strategies and network protocols are needed.

Reset: Neither a specific set of clusters nor the complete macro parallel network can be reset yet. One has to shut down and restart the network for this purpose. Even worse, it is not possible to reset just the master.

A way out: the current implementation of macro parallelism does not interfere too deep into the internals of the MuPAD kernel, but this is needed for resetting. Future versions of the MuPAD kernel will provide special interfaces for this.

Only one network per user: A host can only be participate in one macro parallel network.

No way out: this is a default feature of PVM.
xmupad: Using macro parallelism within xmupad one must use the quit; command instead of the Quit button to leave the MuPAD system. Otherwise, the macro parallel is not shut down correctly and one has to use the shell script netstop (see appendix B.4) to stop the PVM system explicit.

A way out: There is a bug in the session frontend of xmupad and will be fixed in a future version.

Unwanted PVM log files: These log files are created by PVM in the directory /temp of each host which participates on a macro parallel network. It contains error messages and output of PVM clients and may grow rapidly when network delays and/or errors occur.

A way out: We have to patch the source code to make log files run-time configurable.

7.6.4 Desired Features and Future Extensions

The experiences in using the current version of macro parallelism in MuPAD 1.4.1 show that it would be useful to extend the user interface to make programming of parallel algorithms even more convenient and flexible.

Main points are to make macro parallelism more error tolerant and to support tracing and debugging of macro parallel algorithm in a more convenient way as well as to provide information and strategies which can be used for load balancing in a heterogeneous network:

Status information: User should have the possibility to check the status of workers and jobs in a macro parallel network (is the worker alive, how many tasks are in its work queue, how much memory is available and how high is its processor load, ..., is the job still in the work queue or is it busy, idle or ready, ...). This can be used for an error handling concept as well as for load balancing. Currently, this feature is only realized in part with the work group concept.

Cleaning queues and pipes: It should be possible to clean queues and pipes remotely. This is especially needed for an error handling concept but is also very useful when implementing branch-and-bound algorithms etc.

Redirecting output of the slaves: It should be possible to redirect the output of the slaves to the master in a more convenient way. Currently, this is only supported with the module function net::mprint and the user is responsible for the details. Also refer to Section 4.4.
Debugging and tracing features: Users should be enabled to debug a whole macro parallel network to check the control flow if necessary. Since MuPAD provides a graphical source code debugger interface to the MuPAD language, this can be realized by running each cluster of the network under control of the MuPAD debugger in the future. Currently, debugging is only possible on the master of the macro parallel network. Also refer to Section 5.1.

Broadcast and Multicast: Distributing and collecting information in a macro parallel network should be more convenient. Currently, only the functions net::compute and net::define are provided.

Priorities of jobs: The job queues of work groups should be realized as priority queues with user-defined weight functions. This would enable users to experiment with more intelligent strategies than the currently used first-in-first-out method. For example, for Karatsuba's method for multiplying polynomials, jobs can be ordered according to the degree of the polynomials which have to processed (shortest-job-first) or processing times may be estimated with user-defined algorithms.

Asynchronous work: Often the following problem occurs: a cluster has distributed some tasks to other clusters; it cannot continue its work before having received the results. Instead of a readqueue(..., Block), a cluster should be allowed to do some useful work until the results are available.
Chapter 8

Mathematical Benchmarks

The algorithms described in Chapter 3 were used for benchmarks and performance tests for the implementation of macro parallelism as well as to verify the usability of the concept of a math problem solving environment as introduced in Section 5.4.

The results which were achieved with this first prototype are described in brief in the following sections. They cannot really be denoted as high-performance computing at this state, but they demonstrate that the key technologies which are needed for high-performance computing and to solve so-called real world applications, are available with domains, dynamic modules and macro parallelism in MuPAD and also show that this is a very promising way of developing interactive general purpose computer algebra system in the future.

8.1 Gcd over Algebraic Number Fields

The algorithm used for this benchmark is described in section 3.4. For our tests, we always used primes $\approx 10^{20}$.

The algorithm showed nearly full speed up, compared to a sequential version. We used a network of one SPARC-20 (as a master), one UltraSparc-II and three UltraSparc-I to compute the gcd of two polynomials of the form $f*h$ and $g*h$, where $f, g, h \in \mathbb{Q}(i)[x]$ were dense random polynomials, with the real and imaginary part of each coefficient being integers of absolute value smaller than $10^{12}$. The real times for the gcd computation (depending on $\deg f = \deg g$ and $\deg h$) are listed in the third column of Table 8.1 below. The running time of a sequential version of Langemyr's algorithm (on a Ultra-I) is given in the rightmost column.
Table 8.1: Times Used by the Gcd Algorithm Measured in Seconds

<table>
<thead>
<tr>
<th>$deg f$</th>
<th>$deg h$</th>
<th>langemyr</th>
<th>seq. langemyr</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>61</td>
<td>66</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>50</td>
<td>10</td>
<td>37</td>
<td>92</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>97</td>
<td>239</td>
</tr>
<tr>
<td>200</td>
<td>10</td>
<td>309</td>
<td>833</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>53</td>
<td>74</td>
</tr>
</tbody>
</table>

Table 8.1 shows that the speed up greatly depends on the input: if the input polynomials are relatively prime (refer to the first row), any random modular images of them are likely to be relatively prime, too, such that the running time is determined by the fastest slave. The speed up gets better when more modular images are needed.

8.2 Karatsuba’s Method for Multiplying Polynomials

The algorithm used for this benchmark is described in Section 3.6. Since this macro parallel algorithm was chiefly written to test the programming paradigm of work groups, we could not expect it to be very fast.

Using the same configuration as in Section 8.1 (three UltraSparcI, one Ultra-II, and a SPARC-20 as master), two dense univariate integral of given degree could be multiplied slightly faster than by a sequential version of the same algorithm on an Ultra-I.

As expected, the classical multiplication _mult of the MuPAD kernel (on an Ultra-I) was much slower. Refer to the Table 8.2 below for details.

