Parallel Matlab Toolbox

User Documentation

Erland Svahn M.Sc. in Computer Science

Chalmers University of Technology, Sweden

ersva@igb.polymtl.ca

The Parallel Matlab Toolbox was developed at

Ecole Polytechnique de Montréal, Canada

Institut Génie Biomédical

Supported by the National Research Council of Canada, FCAR

A part of the project upon which this publication is based was performed pursuant to the University of Texas Grant CA64597-01 with the NIH, PHS

June 5, 2001

Preface

The Parallel Matlab Toolbox has been developed as a Master of Science thesis project in Computer Science at the department for Computer Science at Chalmers University of Technology, Sweden. The project has been done at the biomedical department of École Polytechnique de Montréal, Canada, for research on techniques for detecting breast and prostate cancer. The local supervisor is Professor Michel Bertrand. The supervisor at Chalmers University of Technology is Associate Professor Philippas Tsigas.

The Parallel Matlab Toolbox is based on the Parallel Virtual Machine (PVM) [5] libraries, and the DP Toolbox version 1.4.9 [6]. The copyright notices of these products can be found on the next page. The copyright notice for this project can be found below.

PM Toolbox project was supported by the National Research Council of Canada, FCAR (Quebec Ministry of Education). A part of the project was performed pursuant to the University of Texas Grant CA64597-01 with the NIH, PHS.

Table of Contents

Introduction

This Manual

This manual begins by defining where further help can be found, and what the system requirements are. The following chapter provides what is needed for getting a fast start to the system: starting it, defining a function to dispatch, and finally dispatching it in a network of computers. The "getting started" chapter is followed by a more extensive user guide that covers all areas of the parallel Matlab system, dispatcher, including the graphic user interfaces. This part can be used as a reference on how to do different things, but it could also be read right through in order to gain a better understanding for the functionality of the parallel Matlab toolbox. The chapter concludes with a few examples of how to define dispatchable functions. Further information about how to execute specific functions can also be found via the Matlab help. The user documentation is terminated by an installation manual.

Online Help

The latest information about the toolbox can be found at:
www.igb.polymtl.ca, or by contacting ersva@igb.polymtl.ca or bertrand@igb.polymtl

System Requirements

The parallel Matlab toolbox has been developed in a network of SUN4SOL2, Pentium PCs and IBM RISC architecture computers. The SUN4SOL2 architecture computers have Sun Solaris Unix kernel version 5.7. The PC run Slackware Linux distribution 7.1.0 with kernel version 2.2.18. The IBM RISC computers use AIX

The preliminary tested system requirements are:

- PVM version 3.4.3
- One Matlab license for each Matlab process
- Matlab 5.3 or 6 (for SUN4SOL2 architecture)
- Matlab 6 (SUN4SOL2, Unix/Linux PCs, AIX IBM RISC architectures)

Tutorial

This section serves as a quick presentation of how what the Parallel Matlab toolbox offers followed by a brief guide of how to get started using it. This latter consists of a description of how to start the parallel system, how to define a dispatchable function and how to dispatch it.

What is the Parallel Matlab toolbox?

The Parallel Matlab toolbox allows a Matlab user to speed up calculations by distributing them to the potential computation power in a heterogeneous network. The distribution is effectuated with load balance in mind. This implies that it dynamically takes into account the constantly varying load on each computer as well as the different capacity of each computer. This allows more efficient computers with less load to effectuate a greater part of the total calculation to be done than a less efficient computer or one with a greater load due to other users in the system. The toolbox is built on a flexible structure that allows the division of one problem into sub-problems either by distributing data – or functionality. The structure also allows the user to dynamically adjust the computer resources used while the distributed calculation is taking place. The distribution is fault tolerant in a way that “lost” partial calculations will be rescheduled and processed. Also, if errors occur in some partial calculations but not in others, this will not interrupt the complete calculation nor prevent the user from retrieving the correct data.

The Parallel Matlab toolbox consists of two layers: the system and the dispatcher. The first defines and manages the computation resources in the network. A computation resource in the in the network corresponds to a Matlab process. By Matlab process it is meant what the user obtains when he at the Unix prompt starts Matlab. The idea is to have multiple communicating Matlab processes in the network, which requires one Matlab license for each Matlab process. The second layer of the toolbox – the dispatcher – is “placed on top” of the first and allows the user to distribute calculations in the defined system.

Which Calculations Can Be Distributed?

The parallel Matlab toolbox is flexible and can distribute a wide range of problems in the network. In general two types of problems can be discerned: either data or functionality can be distributed. Distributing the data means evaluating a fixed expression on different data, whereas distributing the functionality means evaluating different expressions on the same (or different) data. How the distribution is done is defined by a *function definition* – the recipe of the distributed calculation. The toolbox does not provide “automatic” partition of a problem into a parallel version of the program. The user has to create the parallel program, i.e. define the *function definition*, keeping in mind what is required for a program to be distributed. A basic characteristic for a problem that can be partitioned and distributed in the network is that it *consists of separable parts that do not depend of one another*. A simple, yet illustrative, example would be the applications of autocorrelation on several different images. Here, the application of the autocorrelation on each image would form several independent sub-problems. A more general problem description would be a “for loop” where each iteration takes considerable time and does not depend on previous iterations. Now the dispatcher could play the role of the for loop, but distributing the different iterations to other Matlab processes in the network. The code inside the original “for loop” is typically stored as an m-file that will be called from the different Matlab processes – all executing it with different data. This corresponds to data distribution. Functionality distribution could be visualised by several subsequent “for loops” that do not depend on each other. A factor that highly affects the efficiency of the distribution of tasks in the network is, of course, the amount of transferred data relative to the time needed for processing this same data.

The Parallel Matlab System

The Parallel Matlab system provides an interface to C libraries establishing communication and exchange of data between processes in a heterogeneous network. This established network computers with processes that can exchange data, forms what is denoted the Parallel Virtual Machine (PVM). The PVM allows the user to access all included resources in the network regardless of where they are situated or which data format is used on different computers of different architecture. The Parallel Matlab system allows the user to add any compatible and accessible computers in the network to the PVM. Once the computer – which will hereon be referred to as the *host* – is member of the user's PVM it is possible to start Matlab processes on it that will be able to communicate with any other Matlab process in the PVM. The user's first started Matlab process – which is started in the way the user normally starts Matlab – has a console window where the user can provide input and see the results of this. The Matlab processes that the user subsequently adds to the parallel Matlab system can either have a console or run in background. If they have their own console there are two ways of using them – either by accessing them from another Matlab console or by using the Matlab process' own console directly and typing the commands. If a Matlab process is running in background it can of course only be accessed the former way. The structure of the Parallel Matlab systems can be illustrated by the following figure:

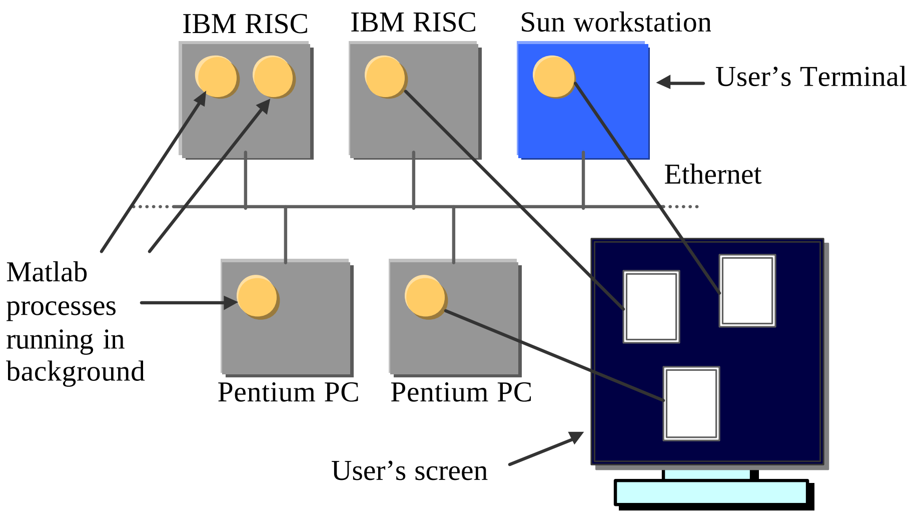


Figure 1. Block diagram of a sample parallel Matlab system

In this example the PVM is constituted by two IBM RISC multiprocessor computers, two Pentium PCs and one SUN workstation connected by an Ethernet. Each one of these computers is identified by its hostname (this can normally be found in the Unix environment variable *host* or *hostname*). The first Matlab process started is the one on the user's terminal. From this Matlab process there have been five other ones started via the parallel Matlab system and PVM on the different other hosts. Since the IBM RISC hosts are multi-processor computers it may be beneficial to start several Matlab processes on them since they will take advantage of the different processors. All the Matlab processes can communicate via the PVM and exchange data structures.