<table>
<thead>
<tr>
<th>degree</th>
<th>karazmult</th>
<th>seq. karazmult</th>
<th>_mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>5</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>700</td>
<td>8</td>
<td>10</td>
<td>29</td>
</tr>
<tr>
<td>1000</td>
<td>13</td>
<td>18</td>
<td>62</td>
</tr>
<tr>
<td>1500</td>
<td>21</td>
<td>34</td>
<td>138</td>
</tr>
<tr>
<td>2000</td>
<td>54</td>
<td>55</td>
<td>246</td>
</tr>
<tr>
<td>3000</td>
<td>79</td>
<td>103</td>
<td>565</td>
</tr>
</tbody>
</table>

Table 8.2: Times Used by karazuba Measured in Seconds
Once the implementation of priority queues for scheduling jobs in a work group is available, it will be interesting to study the speedup that is gained by preferring easy tasks (multiplications of low-degree polynomials) rather than early generated ones.

8.3 Factoring Integers by the Elliptic Curve Method

The algorithm used for this benchmark is described in Section 3.2.

Figure 8.1 displays the running times of the algorithm until the result is returned because one slave succeeded. Since the master currently cannot instruct slaves to abort their current computations immediately, it usually takes a few seconds in addition until all slaves are ready.

In this example, we used a network with a slow master and seven fast slaves (five Ultra Sparc I, one Ultra Sparc II, one Ultra Sparc 5). The running times are marked by gray points. The parameters used were timeout = 30sec and increment = 1sec.

In addition, running times for the sequential ifactor executed on a single Ultra Sparc 5 are given (by black points).

Five experiments were carried out for each integer; the numbers to factor were products of two primes of roughly equal size.

Figure 8.1 indicates that the parallel version achieves full speed up in the average case. (It is clear, that no speed up can be achieved in the best case because with a small probability, a curve of smooth order is found at once by the sequential version.)

It should be noted in the end that function ifactor depends on an internal table of primes\(^1\) such that the sequential version can also fail by reaching the table limit.

8.4 Factoring Integers by Means of a Quadratic Sieve

The algorithm used for this benchmark is described in section 3.3.

---

\(^1\)The size of this table can be set using the command line option \(-L\) on starting MuPAD.
Of all steps of the algorithm, solving the linear system over $\mathbb{F}_2$ is the worst-suited one for computer algebra systems, since bit operations are not available; both time and storage consumption are enormous.

We use Wiedemann's algorithm, see [32]. Its essential idea is to guess the minimal polynomial of the coefficient matrix $A$ from the minimal polynomial of the linearly generated sequence $(uA^ib)_i$, where $u$ is a random row and $b$ is a random column vector.

For the time-critical steps we use two dynamic modules [29]: a lot of matrix-vector multiplications have to be carried out, and the minimal polynomial of a long sequence has to be computed.

The minimal polynomial of a linearly generated sequence is usually determined using the Berlekamp-Massey algorithm, but we simply use the extended eu-
Factoring Integers by Means of a Quadratic Sieve

Euclidean algorithm provided by the dynamic module \texttt{gf2fac}.\textsuperscript{2}

Table 8.3 shows that compared to a sequential version of \texttt{mpqs}, the speedup of the sieve-step is roughly proportional to the number of slave clusters used (as one should expect).

A comparison to other computer algebra systems shows that our \texttt{mpqs} implementation is not inferior to the algorithms – mainly the elliptic curve method – used by others. The only exception is \texttt{magma}, which uses a version of the \texttt{mpqs} algorithm written in C.

<table>
<thead>
<tr>
<th>number to factor $n$</th>
<th>par. \texttt{mpqs}</th>
<th>seq. \texttt{mpqs}</th>
<th>Maple</th>
<th>\texttt{Mathematica}</th>
<th>\texttt{Magma}</th>
</tr>
</thead>
<tbody>
<tr>
<td>300000000000000001769</td>
<td>46</td>
<td>193</td>
<td>78</td>
<td>68</td>
<td>3</td>
</tr>
<tr>
<td>30000000000000000000111</td>
<td>109</td>
<td>462</td>
<td>684</td>
<td>288</td>
<td>5</td>
</tr>
<tr>
<td>100000000000000000004563</td>
<td>221</td>
<td>935</td>
<td>4587</td>
<td>1174</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 8.3: Times Used for Factoring Integers Measured in Seconds

The experiments listed in Table 8.3 were carried out on an Ultra-I for the sequential version respectively a master on a Sparc-20 with five slaves on Ultra-I workstations for the parallel version.

A comparison between a sequential version of \texttt{mpqs} and the function \texttt{ifactor} of the \texttt{MuPAD} kernel is shown in Figure 8.2. The numbers factored were the same as in Figure 8.1.

The small points indicate the running times of \texttt{ifactor} (they are the same as in Figure 8.1), the large ones those of \texttt{mpqs}.

The graphic shows that \texttt{mpqs} is superior to the elliptic curve method (this has been known for long) and that the break-even point is even within the reach of computer algebra systems.

Since solving the linear system is the bottleneck of our current implementation, even larger numbers can be handled if only the sieve step is executed. This can be useful e.g. to determine the frequency / distribution of smooth numbers.

In Figure 8.3, all sieve reports for the seventh Fermat number are displayed: any point $(x, y)$ is coloured white iff a sieve report occurred for the $x$-th polynomial at array position $y$.

\textsuperscript{2}The dynamic module \texttt{gf2fac} is an adaption of the BiPo1Ar package for polynomial arithmetic over the residue class ring $\mathbb{F}_2$ written by Jürgen Gerhard (jngerhar@uni-paderborn.de), Departement of Mathematics and Computer Science, University of Paderborn, Germany. At present, the source code of this module is not public.
Figure 8.2: Running Times for ifactor and mpqs

Figure 8.3: Sieve Reports for the Seventh Fermat Number
Appendix A

The Accompanying CD-ROM

The accompanying CD-ROM contains a hypertext version of this book, a trial version of MuPAD 1.4.1 for Linux and Solaris systems as well as the dynamic module net including its C/C++ source code.

The net module is also available as the net package via the World-Wide-Web. Refer to Appendix B for detailed information.

The contents of the CD-ROM is not public domain but copyrighted. Refer to the following sections for detailed information about A.1 System Requirements, A.2 Installation and A.3 License Agreements.

A.1 System Requirements

To use the CD-ROM, one of the following system configurations is required:

**Linux 2.0:** (tested with S.u.S.E.¹ 5.2/5.3 (libc5 versions))
- IBM PC-compatible i86 (Pentium recommended) with CD-ROM drive
- 16MB main memory (32MB recommended)
- 1MB free disk space to use the CD-ROM
- about 110MB for a complete installation on hard disk
- ELF binary support (this is the default since kernel version 2.0)

**Solaris 2.5:** (also tested with Solaris 2.6)
- SunSPARC workstation with CD-ROM drive
- 32MB main memory (48MB recommended)
- 1MB free disk space to use the CD-ROM
- about 110MB for a complete installation on hard disk

¹Refer to [http://www.suse.de](http://www.suse.de) for further information.
A.2 Using the CD-ROM Live System

To run \\textit{MuPAD} with the dynamic module \textit{net} from the accompanying CD-ROM, this must be mounted and some UNIX environment variables must be configured first. Follow the instructions below to configure your system:

1. Mount the CD-ROM by using the corresponding UNIX command:

\begin{verbatim}
mount /cdrom (on Linux systems)
volcheck (on Solaris systems)
\end{verbatim}

The CD-ROM will be mounted at /cdrom on a Linux system and at /cdrom/cdrom0 on a Solaris system. In all following descriptions it is assumed that the CD-ROM is mounted at /cdrom.

If you cannot mount the CD-ROM in this way then ask your system administrator for information about your local configuration.

2. Include the following lines in your personal file \texttt{~/.cshrc} when using \texttt{csh} or \texttt{tcsh}:

\begin{verbatim}
setenv MuPAD\_ROOT\_PATH /cdrom
set path = (/cdrom/share/bin $path)
\end{verbatim}

Include the following lines in your personal file \texttt{~/.profile} when using \texttt{sh}, \texttt{bash} or \texttt{ksh}:

\begin{verbatim}
MuPAD\_ROOT\_PATH=/cdrom ; export MuPAD\_ROOT\_PATH
PATH=/cdrom/share/bin:$PATH
\end{verbatim}

Execute the corresponding commands within your shell to make the definition available instantly or just logout and login again.

3. To start \textit{MuPAD}, execute either the /cdrom/share/bin/mupad (for the terminal version) or /cdrom/share/bin/xmupad (XView/X11).

To read the hypertext online version of this book type the command \?net within xmupad.

Refer to Section B.3 for additional information about configuring the dynamic module \textit{net}.

\textbf{Note:} To create binaries from the sources available on the CD-ROM, a GNU \texttt{g++} compiler (version 2.7.2 or later) must be installed on your local system.

\textbf{Note:} For information about the license agreements for the software available on the CD-ROM, please refer to Section A.3.
**Note:** Running MuPAD from a local hard disk is much faster than using it from the CD-ROM. Using macro parallelism in a network requires to mount this CD-ROM respectively to install and configure its contents globally in the network.

### A.2.1 Copying on Hard Disk

MuPAD including the macro parallelism can be installed on a hard disk by carrying out the following tasks:

1. Mount the CD-ROM as described above.
2. Create a directory on your hard disk, e.g. `/usr/local/MuPAD`
   ```bash
   mkdir /usr/local/MuPAD
   ```
3. Copy the contents of the CD-ROM in the new MuPAD directory and remove the write protection if necessary:
   ```bash
   cp -r /cdrom/* /usr/local/MuPAD
   chmod -R u+w /usr/local/MuPAD
   ```
   If there is not enough disk space available, the directories `/cdrom/demo` (sources and demos) and `/cdrom/doc` (additional documentation) need not to be copied. To use these data just create a link to it:
   ```bash
   ln -s /cdrom/demo /usr/local/MuPAD/demo
   ln -s /cdrom/doc /usr/local/MuPAD/doc
   ```
4. Change the UNIX environment variables `PATH` (respectively `path`) and `MuPAD_ROOT..PATH` according to your need (see Section A.2).

In order to change and recompile the module `net`, the contents of `/cdrom/demo` (respectively corresponding parts of it) must be copied on your local hard disk.

### A.3 MuPAD License Agreements

The MuPAD versions distributed with the accompanying CD-ROM are trial versions which do not include a MuPAD user license. Thus they contain a memory limitation which can be unlocked with a license key after your copy of MuPAD is registered at *SciFace Software*.

Please read the following general and educational license and refer to Section A.3.4 to register MuPAD.
A.3.1 General License

MUPAD 1.4.1 END USER LICENSE AGREEMENT

Part I applies if you have not purchased a license to the accompanying software (the "Software"). Part II applies if you have purchased a license to the Software. Part III applies to all license grants. If you wish to purchase a license, contact SciFace Software GmbH & Co. KG ("SciFace") at Technologiepark 12, D-33100 Paderborn, Germany or send an email to info@sciface.com or use the online service www.sciface.com.

Special licenses for members of non-profit educational and scientific institutions are available. Please read Section A.3.2.

PART I - Trial License

SciFace grants you a non-exclusive single-user license to use the Software free of charge if your use of the Software is for the purpose of evaluating the Software. The evaluation period is limited to 30 days. You may use the Software in the manner described in Part II below under "Scope of Grant."

DISCLAIMER OF WARRANTY

Free of charge Software is provided on an "AS IS" basis, without warranty of any kind, including without limitation the warranties of merchantability, fitness for a particular purpose and non-infringement. The entire risk as to the quality and performance of the Software is borne by you. Should the Software prove defective, you and not SciFace assume the entire cost of any service and repair. This disclaimer of warranty constitutes an essential part of the agreement.

Even if parts of the disclaimer violates legal rights of some jurisdiction, only these parts are invalid, but not the entire disclaimer.

PART II - Single-user License

Subject to payment of applicable license fees, SciFace grants to you a non-exclusive license to use the Software and accompanying documentation ("Documentation") in the manner described in Part III below under "Scope of Grant."