When starting a Matlab process it is possible to start it with a set of attributes: work directory, priority, start-up task, task to execute if the former generates an error, and run-mode. The run-mode determines whether the process runs in the background or in the foreground with a console. All these attributes are collected into the attributes of what is denoted a *Virtual Machine*, VM. The virtual machines serve two purposes. The first is to define the above mentioned attributes for Matlab processes. The second is to define the calculation resources for a calculation to distribute, e.g. a calculation could be allowed to use all resources of virtual machine 1 and 2 – meaning that it could use all Matlab processes that are members of these two virtual machines. A Matlab process can be a member of any number of virtual machines. A Matlab process is always started as a member of the virtual machine of which it is using the start-up attributes. It may also join other virtual machines. The virtual machines can be used to define complex resources for complex computation problems where certain Matlab processes could be used to solve a certain set of problems, e.g. due to insufficient memory or different software packages installed locally on different computers.

The Parallel Matlab Toolbox Dispatcher

The dispatcher acts as central agent (*master*) in what is referred to as a master and slave paradigm meaning that there is one process (the dispatcher process) that directs the others that are called the slave processes. This can be visualised by the following figure where the dispatcher receives a *function definition* and inputs, and produces an output. The *function definition* basically serves as a recipe for how to make the distribution of the calculation using the input provided.

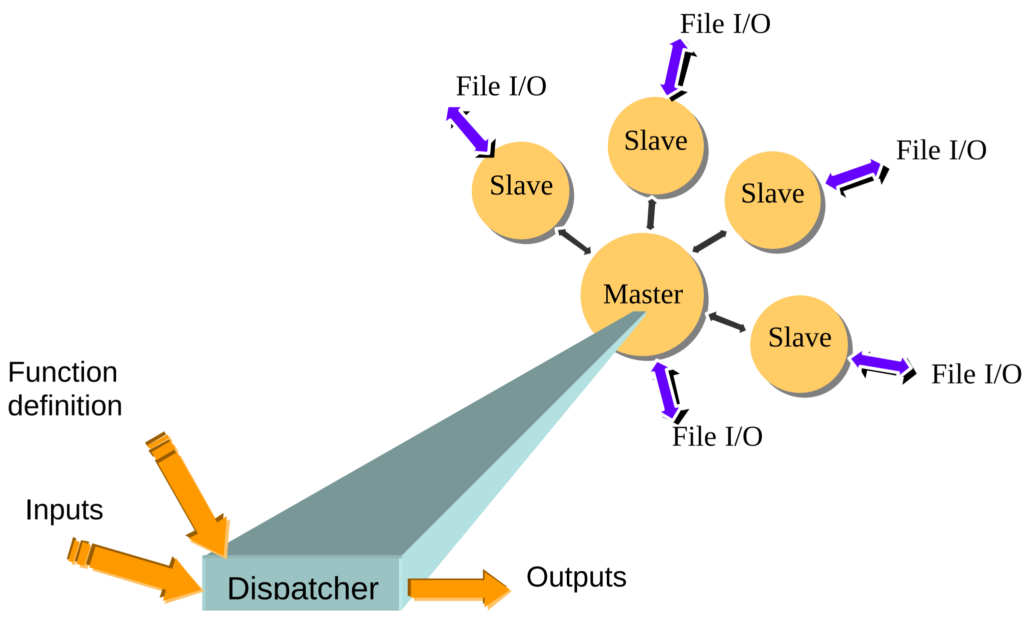


Figure 2. Diagram showing the master/slave layout of the dispatcher

The dispatcher distributes different sub-parts of calculations *dynamically* to a set of Matlab processes defined by the parallel Matlab system (as a *Virtual Machine*). This means that the dispatcher only assigns one sub-part of a calculation at a time to a slave process. Upon completion of the sub-calculation the slave process notifies the dispatcher and returns the possible results. The slave process is then assigned a new task. This gives an automatic distribution of the complete calculation in a way that a slave process running on a more powerful computer – or a less busy one – will do a greater part of the work (a greater number of sub-calculations) than a process running on a less powerful computer – or one that has a heavier load due to other users in the system. This constant adapting to the dynamically changing resources of the system at all times is the main advantage of using the master and slave paradigm. It should however be noted that when a lot of data passes through the dispatcher its readiness to dispatch tasks decreases which seriously brings down its effectiveness. To a certain point this can be avoided by letting slave processes load large amounts of data from disk instead of obtaining the data directly from the dispatcher. Likewise, if the output of a sub-calculation is of large size the slave process can save it to disk instead of returning it to the dispatcher process. This load and save of the input and output to a slave can for example be used to apply an algorithm on different images, etc stored in files.

The dispatching of a calculation needs to be *fault tolerant* and is so in the following ways. First, if a Matlab slave process dies during the dispatch, the task it was evaluating will be rescheduled to another process and the dead Matlab slave process restarted if possible. Second, if a distributed task exceeds a time limit the task will be interrupted and rescheduled. Third, if a Matlab error occurs in an distributed calculation it will be logged and the total calculation will not be interrupted. This is because an error in a sub-calculation does not necessarily imply that the rest of the calculation is erroneous, but may for example be due to exceeding matrix dimensions for that particular part of the calculation. Four, if the dispatcher process (the master) should die, everything needs to be restarted but it is still possible to resume the calculation from a previously saved state. The user determines how often this state should be saved.

The number of slave processes used for a problem is not something that needs to be predefined before starting the dispatch. It is possible to add and remove Matlab slave processes during the dispatch.

Start a Parallel Matlab Session

A parallel Matlab session can either be started using graphical user interfaces or by manually typing Matlab commands. Even if you will use the graphical user interfaces it is recommended to read through this section so as to gain a better comprehension for how it works.

The parallel Matlab session containing different communicating Matlab processes is started using the command `pmopen`, or `pmcfg` using the graphical user interface. The `pmopen` command can take three parameters. The first defines which computers will be used. The second defines the attributes of the *virtual machines*. These are used to specify the priority, working directory, run-mode, etc of different sets of Matlab processes. They are also used later – during the dispatching of a problem – to define which Matlab processes are available for solving the problem. The third argument defines where to start the actual Matlab processes, how many to start, and to which Virtual Machine(s) each Matlab process belongs. The below displayed command can be used to start the parallel Matlab session, the three arguments will be defined further below.

```
>> pmopen(hostnames, virt_m, mat_proc)
```

The first argument contains the hostname(s) of the hosts in the system or 'd' for default. The default is specified by a file called "pvmdefconf.m", see more about this in the user guide chapter. For now, we will content ourselves with providing a cell list of two hostnames (or use an empty list to utilise only the current computer). The current computer will always be part of the system even if it is not listed. An example follows:

```
>> hostnames = {‘platine’ ‘xenon’}
```

The second argument defines one or several virtual machines. It contains any combination of Matlab *struct(s)* or the character 'd' for default. Using 'd' instead of a *struct* leads to the creation of a default virtual machine. To create multiple virtual machines a cell list can be used. To learn more about the attributes see the section on Virtual Machines in the user guide chapter. The following shows three different examples of configurations of virtual machines that can be provided to the `pmopen` command

```
vm.wd      = pwd;     % use current working directory
vm.prio    = ‘low’    % low priortiy Matlab processes
vm.try     = ‘path(‘/home/ersva/myfiles’,path) % initialise
vm.catch   = ‘’;      % do nothing if error during initialisation
vm.runmode = ‘fg’     % run in foreground with a console window

virt_m = ‘d’      % default virtual machine only
virt_m = vm       % use the above defined virtual machine
virt_m = {‘d’ vm} % 1 default virtual machine and 1 defined above
```

The different virtual machines will be referred to by indices, the first created is 0, the second 1, etc. Type “`methods vm`” to learn more about how to manipulate and access the Virtual Machines.

The third arguments contains all the information needed to start all Matlab processes. It is a cell list of three to four entries. The first entry of the cell list contains another cell list specifying all hosts on which to start Matlab processes. The second entry specifies how many Matlab processes to start on each and everyone of these hosts. The third entry specifies to which virtual machine(s) each started Matlab process belongs. Lastly, the fourth entry specifies files that will be used for any output to the screen of each started Matlab process that belongs to a virtual machine with the attribute runmode set to background.

The third argument provides a complex structure where everything can be specified for each slave, but to simplify matters, it can also be used in more easily understood ways. An example would be when the user wants to start two Matlab processes: one on the host *xenon* and another on *platine*. The second entry could then be either *[1 1]* or simply *1*. However, if the user wants to start two Matlab processes on the host *platine* the second entry has to be *[1 2]* since a different number of Matlab processes are to be started on different hosts. The third entry works in a similar manner whereas the fourth entry must be carefully configured to contain different output filenames for all Matlab processes. The following example illustrates an argument to `pmopen` for creating two Matlab processes, one on each of the hosts *xenon* and *platine*. Both of these processes belong to

Virtual Machine 0, which has the run-mode attribute set to foreground, so no console output file needs to be defined.

```
mat_proc = {{‘xenon’ ‘platine’},1,0}
```

Below can be found a complete example of a configuration created to start a session on two hosts, with one Matlab process on the first and two on the second. A virtual machine is created that specifies that the Matlab processes will run in background with normal priority and with the work directory “*~/workdir*”. Since it is the first (and only) virtual machine created it will be accessed with the index 0. The files for the output of each Matlab process running in background is stored to the /tmp directory which is a local directory on each host, so attention must only be made to make sure that the files on each host are different.

```
hosts = {'xenon' 'platine'};
num = [1 2];

%define a Virtual Machine
vm.wd      = '~/workdir'; % work directory
vm.prio    = 'normal';   % run matlab processes with low priority
vm.try     = '';         % no initialisation of Matlab process
vm.catch   = '';
vm.runmode = 'bg';       % run matlab processes in background

%each Matlab process will have an output file for its console
%output:

for m=1:length(hosts)
  for n=1:num(min(length(num),m))
    console_out_files{n,m}=['/tmp/stdout' int2str(n)];
  end
end

% open the Parallel Matlab System session
pmopen( hosts, vm , {hosts,num,0,console_out_files} )
```

Each Matlab process is uniquely identified by a task id. The numbers returned by `pmopen` are the task ids for the newly started Matlab processes. Information about the current Parallel Matlab System and its Matlab processes can for instance also be given by command `pmstate`. Note however that this command displays information on all of the user’s *PVM tasks* in the network, e.g. if XPVM (which is a PVM graphical monitor) is running this will be one PVM task. To only see the Matlab processes that belongs to this parallel Matlab session, use `pmall` or `pmothers`. An example of using `pmstate` can be found on the following page.

```
>> pmstate
hosts and their PVM id:s and architectures
xenon    262144   SUN4SOL2
helium   524288   SUN4SOL2
There are 3 task(s) running:
task id:262145   spawned by:0    and running on:262144
task id:262146   spawned by:262145    and running on:262144
task id:524289   spawned by:262145    and running on:524288
```

Define a Function to Dispatch

A basic dispatchable function is a function where the same expression is evaluated several times but on different data. The easiest way to define a function may be to use the graphics user interface *funed* described in the user guide, but it is also possible to "program" it directly from the Matlab prompt by manually filling out the fields of the definition structure. The function definition is stored in a Matlab user object called *pmfun*. To manually make a dispatchable test function use the example in the Matlab help for dispatch (`help pmfun/dispatch`).

Dispatch a Function

Once a dispatchable function is defined (in a *pmfun*) and we have the input data all we need to define is which Matlab processes that will be available for the calculation. This is specified by defining sets of Matlab processes denoted V*irtual Machines* (VM). All new Matlab processes are by default members of *Virtual Machine* “0”. The *Virtual Machine* entry required by the dispatch command should thus in this basic case be 0.

```
[errors,output]=dispatch(fun,0,input)
```

This will automatically start a graphic status display that shows the progress of the dispatch. It also allows the user to dynamically update the system by adding or deleting hosts and/or Matlab processes to the Parallel Matlab System while dispatching. If for some reason no graphic user interface is desired it can be cancelled by adding an option to the dispatcher as follows:

```
[errors,output]=dispatch(fun,1,input,[],[],’gui’,0)
```

User Guide

This user guide serves as reference where the user can quickly find help, descriptions and further indications on how to proceed. It is divided into sections of different topics, but could also be read right through in order to get a better grip of the Parallel Matlab toolbox. In most cases only indications are given to which functions to use. In these cases further help can be found in the Matlab help files, which also contain example Matlab code. The different topics discussed in the user guide range from how to start up the system, and what the different names and concept mean, to installation manual and trouble shooting. The user guide concludes with examples *function definitions* of some distributed functions. To obtain a list of all parallel Matlab toolbox system commands type `help pm` at the Matlab prompt.

The Parallel Matlab Session

Once installed, the Parallel Matlab Toolbox is available to the user, however no parallel Matlab programs or dispatches can be executed before a Parallel Matlab session is started. Starting the Matlab session means that Matlab becomes linked to the underlying communication libraries and new Matlab processes can be started and stopped. The only part of the Parallel Matlab Toolbox that can be used without starting a parallel Matlab session is the part for defining dispatchable functions, but without executing them. The parallel Matlab session can be seen as the workspace of the user's distributed applications, and there can only be one session at a time. The idea is to not have to restart the Matlab processes every time the user wants to distribute a calculation in the network. To divide the resources into different sets, with different configurations, the user can define multiple "Virtual Machines" in the defined session. Each Matlab process belongs to one or several virtual machines.

Starting and Closing the Parallel Matlab Session

A parallel Matlab session is manually started with `pmopen`, or using `pmcfg` for a graphical user interface. `pmcfg` can be used at any time to modify the parallel Matlab system. The current configuration at any time can be retrieved using `pmgetconf`. This configuration can later be given to `pmopen` to reinstate the system. To close the parallel Matlab session use `pmclose`.

When defining the parallel Matlab session there are three things to define: which *hosts* should be a part of it, which *virtual machines* there are, and finally – which *Matlab processes* there are and to which virtual machine(s) each and everyone of them belong. Each Matlab process, except the one originally started by the user as a normal Matlab process, must belong to a virtual machine when started. It can later leave this virtual machine and join other(s). For each of the three above mentioned things to define there is a default.

The `pmopen` default for which hosts are to be part of the parallel Matlab system is given by a file called "pvmdefhosts.m". This file can either exist in the user's path at start-up of Matlab, or can later be specified using `pvm_setdefaultconfig`. This is basically a configuration file for PVM and all details on how to define this can be found in the user documentation for PVM. Here, we will only give a brief description of what is important for the parallel Matlab toolbox. The file, which is a text file, must contain "`* ep=$PATH`" so as to specify to PVM where to find the executable files to start when spawning new Matlab processes. If this is not present, or if the user's path does not contain the parallel Matlab toolbox's binary path for the correct architecture, a PVM error (-7) will be given for each Matlab process the user attempts to start. The following lines contain the desired hosts on separate lines. An example of the "pvmdefconf.m" follows:

```
# PVM configuration file.
* ep=$PATH
helium
xenon
# End of File
```

The default virtual machine specifies the current work directory as the work directory for all member Matlab processes. Furthermore, it specifies the processes to have normal priority and to run in foreground. No initialisation is done of Matlab processes when started as members of the default virtual machine.

The default for the third argument of `pmopen`, which decides where and how to start Matlab processes, is to spawn one Matlab process on each defined host in the session. The Matlab processes started will belong to virtual machine 0. If this virtual machine does not exist a default virtual machine is created. If virtual machine 0 specifies that its Matlab processes start in background the default is that their console output will be lost (sent to the file /dev/null).

When a parallel Matlab session is opened a Matlab process id or an error code is returned for each Matlab process. The error codes are described in the section on PVM and Matlab, and the Matlab process ids are described in the following section.

Identifying Matlab Processes

Each Matlab process is uniquely identified by a task id that is returned at its creation (either by `pmopen` or `pmspawn`). The id of a Matlab process can be used to specify Matlab processes for evaluating expressions, sending data, receiving data, etc. The Information about the current Parallel Matlab System and its Matlab processes can be given by the following functions:

`pmstate`	Displays all hosts and Matlab processes with their ids. Also displays all other PVM tasks.
`pmall`	Returns the ids of all Matlab processes in the system, including the Matlab process where the function is executed.
`pmothers`	Returns the ids of all Matlab processes in the system except the Matlab process where the function is executed.
`pmid`	Returns the id of the Matlab process where the function is executed.
`pmparent`	Returns the id of the Matlab process that created the Matlab process where the function is executing. Returns zero for the first Matlab process.
`pmhostname`	Returns the name of the host from an id.
`Pmlistvm`	Can be used to lists the ids of all Matlab processes of a specific VM

Table 1. Commands for retrieving information on the PVM Matlab process ids.