LIMITED WARRANTY

SciFace warrants that for a period of ninety (90) days from the date of acquisition, the Software, if operated as directed, will substantially achieve the functionality described in the Documentation. SciFace does not warrant, however, that your use of the Software will be uninterrupted or that the operation of the Software will be error-free. SciFace also warrants that the media containing the Software, if provided by SciFace, is free from defects in material and workmanship and will so remain for ninety (90) days from the date you acquired the Software. SciFace's sole liability for any breach of this warranty shall be:

(a) to replace your defective media; or
(b) to advise you how to achieve substantially the same functionality with the Software as described in the Documentation; or
(c) if the above remedies are impracticable, to refund the license fee you paid for the Software. Repaired, corrected, or replaced Software and Documentation shall be covered by this limited warranty for the period remaining under the warranty that covered the original Software, or if longer, for thirty (30) days after the date
(a) of shipment to you of the repaired or replaced Software, or
(b) SciFace advised you how to operate the Software so as to achieve the functionality described in the Documentation.

Only if you inform SciFace of your problem with the Software during the applicable warranty period and provide evidence of the date you purchased a license to the Software will SciFace be obligated to honor this warranty. SciFace will use reasonable commercial efforts to repair, replace, advise or, for individual consumers, refund pursuant to the foregoing warranty within 30 days of being so notified.

This is the only warranty made by SciFace. SciFace makes no other express warranty and no warranty of noninfringement of third parties' rights. The duration of implied warranties, including without limitation, warranties of merchantability and of fitness for a particular purpose, is limited to the above limited warranty period.

Even if parts of the limited warranty violates legal rights of some jurisdiction, only these parts are invalid, but not the entire warranty.

If any modifications are made to the Software by you during the warranty period; if the media is subjected to accident, abuse, or improper use; or if you violate the terms of this Agreement, then this warranty shall immediately be terminated. This warranty shall not apply if the Software is used on or in conjunction with hardware or software other than the unmodified version of hardware and software with which the software was designed to be used as described in the Documentation.

PART III

SCOPE OF GRANT

You may:
• use the Software on any single computer;
• use the Software on a network, provided that each person accessing the Software through the network must have a copy licensed to that person;
• use the Software on a second computer as long as only one copy is used at a time;
• copy the Software for archival purposes, provided any copy must contain all of the original Software's proprietary notices.

You may not:
• permit other individuals to use the Software except under the terms listed above;
• permit concurrent use of the Software;
• modify, translate, reverse engineer, decompile, disassemble or create derivative works based on the Software;
• copy the Software other than as specified above;
• rent, lease or otherwise transfer rights to the Software; or
• remove any proprietary notices or labels on the Software.

TITLE

Title, ownership rights, and intellectual property rights in the Software shall remain in SciFace and/or its suppliers. The Software is protected by the copyright laws and treaties. Title and related rights in the content accessed through the Software is the property of the applicable content owner and may be protected by applicable law. This License gives you no rights to such content.
The license will terminate automatically if you fail to comply with the limitations described herein. On termination, you must destroy all copies of the Software and Documentation.

A.3.2 Educational License

MUPAD 1.4.1 END USER LICENSE AGREEMENT FOR MEMBERS OF NON-PROFIT EDUCATIONAL AND SCIENTIFIC INSTITUTIONS

If you have any questions concerning the license to the accompanying software (the "Software") or if you wish to purchase a license, please contact SciFace Software GmbH & Co.KG ("SciFace") at Technologiepark 12, D-33100 Paderborn, Germany or send an email to info@sciface.com or use the online service www.sciface.com.

SciFace grants you a non-exclusive single-user license to use the Software free of charge if you are a student, faculty member or staff member of an educational institution (junior college or college), a staff member of a non-profit research institution, or an employee of an organization which meets SciFace's criteria for a charitable non-profit organization. Government agencies are not considered educational or charitable non-profit organizations for purposes of this Agreement.

Free of charge licenses are not available for the following version of the Software: MuPAD Pro for Windows 95. Please contact SciFace to get information about special rates.

If you fit within the description above, you may use the Software in the manner described in "Scope of Grant."

If you don't fit within the description above, you can use this version of MuPAD free of charge if your use of the Software is for the purpose of evaluating the Software. The evaluation period is limited to 30 days. Please read Section A.3.1.

DISCLAIMER OF WARRANTY

Free of charge Software is provided on an "AS IS" basis, without warranty of any kind, including without limitation the warranties of merchantability, fitness for a particular purpose and non-infringement. The entire risk as to the quality and performance of the Software is borne by you. Should the Software prove defective, you and not SciFace assume the entire cost of any service and repair. This disclaimer of warranty constitutes an essential part of the agreement.

Even if parts of the disclaimer violates legal rights of some jurisdiction, only these parts are invalid, but not the entire disclaimer.

SCOPE OF GRANT

You may:
• use the Software on any single computer;
• use the Software on a network, provided that each person accessing the Software through the network must have a copy licensed to that person;
• use the Software on a second computer as long as only one copy is used at a time;
• copy the Software for archival purposes, provided any copy must contain all of the original Software's proprietary notices.
You may not:
- permit other individuals to use the Software except under the terms listed above;
- permit concurrent use of the Software;
- modify, translate, reverse engineer, decompile, disassemble or create derivative works based on the Software;
- copy the Software other than as specified above;
- rent, lease, grant a security interest in, or otherwise transfer rights to the Software; or
- remove any proprietary notices or labels on the Software.

TITLE
Title, ownership rights, and intellectual property rights in the Software shall remain in SciFace and/or its suppliers. The Software is protected by the copyright laws and treaties. Title and related rights in the content accessed through the Software is the property of the applicable content owner and may be protected by applicable law. This License gives you no rights to such content.

TERMINATION
The license will terminate automatically if you fail to comply with the limitations described herein. On termination, you must destroy all copies of the Software and Documentation.

A.3.3 About PVM

What is it: *PVM* is a system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource.

Developers: *PVM* is developed by J. J. Dongarra, G. E. Fagg, G. A. Geist, J. A. Kohl, R. J. Manchek, P. Mucci, P. M. Papadopoulos, S. L. Scott, and V. S. Sunderam at the University of Tennessee, Knoxville TN., Oak Ridge National Laboratory, Oak Ridge TN., Emory University, Atlanta GA.

Usage: *PVM* is used as transportation layer and for managing an arbitrary number of MuPAD clusters in a macro parallel network.

Documentation: The directory demo/NET/doc/ on the CD-ROM contains a complete reference manual (~280 p.).

License: Refer to the file demo/NET/Readme for detailed license information.

A.3.4 How to Register MuPAD

After registering your copy of MuPAD at *SciFace Software* you will get a license key via email. This can be used to remove all memory limitations and your copy of MuPAD will become a full version.
Note, that MuPAD has to be installed on disk before entering the license key because registration does not work on write protected media. Refer to Section A.2.1 for information about installing MuPAD on your hard disk.

Refer to the web at http://www.sciface.com/products/licenses.shtml for further information and to register your copy of MuPAD to obtain a license key. On questions please send an email to info@sciface.com or contact:

SciFace Software GmbH & Co. KG
Technologiepark 12
D-33100 Paderborn
Germany
Email info@sciface.com
Fax +49-5251-6407-99
Web http://www.sciface.com

A.3.5 MuPAD Manuals

The following MuPAD manuals describe the MuPAD system and its programming language including graphics and mathematical capabilities:


Detailed information about using and writing dynamic modules in MuPAD are available with the manual

Appendix B

The Net Package

The Net package is meant for users who have already installed MuPAD 1.4.1 and want to update the features of macro parallelism. For installing this package, a tar utility which supports option -z or the gunzip utility is needed.

Remember that on the accompanying CD-ROM the net module is installed (see Appendix A) and just need to be configured (see Section B.3) according to your local system configuration.

B.1 Copying the Net Package

The Net package contains all binaries and scripts which are needed for using macro parallelism within MuPAD 1.4.1 on PCs running Linux 2.0 and Sun-SPARC workstations running Solaris 2.5.

It is available as a gzip compressed tar archive net_sys_version.tgz and can be copied from the site http://www.mupad.de/PAPER/PARALLEL/NET/. Here, sys stands for the operating system and version stands for the MuPAD version which is necessary to use the net module.

The Net package is also available on the accompanying CD-ROM in the directory /cdrom/packages/ assuming the CD-ROM is mounted at /cdrom/.

B.2 Installing the Net Package

To install the Net package, copy the tar archive net_sys_version.tgz into the directory /tmp/ and carry out the following steps:
1. Change to the directory in which your copy of MuPAD is installed. This is typically the directory /usr/local/MuPAD/: refer to the UNIX environment variable MuPAD_ROOT_PATH or ask your system administrator.
   \[\text{cd /usr/local/MuPAD}\]

2. Extract the data from the tar archive as shown below:
   \[\text{tar xzf /tmp/net.sys_version.tgz}\]
   If your version of tar does not support option \(-z\), you must uncompress the tar archive using the command gunzip before extracting its contents:
   \[\text{gunzip net.sys_version.tgz}\]
   \[\text{tar xfp net.sys_version.tar}\]

Now all necessary files are installed and the net module can be configured.

**B.3 Configuring the net Module**

After installing the Net package it is necessary to configure the MuPAD environment in order to make use of macro parallelism. For this reason the following actions has to be carried out:

1. Change the UNIX environment variable MuPAD_ROOT_PATH to the directory where MuPAD is installed (typically /usr/local/MuPAD/). This can be done by including the following lines in your personal file \(\sim/.cshrc\) when using csh or tcsh:
   \[\text{setenv MuPAD_ROOT_PATH /usr/local/MuPAD}\]
   or including the following lines in your personal file \(\sim/.profile\) file when using sh, bash or ksh:
   \[\text{MuPAD_ROOT_PATH=/usr/local/MuPAD ; export MuPAD_ROOT_PATH}\]

2. Include the following lines into your personal file \(\sim/.mupadinit\) in order to use macro parallelism within a MuPAD session:
   \[\text{NETCONF:= } ["host1"=1, "host2"=2, ...];\]
   \[\text{module(net);}\]
   With this, a first macro parallel network is configured. Refer to the MuPAD online documentation for additional and more detailed information.
To start the clusters (MuPAD kernel processes which participate on macro parallelism) on remote hosts of the network, the program ssh\(^1\) is used by default. If one wants to use the program rsh\(^2\) for a faster network set up, this can be achieved in the following way:

Make a copy of the program rsh, name it ssh and put it into a directory which is listed in your search path (environment variable PATH respectively path) before(!) that directory which contains the original ssh. Then, a call of the program ssh results in using rsh instead.

For PVM wizards only:  To hide PVM to the normal user, a global configuration file is used to specify the PVM run-time environment and to tell the PVM daemons where MuPAD is installed. The default configuration file is installed at $MuPAD..ROOT..PATH/sys/modules/lib/pvmd.conf. Users who want to utilize special features of the PVM system can change this file according to their need. The following details have to be considered:

1. The PVM_ROOT environment variable must be set to an existing PVM environment or to the directory $MuPAD..ROOT..PATH/sys/modules.

2. If the directory specified in point 1 contains a file ./pvmd.conf, this is used as a personal configuration file for the macro parallel network (more exact: for the underlying PVM system). Refer to the pvmd UNIX manual page for detailed information about the format of this file.

### B.4 Special Tools

Because of the fact, that in some rare cases the MuPAD macro parallel network may not be terminated correctly some tools are provided which helps the user to get status information about a running PVM system and to halt respectively kill it. The following tools are available:

- `netstatus` - Displays the status of a running PVM system.
- `netstop` - Halts a running PVM system.
- `netkillpvmd` - Kills a running pvmd daemon.

\(^1\)ssh - secure shell client (remote login program), refer to the UNIX manual ssh(1).

\(^2\)rsh - remote shell, refer to the UNIX manual rsh(1)
netstatus

NAME
netstatus - displays the status of a macro parallel network

SYNOPSIS
netstatus [-h]

DESCRIPTION
netstatus displays some information about a running MuPAD macro parallel network. The script should be started on the same host where the master of macro parallelism was started.