Adding and Deleting Hosts

After `pmopen` has been successfully effectuated the system can be further expanded or shrunk. New hosts can be added or existing hosts (with all their Matlab processes) can be removed. This is mainly controlled on a PVM level through the commands `pvm_addhosts` and `pvm_delhosts`.

Adding and Deleting Matlab Processes

Matlab processes can be added and deleted from the system after the parallel session has been initiated by `pmopen`. An error will be produced if a Matlab process is attempted to be started on a host that does not exist in the parallel Matlab system. Hosts can be added as described above. `pmspawn` is used to add Matlab processes. The parameters taken by `pmspawn` correspond to the four entities of the third argument of `pmopen`. Indeed, the arguments are exactly the same. The first argument specifies the host(s) on which Matlab processes will be started. The second argument specifies how many Matlab processes to start on each one of the by argument one specified hosts. The third argument specifies to which virtual machines each of the Matlab processes will belong, and the fourth argument specifies the files that will be used to store the screen output of the processes running in background. Examples of how these works can be found in the section on how to start a parallel Matlab session, as well as in the Matlab help for `pmspawn`. A last argument can also be given to determine if the command should block and wait for the started Matlab processes to acknowledge their successful startup.

When specifying hosts this can be done in several ways. Hosts can be specified to be chosen by `pmspawn` itself or if a specific hostname is given that host will be used for the spawning. If an asterisk ("*") is given the host will be determined by `pmspawn`. For example the user may choose to start two Matlab slaves on a certain host, then three more on hosts determined by `pmspawn` (these hosts must of course already be a part of the system). The hosts determined by pmspawn are chosen in a way intrinsic to PVM and may change with different versions of the PVM C libraries. Further information on how the hosts are chosen can be found in the PVM user documentation.

```
>>pmspawn({‘host1’ ‘*’},[2 3],0)
```

It is also possible to specify to spawn Matlab processes only on computers of a certain type of architecture or on the local host only.

There are two ways of deleting Matlab processes. The first is to, in its console, type `exit` followed by enter. The second is to execute `pmkill` from another Matlab process in the system.

Sending and Receiving Data

`pmsend` and `pmrecv` can be used to transfer data from one Matlab process to another. Data can be sent to one or several processes at the same time. The destination of the data is specified by providing the ids of the target Matlab processes. When sending the data a name can be specified, and when receiving the data it can be specified to only receive data with a specific name, or coming from a specific Matlab process. The `pmrecv` is automatically blocking, i.e. does not return the prompt before it has returned something. However, using a timeout argument it can be set up to be non-blocking. An example of sending and receiving Matlab data follows:

On process 262145:

```
aa =  magic(5);
pmsend ([262146 528289],aa)
```

On process 262146 and 528289 *aa* can thus be received by:

```
pmrecv(262145) % receive any data sent from process 262145
```

or

```
pmrecv
```

by itself if we know that no other process has sent data to the receiving process.

Evaluating on Other Matlab Processes

`pmeval` can be used as Matlab's `eval` but to evaluate expressions on other Matlab processes. It has the features of *try* and *catch* expressions so that one expression can be tried, and if an error occurs it will be ignored and the catch expression will be evaluated instead. The `pmeval` can set to be executed in *quiet* mode, which means that no output will happen on the remote console whatsoever, unless the user by for example omitting a semicolon forces the console to output the value of the expression. If the `pmeval` generates an error and the user has not specified a catch expression the error message will be returned to the issuing Matlab process with the name "EVAL_ERROR". An example that creates an error and retrieves the error message follows. The output of the evaluation (which in this case is an error message) is on the remote console, in addition to that the error message is retrieved to the issuing Matlab process.

```
>>pmeval(262146,'aa=ones(2,2);b=a(3,2)');
>>pmrecv('EVAL_ERROR')

ans =

Undefined function or vaiable 'b'
```

Note that the `pmrecv` is blocking and will not be returned until it has received the desired data. This can be avoided using a timeout for the pmrecv:

```
>>pmrecv(-1, 'EVAL_ERROR', 1)
```

The first argument in the above expression specifies that the input may come from any Matlab process. The second specifies the name of the variable to receive, and the third is the timeout value in seconds for trying to receive the data. If the receive function times out it will simply return an empty matrix.

A more advanced method for extern evaluation of expression is provided by `pmrpc`, where RPC stands for remote procedure call. When using this input to the function is sent to the executing Matlab process(es) and after the provided expression has been evaluated selected variables can be retrieved automatically. Just as `pmeval` it can be set up to print what it does on the remote console or not.

Extern and Interactive mode

In order to evaluate expressions on other Matlab processes or to execute remote procedure calls, the remote Matlab process must be ready to execute commands coming from other processes. When the Matlab process is ready to receive and automatically execute the received commands it is said to be in *extern* mode. When the Matlab process is first started it is always in extern mode. A Matlab process in extern mode is waiting for commands from other Matlab processes as well as from the local console through the keyboard. If the Matlab process in extern mode receives an evaluation request (through `pmeval` or `pmrpc`) it is executed unless it is already busy evaluating something else. If the Matlab process is not in extern mode when the request arrives the request is simply queued in the message queue and will be executed as soon as the extern mode is turned on. There is no limit to the size of the message queue – this is more described in the PVM user documentation. The extern mode is left, and the *interactive* mode entered, as soon as the user presses a key in the console of the Matlab process. Background processes can only be in extern mode.

There are two ways to recognise if a Matlab process is in extern mode or interactive mode. First, the user can look at the Matlab prompt of the process' console. If it begins with the id of the Matlab process it is in extern mode. It could look as follows:

```
262145>
```

Otherwise, if the prompt looks like a normal Matlab prompt, the Matlab process is in interactive mode. Second, the function `pmgetinfo` reveals if a given Matlab process is in extern mode or interactive. This command can be executed from any Matlab process to interrogate if a specified Matlab process is in extern mode or not.

Computation Resources – Virtual Machines

The Virtual Machines (VM) have two purposes. First, a virtual machine defines start-up parameters for a Matlab process. Second, all Matlab processes that are members of a virtual machine define a set of computational resources for dispatching – when dispatching, the Matlab processes to be used are not specified by the Matlab process ids, but by virtual machine id(s). Several virtual machines can coexist in the same parallel Matlab system since one might want to reserve a certain type of calculation for a certain ensemble of Matlab processes. The different virtual machines may also be defined in order to solve the same task on computers with work directories or where different initialisation of Matlab processes is desired.

The virtual machine is physically defined as a distributed object accessed by indices into a global data base. The attributes of the virtual machine are displayed in the following table:

Attribute	Possible values	Default
wd	string designating a file path	current working directory
prio	‘same’, ‘normal’, ‘low’	‘normal’
try	a Matlab expression that can be evaluated by eval	‘’ (empty string)
catch	a Matlab expression that can be evaluated by eval	‘’ (empty string)
runmode	‘bg’, ‘fg’	‘fg’

Table 2. The attributes of the *Virtual Machine*, with default values.

The attributes are mostly self explanatory however some things may need to be clarified. For the priority: specifying ‘same’ will give the Matlab process the same priority as the Matlab process spawning the new Matlab process. The ‘try’ expression specifies an expression that will be evaluated using `eval(try)` when a Matlab process is started. The ‘catch’ expression is the “catch” expression when evaluating using `eval(try,catch)`. The run-mode specifies whether processes are started in background or foreground with a console window. The Matlab process cannot switch between the different modes once it has been started. If it is defined in background a file can be defined to store the output that is normally sent to the Matlab console.

To create a virtual machine a Matlab *struct* must be created with the above shown attributes. The *struct* must then be passed to the constructor `vm`. The constructor can also take the string ‘d’ to create a default object. The default values can be seen in the previous table. The constructor will return an integer which is the index that can be used to access the virtual machine attributes. Now, when starting a Matlab process it can be chosen to belong to one or several virtual machines. The attributes of the first virtual machine will be used to start the Matlab processes. The other virtual machines to which a Matlab process belongs are just for creating different sets of Matlab processes. Note that once a virtual machine has been created it cannot be modified. This is because the information stored in it is used by the dispatcher to restart possible dead processes in the exact same way the were originally spawned. A virtual machine may be deleted using `vmdel` only if it has no member Matlab processes. A list of all virtual machines in the system is obtained through `vms`, and the attributes for a specific virtual machine can be obtained using `vmget`. A new virtual machine will always be given the lowest available index. If we have the virtual machines 0,1,2 and then delete “1” and add a new virtual machine it will take the newly vacant index “1”. All these “commands” are methods of the vm object and will be listed by typing “`methods vm`”.