OPTIONS
-h Displays help information

EXAMPLES
> netstatus
Hosts of the existing MuPAD Macroparallel System
==============================================
diophant
planck

Running jobs of the MuPAD Macroparallel System
==============================================

<table>
<thead>
<tr>
<th>HOST</th>
<th>JOB</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>diophant</td>
<td>-</td>
<td>40001</td>
</tr>
<tr>
<td>planck</td>
<td>mupad</td>
<td>80001</td>
</tr>
<tr>
<td>planck</td>
<td>mupad</td>
<td>80002</td>
</tr>
</tbody>
</table>

SEE ALSO
netstop(1), netkillpvmd(1), pvm(1)

NOTES
netstop does not list the MuPAD process which runs as master.

AVAILABILITY
netstop is available for the following operating systems:
PC/AT: Linux
Sun(SPARC): Solaris 2.5 (and higher)

TECHNICAL SUPPORT
For technical support send a detailed bug-report via email to <bugs@mupad.de>
netstop

NAME
netstop - stops a running macro parallel network

SYNOPSIS
netstop [-hm] [host1 host2...]

DESCRIPTION
netstop is a tool for stopping a running macro parallel network by killing the underlying PVM system. This script should be started on the same host where the master of the macro parallelism was started.

If additional arguments host1, host2, ... were given, only the cluster on these hosts are killed. This is useful if the master of the macro parallelism has been vanished somehow.

OPTIONS
-h     Displays help information
-m     Uses MuPAD to kill a still running macro parallel network. A MuPAD kernel is started which reads out the variable NETCONF and terminates all the clusters which are set. This is useful, if the system isn’t terminated correctly and there are still some clusters running, but the master of the macro parallelism isn’t available anymore. -h Gives some help

SEE ALSO
netstatus(1), netkillpvmd(1), pvm(1)

NOTES
netstop uses the script netkillpvmd for killing the still active clusters.

AVAILABILITY
netstatus is available for the following operating systems:
PC/AT: Linux
Sun(SPARC): Solaris 2.5 (and higher)

TECHNICAL SUPPORT
For technical support send a detailed bug-report via email to <bugs@mupad.de>
netkillpvmd

NAME
netkillpvmd - kills a running pvmd master daemon

SYNOPSIS
netkillpvmd [-h]

DESCRIPTION
netkillpvmd kills the running pvmd daemon on the local host.

OPTIONS
-h  Displays help information

SEE ALSO
netstatus(1), netstop(1), pvm(1)

NOTES
netkillpvmd is used by the script netstop.

AVAILABILITY
netkillpvmd is available for the following operating systems:
PC/AT:    Linux
Sun(SPARC): Solaris 2.5 (and higher)

TECHNICAL SUPPORT
For technical support send a detailed bug-report via email
to <bugs@mupad.de>
Appendix C

Changes

The dynamic module net described in this manual provides the new version of macro parallelism to MuPAD and was implemented for release 1.4.1.

Latest information about macro parallelism are always available at the MuPAD web site. Refer to http://www.mupad.de/PAPER/PARALLEL/.

C.1 With Respect to the ALPHA Release

The accompanying release of macro parallelism provides minor internal changes but also a very important improvement:

- The net module contains a new implementation of the MuPAD specific binary format MCODE which is used to encode MuPAD objects for transferring them in a heterogeneous network and for writing them to files.

With this, macro parallelism can also be used in combination with the currently available release of MuPAD 1.4.1 in a heterogeneous network environment of Solaris and Linux machines. The new MCODE will be a standard in coming MuPAD releases.

C.2 With Respect to Release 1.4

For release 1.4.0 a first prototype of the new macro parallelism was available. With release 1.4.1 the MuPAD kernel was changed due to some bugs in the line editor and the signal handling which is very important for macro parallelism.
The following improvements were introduced to macro parallelism:

- Work groups for automatic scheduling of jobs. This makes the implementation of *branch and bound* and highly recursive algorithms more convenient and efficient.

- Each global (network) variable is now locked separately and slaves no longer have to retry getting a lock, since the master stores all permit requests. The performance of the first naive approach was too bad.

- **MCode** is used a the default encoding method for transferring **MuPAD** objects over the network.

- The **PVM** [1] environment was reduced to a minimum and mostly hidden from the **MuPAD** user using **PVM** configuration files.

- The method `net::mprint` was introduced to enable slaves to display messages on the master’s console in a convenient way.

### C.3 With Respect to Release 1.3

The description of the macro parallelism as given in the *MuPAD User’s Manual 1.2.2* [31] is out of date and has no further relevancy.
Appendix D

Glossary

atomic operation An operation which cannot be interrupted.
built-in function A MuPAD user function that is implemented in the MuPAD kernel.
CAS The abbreviation of computer algebra system.
cluster A MuPAD kernel which has loaded the dynamic module net. It is either the master or a worker.
displacement To unload a dynamic module, i.e. to unlink and remove it from memory. Modules can be unloaded by the user as well as by automatical displacement and replacement strategies of the module manager.
domain A data structure (DOM_DOMAIN) for user-defined data types in MuPAD. It is also used to represent library packages and dynamic modules. Refer to the MuPAD User’s Manual [31] Section 2.3.18 and the paper [6] for detailed information.
domain element An element of a constructed or user-defined data type (domain) in MuPAD.
dynamic library A special kind of a machine code library (also refer to PIC). It can be dynamically linked to a program at run-time.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic linking</td>
<td>A dynamic library can be linked into a program or a process at run-time. Under special technical conditions, such a library can be unlinked and removed from the memory at run-time.</td>
</tr>
<tr>
<td>dynamic module</td>
<td>A special kind of a dynamic library which contains so-called module functions. It can be loaded into MuPAD and used similar to a library package. Dynamic modules can be displaced at run-time. Refer to the Dynamic Modules Manual [29] for detailed information.</td>
</tr>
<tr>
<td>evaluation</td>
<td>To evaluate a MuPAD expression -which is represented as a tree- means to derive it by visiting each node recursively and substituting it with its derivation.</td>
</tr>
<tr>
<td>GNU</td>
<td>Read the GNU General Public License at the Internet.</td>
</tr>
<tr>
<td>garbage collection</td>
<td>Frees unused memory cells maintained by the memory manager.</td>
</tr>
<tr>
<td>global variable</td>
<td>Refer to network variable.</td>
</tr>
<tr>
<td>group job</td>
<td>A task sent to a group master of a work group.</td>
</tr>
<tr>
<td>group master</td>
<td>This cluster holds the jobs of a work group and is responsible for job scheduling within this group. It selects a group slave to execute a job and collects the result afterwards. Do not confuse with the master of macro parallelism.</td>
</tr>
<tr>
<td>group slave</td>
<td>A worker which is part of a work group.</td>
</tr>
<tr>
<td>heterogeneous</td>
<td>A heterogeneous network contains computers of different hardware architectures and/or computers running under different operating systems.</td>
</tr>
<tr>
<td>library</td>
<td>This is either a dynamic library, static library or MuPAD library.</td>
</tr>
<tr>
<td>library function</td>
<td>A function of a library package. It is written in the MuPAD programming language.</td>
</tr>
<tr>
<td>library package</td>
<td>A collection of library functions. A package pack can be loaded within a MuPAD session using the command loadlib(&quot;pack&quot;);</td>
</tr>
</tbody>
</table>
linking  Means to glue together machine code into an executable program or a dynamic library.