A Matlab process can at any time join or leave different virtual machines. The commands for leaving or joining different Matlab processes can only be executed on the actual Matlab process itself. Thus, if it is desired to make a Matlab process join a virtual machine from another this must be done using `pmeval`. The following functions exist for handling virtual machines.

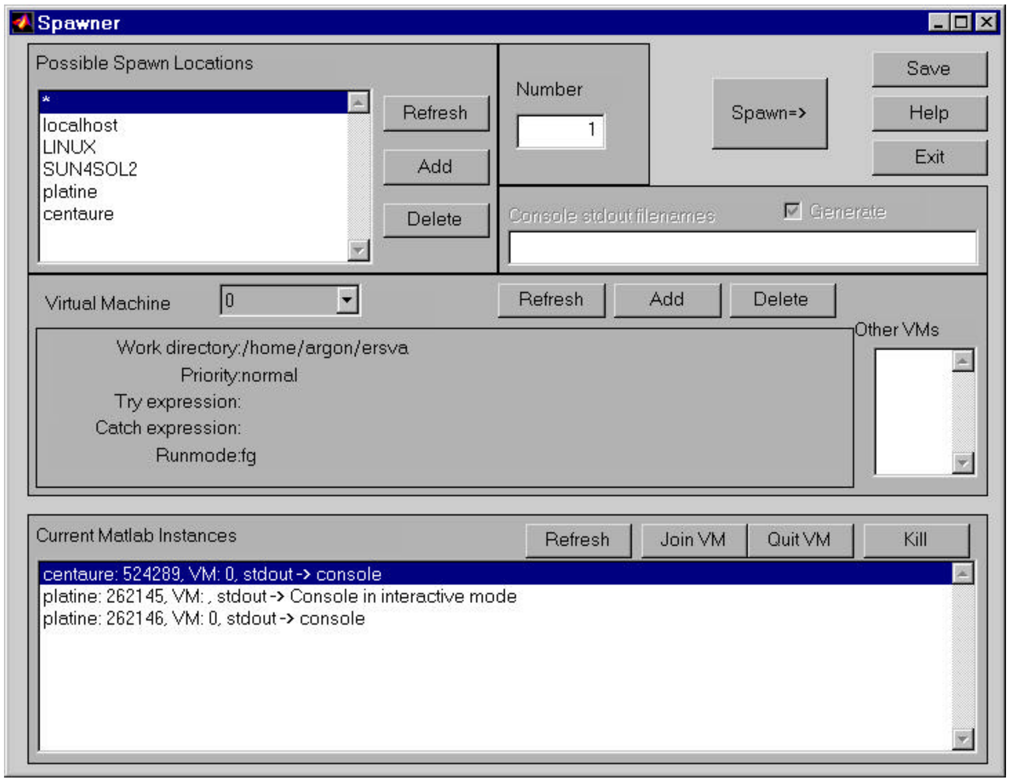
`pmjoinvm`	Let the executing Matlab process join a specific virtual machine
`pmquitvm`	Let the executing Matlab process leave a specific virtual machine
`pmlistvm`	List all virtual machines in the parallel Matlab system, or all the virtual machines that a specific Matlab process is member of
`pmmembvm`	List all member Matlab processes of a virtual machine
`pmmergevm`	Merge several virtual machines

Table 3. Commands for managing the memberships of *Virtual Machines*.

In the general and most common case only one virtual machine is created, with all the existing Matlab processes members of this virtual machine except for the Matlab process running the dispatcher. This way all resources are used to solve one problem in an environment that is similar on all computers in the network.

Modifying the System Using the Graphical User Interface

The graphical user interface is invoked with `pmcfg`. It can be used to open a parallel Matlab session as well as modifying an existing system. When used to open a session it will define a system consisting of the current host, a default virtual machine and no Matlab processes. The user can then add or remove hosts, virtual machines and Matlab processes by simple mouse clicks. The following figure displays the interface:



Work directory:/home/argon/ersva
Priority:normal
Try expression:
Catch expression:
Runmode:fg

centaure: 524289, VM: 0, stdout-> console
platine: 262145, VM: , stdout-> Console in interactive mode
platine: 262146, VM: 0, stdout-> console

Figure 3. Interface for configuring the parallel Matlab system.

The *possible spawn locations* list contain the names of all hosts currently in the system in addition to other possible choices for where to add Matlab processes. The adjacent add and delete button can be used to add or remove hosts. The highlighted entry of the list is where new Matlab process(es) will be spawned. The Virtual Machine popup menu is used to select which attributes will be used for spawning a new Matlab process. If the selected virtual machine specifies to add processes in background it is possible to choose to generate output filenames. Otherwise these can be entered manually the corresponding edit box. *Number* determines how many Matlab processes to spawn on the selected location and in the selected virtual machine. In addition, a spawned Matlab process can also be member of other virtual machines in the system. This is set by highlighting virtual machine ids in *other vms*. When all settings are set the *spawn* button is used to effectuate the addition of the Matlab process.

The lower part of the interface is merely used for displaying the different Matlab processes in the parallel Matlab system and to let these change their virtual machine memberships. Matlab processes can be deleted using the kill button. The list-box displays first the host of each Matlab process followed by the Matlab process id, and virtual machine ids. The first virtual machine id (if several) correspond to the one that will lend its attributes in case the Matlab process needs to be replaced during the dispatch. The last bit of information on the line is whether the Matlab process has a console or if not – to where its console output is directed.

Definition of a Function to Dispatch

In general a function can be perceived as a black box performing something on an input data thus producing an output.

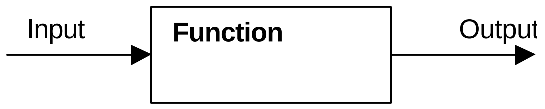


Figure 4. Function block diagram

A dispatchable function is different in that it describes a certain problem that is partitioned by the programmer and where these parts do not need to be executed sequentially. They may stem from a *for-loop* that calculates something by partitioning the problem and where the order of the execution of calculations is not important. Or, they may stem from an algorithm applied on ultrasound data from different patients.

Dispatching a dispatchable function can be seen as if several functions are queued to be evaluated. A function in this case corresponds to an arbitrary Matlab expression to evaluate. The functions, or expressions, in the dispatcher queue may all be the same, but operating on different data. Or, the data may be the same and the expressions evaluated on it different. The basic dispatchable function consists of a Matlab expression to be evaluated on different input data at each evaluation. In reality, some data may only be needed to be provided once, and is invariable throughout the calculations. This may be parameters, or a reference image, etc. To optimise the calculation and data transfer time these can be set to be distributed only once. On the other hand, there is data that needs to be provided at each evaluation of the expression. Denoting the former as *common* and the latter as *specific* a function can be visualised on the following page, where each *specific* input and output corresponds to an evaluation of the function on different data:

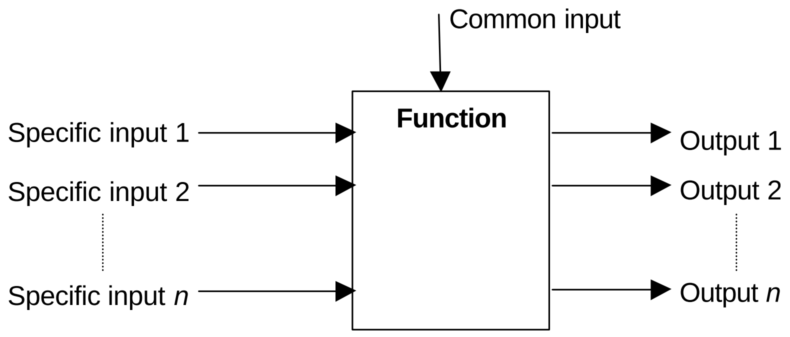


Figure 5. Illustration of a parallel function

A function where the function expression changes at each evaluation rather than the input data can be defined by creating arrays of this basic dispatchable function definition.

When executing a same function on large amounts of data it is often the case that this data is stored in different data files. Therefore a Matlab process evaluating a part of a bigger dispatched problem can get data directly from files instead of from the dispatcher.

A dispatchable function is defined by filling out the fields of provided function definition objects. These objects are accessed in the same way as *structs* in Matlab.