MCODE  A MuPAD specific binary format for encoding MuPAD objects when transferring them in a network and/or writing them to files.

MuPAD function  Any function available within a MuPAD session. This can be a built-in function, library function or module function.

MuPAD library  Collection of library packages. It contains most of the mathematical knowledge of the CAS MuPAD.

master  The master of macro parallelism. Do not confuse with a group master of a work group.

message passing  Note, that in MuPAD there is principally no difference between data and programs written in the MuPAD programming language. Thus, sending messages between clusters also includes sending and evaluating programs. At this point, the use of the term message passing in the context of macro parallelism differs slightly from its usual meaning.

mmg  See module generator.

module  See dynamic module.

module function  A MuPAD function defined in a module.

module generator  The tool to create a loadable and executable module from a C/C++ module source code file. It uses a usual C++ compiler and linker for this. Refer to the Dynamic Modules Manual [29] for detailed information.

network pipe  A pipe of a remote host in a computer network which can be written to remotely. Refer to net::writepipe for details.

network queue  A queue of a remote host in a computer network which can be written to remotely. Refer to net::writequeue for details.
network variable  A variable which is available in the whole network. If it is changed on one cluster then this change effects all other clusters too.

PIC  Abbreviation of *position independent code*. Typically, a dynamic library is compiled as PIC to enable programs to dynamically link it (compiler option -fpic, -pic or -PIC). Refer to your compiler manual for detailed information.

PVM  Parallel Virtual Machine System. This is used by the macro parallelism as transportation layer and for managing MuPAD processes in a heterogeneous network. Refer to Appendix A.3.3 for additional information.

pipe  Refer to network pipe.

queue  Refer to network queue.

scheduling  Here: managing a queue of group jobs on a group master, sending them to idle group slaves for evaluation and storing the result until it is fetched by the orderer.

slave  See worker.

software integration  Exchange of data and commands between two or more software components. Also refer to dynamic module. Detailed information about software integration in MuPAD are given in [29] [30] [28] [27] [24].

static library  An archive of machine code routines.

static module  A *dynamic module* carrying the attribute static, which protects it from being displaced.

worker  Any cluster but the interactive master.

work group  Consists of a group master and workers called group slaves. Jobs may be sent to the master which instructs one of its slaves to compute it.
Bibliography


Index

algebraic number fields, 57
algorithm, 49
   debugging, 69
   elliptic curve method, 52
   factoring integers, 52, 54
   Fibonacci numbers, 59
   gcd, 57
   modular-, 7, 57
   multiplying polynomials, 60
   parallel search, 50
   profiling, 69
   quadratic sieve, 54
   recursive, 59, 60
atomic operation, 2, 11
automatic scheduling, 2, 11

benchmarks, 99
benchmarks
   gcd, 99
   Karatsuba, 100
   quadratic sieve, 101
branch and bound, 3, 97

CD-ROM, 105
   copying to disk, 107
   license, 107
   live system, 106
   PVM, 111
   system requirements, 105
changes, 119
   ALPHA, 119
   release 1.3, 120
   release 1.4, 119

cluster, 1
common errors, 73
contributions, 4, 86
dead lock, 64
debugging, 13, 65, 66, 69
design and implementation, 81
distributed computing, 15, 16, 33
divide and conquer, 2
documentation
   net, 17
   help pages, 13
   related, 4
domains, 70, 71
dynamic modules, 3, 4, 52, 56, 69, 70, 74, 122
equation method, 52
email address
   MuPAD, 5
   SciFace Software, 112
   the authors, 5
examples, 49
factoring integers, 52, 54
Fibonacci numbers, 59
gcd, 57
global definition, 13
global evaluation, 13
global variable, 2, 22, 23, 64
   read from, 10
   write to, 10
glossary of terms, 121
GMP, 70
group
  master, 2
  slave, 2
  work-, 2
help pages, 13
illegal functions, 65
IMSL, 70
inside
  compiling net, 93
  debugging net, 94
  desired extensions, 97
  error handling, 91
  event handling, 91
  global variables, 84, 89
  implementation, 86
  locks and semaphores, 84
  missing features, 95
  net, 83
  source code, 92
  test files, 93
  using dynamic modules, 86
  using PVM, 85, 87
  work groups, 85, 90
installation, 106, 107, 113
job
  handle, 11
  scheduling, 11
Karatsuba, 60
late results, 74
license agreement, 107
  educational, 110
  general, 108
  PVM, 111
line length, 65
Linux, 105
load balancing, 97
lock, 11, 30, 30, 40, 41, 45
master, 1
message passing, 9, 38, 39, 46, 47, 63, 123
missing definitions, 73
MP, 70
MuPAD
  contributions, 4, 86
  email address, 5, 112
  installation, 107
  license, 107
  manual, 4
  manuals, 112
  register, 111
  web site, 4, 112
NAGC, 70
naming convention, 2, 65
navigator
  user interface, 14
net package, 113
  configuring, 114
  copying, 113
  installing, 113
net::acquiresem, 14
net::compute, 15
net::define, 16
net::doc, 17
net::freesem, 18
net::gclusternet::gresult, 27
net::gglobal, 22
net::gmasterinit, 25
net::gmasterquit, 25
net::gmaster, 24
net::gprocess, 26
net::gresult, 27
net::gslaveinit, 28
Index