PMRPCFUN – Remote Procedure Calls

This is the most basic object among the function definition classes; indeed, as the name suggests it defines a remote procedure call (RPC), and not a complete dispatchable function. The RPC consists of a Matlab expression in form of a character array (which can be evaluated using `eval`), and the definition of the input as well as output arguments that should be used. These arguments will be automatically transferred to and from the target Matlab process of the remote procedure call. The *pmrpcfun* consists of the following fields:

`Expr`	An expression that will be evaluated on a target Matlab process
`Argin`	The name(s) that the input data will take on the target Matlab process
`Argout`	The name(s) of the output data that should be retrieved from the target Matlab process

Table 4. Attributes of the *pmrpcfun* object

The *pmrpcfun* contains one single method: `rpc`, which is used to effectuate the remote procedure call. An example of a simple RPC function and its execution would be:

```
r = pmrpcfun('a=b+1',{'b'},{'a'});
c = rpc(r,524289,10);
```

After this is executed in a parallel Matlab session containing a Matlab process with the id 524289 in *extern mode*, *c* will have the value 11. A more complex example can be found in the Matlab help for *pmrpcfun/rpc*. It should be noted that the user can also do a non-blocking remote procedure call, i.e. one that does not wait for output. This is simply done by not specifying the output:

```
rpc(r,524289,10);
…other code…
c = pmrecv('RPC_OUT'); % now, receive output from RPC!
```

It is important to grasp how the pmrpcfun works since it is the basis of all dispatches.

PMFUN – Function Definition For Basic Dispatchable Function

The basic dispatchable function is defined by a Matlab user object *pmfun* that inherits the *pmrpcfun*, i.e. its first three attributes are the same. A *pmfun* object can be created by inheriting an already existing *pmrpcfun* object, but it can also be constructed from scratch.

The *pmfun* objects can be described to contain three main parts. The first specifies what should be done at each evaluation of the function expression, i.e. which data to send, what name this data should take on the remote Matlab process, which data should be conserved after the expression has been evaluated, and how to store this desired data. This part is described by the following fields in the *pmfun* object: `expr`, `argin`, `argout`, `datain`, `dataout` and `blocks` where the three first ones are attributes inherited from *pmrpcfun*. The last attribute, `blocks`, is what describes the actual data for each evaluation scheduled by the dispatcher. `datain` and `dataout` contain references, or pointers, into this structure that is defined by a mx1 vector of *pmblocks* – one for each evaluation (see section about *pmblock*).

The second part of the *pmfun* object defines the initialisation of each Matlab process before the actual dispatching commences. The *common* data, i.e. the data that is provided to all Matlab processes only once for initialisation, is specified here. It is also possible to define an expression to be evaluated before the dispatching takes place as well as one that will be evaluated after the dispatch has terminated. This part is specified by the following fields in the *pmfun* object: `comarg`, `comdata`, `prefun` and `postfun`.

The third part of the *pmfun* object defines settings for the dispatch behaviour. At present there is only one setting: `singlemode`. It specifies whether several Matlab processes should be allowed to process the same sub-problem. This may be wanted in some cases for redundancy, or for sending the same sub-problems to several Matlab processes and only keep the result from the one that replied faster than the others. In the normal case this is however not wanted, especially when the results are saved directly to disk, because of the risk of one process partly overwriting the data saved by another.

Additionally the *pmfun* contains a `userdata` field that can be utilised for whatever purposes the user desires, e.g. comments about the function, etc.

The following table describes the meanings and valid values for all the attributes of the *pmfun* object:

`expr`	An expression that will be evaluated on a target Matlab process	
`argin`	The name(s) that the *specific* input data will take on the target Matlab process.	
`argout`	The name(s) of the output data that should be retrieved from the target Matlab process at each evaluation of the expression.	
`datain`	Definition of the *specific* data to send to the target Matlab process. `datain{1}` corresponds to the variable `argin{1}`, etc. These entries consist of character arrays that serve as references to the *blocks* field. *m* corresponds to the evaluation number.	
	`'GETBLOC(`*n*`)'`	Use the data in entry *blocks(m).src{n}* as indices to address a sub-matrix of input *n* to the dispatcher.
	`'LOADFILE(`*n*`)'`	Load variable from the file with the name found in *blocks(m).srcfile{n}*. Several variables could be loaded from the same file by referencing the same file *n* for each variable.

datain (advanced)*	'USERDATA(*n*)'	Get the data in *blocks(m).userdata{n}*
	'SRC(*n*)'	Get the data in *blocks(m).src{n}*
	'SRCFILE(*n*)'	Get the data in *blocks(m).srcfile{n}*
	'DST(*n*)'	Get the data in *blocks(m).dst{n}*
	'DSTFILE(*n*)'	Get the data in *blocks(m).dstfile{n}*
	'MARGIN(*n*)'	Get the data in *blocks(m).margin{n}*
dataout	Definition of where to store the *specific* data output from the target Matlab processes at each evaluation. `dataout{1}` corresponds to the variable `argout{1}`, etc.	
	'SETBLOC(*n*)'	Use the data in the entry *blocks(m).dst{n}* as indices to address a sub-matrix of output *n* of the dispatched function.
	'SAVEFILE(*n*)'	Save variable to the file with the name found in *blocks(m).dstfile{n}*.
blocks	This is a *m*x1 vector of *pmblocks* that define the specific data. *m* corresponds to the number of evaluations – or blocks – that will be executed by the dispatcher.	
comarg	The name(s) the *common* input data will take on the target Matlab process. This data will be provided to the Matlab process before the actual dispatching commences. It can also be distributed to chosen Matlab processes using the method `setcommon`.	
comdata	The actual *common* data or pointers to it. The pointers can take the following values where *n* is a reference to the input of the dispatcher. `comata{1}` corresponds to the variable `comarg{1}`, etc.	
	'INPUT(*n*)'	Use the *n*:th input to the dispatcher.
	'LOAD(*n*)'	Load variable from the file with the name found in the *n*:th input to the dispatcher.
prefun	A Matlab expression that will be evaluated on every target Matlab process before the dispatching is started. This expression will be executed *after* the common data has been distributed.	
postfun	A Matlab expression that will be evaluated on every target Matlab process after the dispatching has terminated.	
userdata	For the user's convenience.	
singlemode	A boolean switch. 1 (default) means that only one Matlab process is allowed to process one particular sub-problem at a given time. 0 means that several Matlab processes can work on the same sub-problem at the same time.	

Table 5. Attributes of the *pmfun* objects

*The grey fields are meant for the more advanced user or for debugging. The normal use of the dispatcher does not require this functionality.

The *pmfun* object has the following methods: `rpc`, `dispatch`, `setcommon`, `addspecinput`, `delspecinput`, `addcominput`, `delcominput`, `addoutput` and `deloutput`.

`rpc` is a remote procedure call. It can be used to partly try the function definition by evaluating one sub-problem on a specified Matlab process since it effectuates its call according to the function definition. `setcommon` can be used to initiate chosen Matlab processes with the common data specified by the attributes of *pmfun*. `dispatch` is of course the actual dispatch method. The last methods are used for updating the input and output definitions of the *pmfun*. These are the recommended way for doing this because they maintain the consistency and correctness of the *pmfun*. More information on the use of these can be found in their respective Matlab help files.

PMBLOCKS

The *pmblock* object specifies the actual in and out data for each evaluation. The *pmfun* and *pmjob* objects each contain a *mx*1 vector of *pmblocks*, where *m* is the number of evaluations to be dispatched and executed. It can contain filenames, matrix indices, timeout values, or user defined data. What is later done with these attributes is decided in the function definition. Its use is flexible and the attributes could contain any values and be used in any way. Below follows a table describing the recommended use for normal behaviour. There are two methods for easily defining filenames and indices to be put into vectors of *pmblocks*: `createfnames` and `createinds` respectively.

`src`	Contains character arrays that describe how to construct indices into an arbitrary Matlab matrix. This is used via the '`GETBLOC(`*`n`*`)`' command in *pmfun.datain* to access a sub-matrix of input *n* to the dispatcher. *n* is also the index into the `src` field and has thus a double use. It can also be accessed by '`SRC(`*`n`*`)`' in *pmfun.datain* to access the actual value of the *n*:th `src` entry. Index descriptors can be created using the function `createinds`.
`dst`	Contains character arrays that describe how to construct indices into an arbitrary Matlab matrix. This is used via the '`SETBLOC(`*`n`*`)`' command in *pmfun.dataout* to store output data in a sub-matrix of output variable *n* from the dispatcher. *n* is also the index into the `dst`

	field and has thus a double use. It can also be accessed by '`DST(`*`n`*`)`' in *pmfun.datain* to access the actual value of the *n*:th `dst` entry as input to an evaluation. Index descriptors can be created using the function `createinds`.
`srcfile`	Contains filename(s) that can be used by the command '`LOADFILE(`*`n`*`)`' in *pmfun.datain* to make the target Matlab process load its input from a file instead of getting it from the dispatcher for each evaluation. The actual filenames could also be sent to the target Matlab processes using '`SRCFILE(`*`n`*`)`' in *pmfun.datain*.
`dstfile`	Contains filename(s) that can be used by the command '`SAVEFILE(`*`n`*`)`' in *pmfun.dataout* to make the target Matlab process save its output to a file instead of returning it to the dispatcher after each evaluation. The actual filenames could also be sent to the target Matlab processes as input using '`DSTFILE(`*`n`*`)`' in *pmfun.datain*. Lists of filenames from directories or from a file containing filenames can be created by `createfnames`.
`margin`	Reserved for future use.
`timeout`	The evaluation of the sub-problem described by this *pmblock* will be interrupted and the sub-problem rescheduled to another Matlab process after the time specified here. Default is Inf.
`userdata`	Can be used at the user's convenience. Can be accessed as input to an evaluation by using the command '`USERDATA(`*`n`*`)`' in *pmfun.datain*.

Table 6. Attributes of the *pmblock* object.

The *pmblock* object contains the methods `setattr`, `getattr`, `setbloc` and `getbloc`. These can be used for efficiently modifying the contents of a vector of *pmblocks*.

PMJOB – Definition for Vectors of Function Definitions

The *pmjob* object inherits the *pmfun* and encapsulates also the input and output of the dispatched function, as well as a specification on which computer resources to use for the execution, i.e. which virtual machine(s) to use. By encapsulating this information into the *pmjob* object several functions to dispatch can be described by a vector of *pmjob* objects and then dispatched simultaneously. This is good if the parallelism to be achieved consists not only of distribution of data but also distribution of functionality. The following table describes the attributes of the *pmjob* object. Since most of them are inherited from the *pmfun* object more detail can be found in the section describing the *pmfun* object.

`expr`	An expression that will be evaluated on a target Matlab process
`argin`	The name(s) that the *specific* input data will take on the target Matlab process.
`argout`	The name(s) of the output data that should be retrieved from the target Matlab process at each evaluation of the expression.
`datain`	Definition of the *specific* data to send to the target Matlab process. `datain{1}` corresponds to the variable `argin{1}`, etc. These entries consist of character arrays that serve as references to the *blocks* field. *m* corresponds to the evaluation number.
`dataout`	Definition of where to store the *specific* data output from the target Matlab instances at each evaluation. `dataout{1}` corresponds to the variable `argout{1}`, etc.
`blocks`	This is a *m*x1 vector of *pmblocks* that define the specific data. *m* corresponds to the number of evaluations – or blocks – that will be executed by the dispatcher.
`comarg`	The name(s) the *common* input data will take on the target Matlab process. This data will be provided to the Matlab process before the actual dispatching commences. It can also be distributed to chosen Matlab processes using the method `setcommon`.
`comdata`	The actual *common* data or pointers to it. The pointers can take the following values where *n* is a reference to the input of the dispatcher. `comata{1}` corresponds to the variable `comarg{1}`, etc.
`prefun`	A Matlab expression that will be evaluated on every target Matlab process before the dispatching is started. This expression will be executed *after* the common data has been distributed.
`postfun`	A Matlab expression that will be evaluated on every target Matlab process after the dispatching has terminated.
`userdata`	For the user's convenience.
`singlemode`	A boolean switch. 1 (default) means that only one Matlab process is allowed to process one particular sub-problem at a given time. 0 means that several Matlab processes can work on the same sub-problem at the same time.
`vm`	Describes which virtual machines that are used as computation engines when dispatching this function. Default is 1.
`input`	Input to the dispatcher in the form of a cell array.
`output`	Output from the dispatcher for this dispatched function. A cell array with one cell for each output variable.

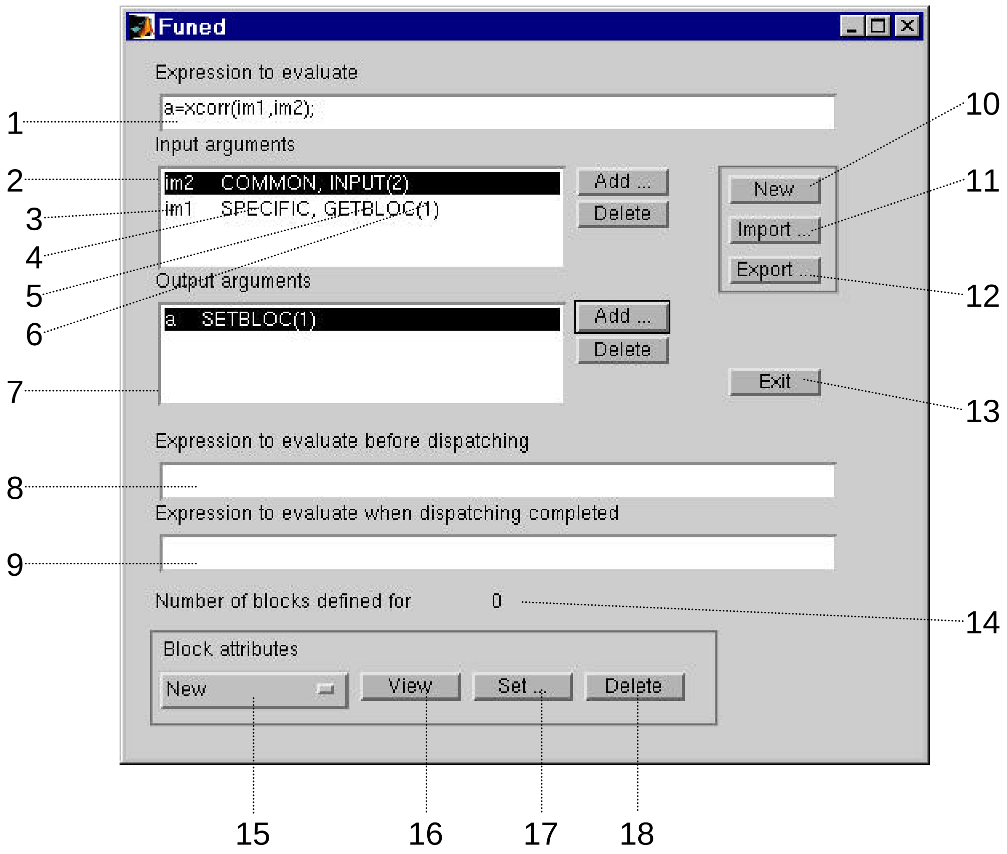
Table 7. Attributes of the *pmjob* object.

The *pmjob* object has the same methods as the *pmfun* object.

Using the Function Editor FUNED

The function editor *funed* has been developed to provide the user with an easy-to-use interface for completely defining a function to dispatch. It can be used to define dispatchable functions that read their input from files, get their input from the master Matlab process (i.e. the process where the dispatcher is executing), or a combination of both. *Funed* has been made in such a way that the user should not have to be concerned with the structure of the underlying *pmfun* or *pmjob* object. Instead, indices are automatically updated and the data pointers stored in their proper locations. The user can thus simply enter all the desired inputs and outputs to the function and afterwards read how to pass the function to the dispatcher.

The *funed* interface can be seen below and a description for each numbered item is to be found on the following pages.



```
a=xcorr(im1,im2);
```

```
im2   COMMON, INPUT(2)
im1   SPECIFIC, GETBLOC(1)
```

```
a   SETBLOC(1)
```

Number of blocks defined for 0

Figure 6. *FUNED*, used for editing parallel function definitions

1. The “Expression to evaluate” is the expression that will be evaluated on the different Matlab slave processes for each “block” that is defined for this function. The “blocks” describe which data that should be used for each evaluation. This expression can thus be seen as a function to be evaluated on all the data described by the blocks.

2. The input arguments should describe all the data that the above expression, as well as the expression in 8, require to be evaluated. Highlighted input arguments can be deleted by pressing the adjacent delete button. When pressing the add button the dialogue box to the right of this text will pop up. A variable name must be entered in the edit box and must be different from other already existing variable names among the already defined input arguments (there is of course no purpose in providing two inputs to a variable with the same name since one of the will overwrite the other).