net::gslavequit, 28
net::gstatus, 29
net::halt, 30
net::initlock, 30
net::initsem, 31
net::ismaster, 32
net::map, 33
net::master, 35
net::mprint, 36
net::pref, 36
net::readpipe, 38
net::readqueue, 39
net::setlock, 40
net::shutdown, 41
net::spinlock, 41
net::spinsem, 42
net::topology, 44
net::unsetlock, 45
net::writepipe, 46
net::writequeue, 47
NETCONF, 3, 8, 35, 83
netkillpvmd, 118
netstatus, 116
netstop, 117
network
  communication, 63, 76
  computing, 13, 16
  concept, 1
  configuration, 7, 35, 63
  definition, 13
  documentation, 13
  emergency exit, 8
  group, 2, 25, 59, 60
  lock, 2, 11, 30, 30, 40, 41, 45
  mapping, 13, 15, 16, 33
  output, 13, 36, 66
  preferences, 8, 36
  semaphore, 2, 11, 14, 18, 31, 31, 42
  shutdown, 7, 30, 41
  startup, 7, 35, 88
status information, 115
synchronization, 11
topology, 9, 32, 44
tracing, 36, 66
variable, 2, 10, 22, 23, 39, 47

NTL, 70

online documentation, 13
OpenMath, 70
operating system
  Linux, 105
  Solaris, 105

parallel
  algorithms, 49
  debugging, 69
  Fibonacci, 59
  greatest common divisor, 57
  multiplying polynomials, 60
  profiling, 69
  search, 50
pipe, 2, 38, 46
  naming convention, 2
  read from, 9, 38
  write to, 9, 46
priority queue, 98
problem solving environment, 71
problems, 73, 76, 77
  ASCII encoding, 76
  common errors, 73
  dynamic modules, 74
  late results, 74
  MCODE encoding, 76
  missing definitions, 73
  PVM, 75
  quick help, 77
  stack overflow, 77
  unintended evaluation, 73
profiling, 69
programming
addition(al aspects, 69
efficiency, 63
good code, 65
guidelines, 63
illegal functions, 65
PSE, 71
PVM, 75, 111
tools, 115
quadratic sieve, 54
queue, 2, 39, 47
  naming convention, 2
  read from, 9, 39
  work-, 1, 9
  write to, 9, 47
quick start, 3
real world application, 71
reference manual, 13
register MuPAD, 111
related documentation, 4
release notes, 119
rsh, 115
scheme of
  module net, 81, 82
  network communication, 2
  problem solving environment, 71
  work groups, 2
SciFace Software, 111
search parallelism, 50
semaphore, 11, 14, 18, 31, 31, 42
shell scripts, 115
  netkillpvmd, 118
  netstatus, 116
  netstop, 117
slave, 1
software
  installation, 106
  integration, 70
manuals, 112
MuPAD license, 107
net package, 113
register, 111
Solaris, 105
ssh, 115
stack overflow, 64, 77
synchronization, 11
system requirements, 105
tutorial, 7
unintended evaluation, 73
updating module net, 113
web site
  license, 112
  macro parallelism, 113
  MuPAD, 4
  net package, 113
  parallelism, 119
  SciFace Software, 112
work group, 2, 11, 59, 60, 64
  configuration, 25, 28
  job scheduling, 21, 26, 27
  status information, 19, 20, 24, 29
worker, 2
Reihe MuPAD Reports

Dynamische Module
Eine Verwaltung für Maschinencode-Objekte zur Steigerung
der Effizienz und Flexibilität von Computeralgebra-Systemen
Andreas Sorgatz, Paderborn
Okt. 1996. 150 Seiten. ISBN 3-519-02195-1

Ein denotationales Modell für parallele objektbasierte Systeme
Holger Naundorf, Paderborn
Dez. 1996. 186 Seiten. ISBN 3-519-02197-8

Entwicklung einer Programmierumgebung für die
Parallelverarbeitung in der Computer-Algebra
Oliver Kluge, Paderborn

MAMMUT
Eine verteilte Speicherverwaltung für symbolische Manipulation
Holger Naundorf, Paderborn

User's Guide to Macro Parallelism in MuPAD 1.4.1
T. Metzner, M. Radimersky,
A. Sorgatz, S. Wehmeier, Paderborn

Algebraische Algorithmen zur Lösung von
linearen Differentialgleichungen
Winfried Fakler, Paderborn

B. G. Teubner Stuttgart · Leipzig
Before unsealing this CD-ROM, make sure you have read and understood the license agreements listed in Appendix A of this book.

This CD-ROM contains a trial version of MuPAD for PC (Linux 2.0) and SunSPARC (Solaris 2.5) including the net module which provides macro parallelism to MuPAD. Note that the operating system must support ISO 9660 with Rock Ridge extension to mount this CD-ROM file system. Use the command `mount /cdrom` respectively `volcheck` to mount this CD-ROM.

For all queries regarding MuPAD please contact Sciface Software GmbH & Co. KG:
Email: info@sciface.com, Fax: +49-5251-6407-99
Interactive general purpose computer algebra systems gain more and more relevancy in mathematics, education and engineering but also in related science. They are developed from monolithic systems to more open integrated working environments providing object oriented programming languages, flexible and efficient concepts of software integration as well as parallel programming features.

This book is addressed to users and developers of parallel algorithms in MuPAD. It describes Macro Parallelism in MuPAD which is based on the concepts of message passing, global variables and work groups and enables users to implement parallel algorithms for distributed computations in a heterogeneous computer network in a very easy way.

This book includes the CD-ROM Macro Parallelism in MuPAD 1.4.1 with a hypertext version of this manual, trial versions of MuPAD 1.4.1 for Linux 2.0 and Solaris 2.5 as well as the net module which provides macro parallelism to MuPAD. The CD-ROM also contains the C/C++ source code of the dynamic module net. Thus users can extend macro parallelism by any features desired or adapt it according to their need and can port it to other platforms and operating systems.

The MuPAD web site provides additional information, files and support and can be found at http://www.mupad.de.