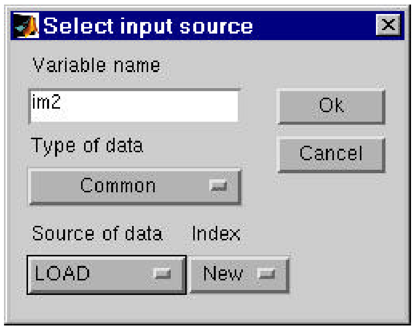
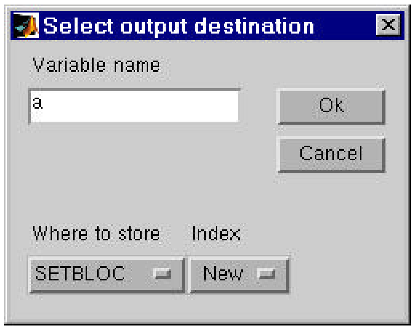


Figure 7. GUI for defining inputs of a parallel function

There are two types of data: *common* and *specific*, and depending on which one is chosen the source of the data can have different values. A brief description of the values follows, but these are further described in the section describing the *pmfun* if more specific information should be desired. A *common* variable is a variable that will be provided only once to each slave Matlab process to be used. The data can either be taken directly as the input (*input*) or loaded from a file whose filename is provided by the input (*load*). The *index* chosen in the popup list corresponds in these cases to the index of the input data to the dispatcher, or the input field of the *pmjob* object – i.e the actual data to send or the filename of the file to load will be provided as input to the dispatcher and the index specifies which input this variable comes from.

A *specific* data is different for each evaluation so the *source of data* is in this case a pointer into the block definition of the function (see 15-18). Possible values are *getbloc*, *loadfile* or direct pointers into the fields of the block definition. *Getbloc* allows the user to access a sub-matrix of the input to the dispatcher, whereas *loadfile* allows the user to make the slave Matlab process load its input from a file. The *index* corresponds here to an index into the pmblock structure and the *source of data* determines which field of the pmblock structure. For example *index* 2 together with *loadfile* would specify that the filename of the file to load is found in the source filename index 2 entry, as specified in the *pmblock*. A special case is the *getbloc* value (as *source of data*), whose index corresponds at the same time to the input index of the dispatcher input (as for *common* data). This is why *getbloc* always has lower indices than the *common* data. The user does however not need to worry about this since it is automatically managed by *funed*. When the OK button is pressed an interface for creating the corresponding *pmblock* attributes pops up. These can also be set at a later stage (see 17), however it is recommended that the user creates them as the input variables are added since this may avoid confusion. See the two following sections (*Creating Indices for Sub-Matrices* and *Creating Filenames for Input/Output Arguments*) for information on how to use these interfaces boxes.

3. Name of variable. Should be different from other variable names among the input arguements.
4. Specifies if the data is *common* (given at startup only) or *specific* (different for each evaluation and defined by the function blocks).
5. Defines the source of the data.
6. Index to the data. This has different meanings for *common* and *specific* data, see 2.
7. The output arguments describe which variables to retrieve after each evaluation at a Matlab process, and where to store this result. Highlighted arguments can be deleted by pressing the adjacent delete button. Pressing the add button invokes the dialogue box displayed to the right. The *where to store* field can either be *setbloc* or *savefile*. In the former case the data will be stored as a sub-matrix of output argument number *index*. *Index* is also the index into the *dst* field of the function blocks, which specifies the sub-matrix position. Using *savefile* the variable will not be


Select output destination

Variable name: a

Where to store: SETBLOC

Index: New

Figure 8. GUI for defining output variables

retrieved to the dispatcher Matlab process, but instead be saved directly from the slave process using the filename provided by the function *blocks* field *dstfile{index}*. As for when adding the input variables an interface for creating the corresponding *pmblock* attributes pops up when the OK button is pressed. The block attributes can also be set at a later stage, however it is recommended that the user creates them as the input variables are added (see (17) and the two following sections).

8. The *expression to evaluate before dispatching* will be evaluated once on all slave Matlab processes after the common data has been distributed, but before the actual dispatching commences.
9. The *expression to evaluate when dispatching completed* will be dispatched once on all slave Matlab processes, after the dispatching of all sub-problems has been executed for all functions queued.
10. *New* clears the interface fields.
11. *Import* allows the user to edit a *pmfun* object that already exists in the user's workspace. This can also be done by calling *funed* again with the desired *pmfun* object as an argument.
12. *Export* allows the user to return a function definition as a *pmfun* object to the user's workspace.
13. *Exit* discontinues *funed* after verifying that this is really what the user wants.
14. This number displays how many function blocks that are defined for the function. The function blocks define the data for each evaluation of the *expression to evaluate*. Once this number displays another number than zero, the other block attributes in the function blocks must also be created to the same number.
15. This dropdown menu displays all block attributes defined for the existing blocks. The list will always contain *New*, which allows the user to create a new attribute by pressing the *set* button (17). Note the number of attributes must be consistent for all attributes.
16. The *view* button displays the – by button 15 selected – attribute for all blocks. The result is displayed in the Matlab console window.
17. The *set* button can be used to change the contents of an attribute or to create a new attribute. The action to be performed is specified by the adjacent dropdown menu

(15). If this displays *New* the dialogue box to the right will appear. It allows the user to choose an attribute to add. Depending on the choice, two different dialogue boxes can follow. These are further described in the following pages. One allows the user to create filenames for the *srcfile* or *dstfile* field of the block. The other allows the user to create indices into sub-matrices of the input or ouptut defined for the function to dispatch. This is for putting in the *src* or *dst* field of the block. Both of these dialogue boxes can remain open and be used for creating several attributes. Choosing another attribute value in the popup menu and then clicking the *set* button changes the output target for the attribute creator dialogue boxes. Choosing another attribute than the above mentioned brings up a dialogue box that the user can use to enter an expression that will be evaluated in the user's workspace. The result of this evaluation must be a vector – for *userdata* a vector of cells, and for *timeout* a vector of doubles. Each entry in the vector will be placed in one of the *pmblocks* in *pmfun.blocks* – the first entry in the first block, etc. Since the entered expression is evaluated in the user's workspace the user can access any previously created variables. This dialogue box will cannot remain open for affectation of its values to several attributes.

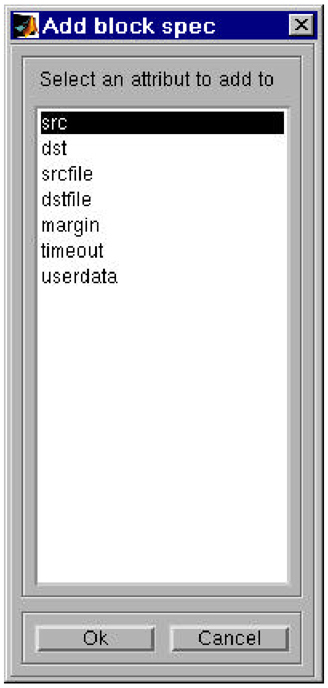


Figure 9. Attribute selector for set button

18. Pressing the *delete* button deletes the chosen attribute in the dropdown menu (15). If all attributes are deleted the whole block array will be deleted and the number of blocks defined, as shown by 14, will become 0. Now, new attributes can be created into an arbitrary number of blocks.

Creating Indices for Sub-Matrices

This interface is reached via the block attributes section of the *funed* interface (17). The indices created with this interface can be used from the *getbloc* or *setbloc* command in *pmfun.datain* and *pmfun.dataout* to access sub-matrices of the input or the output respectively. The indices created can be used to access an arbitrary Matlab cell or normal matrix (including character arrays). Each index entry created corresponds to one sub-matrix and will be put in one function block. All function blocks contain all together the indices to the complete matrix.

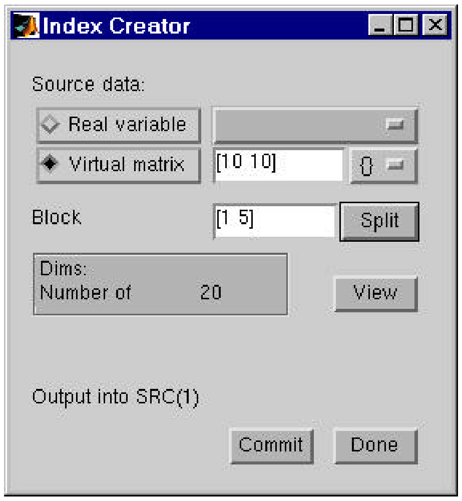


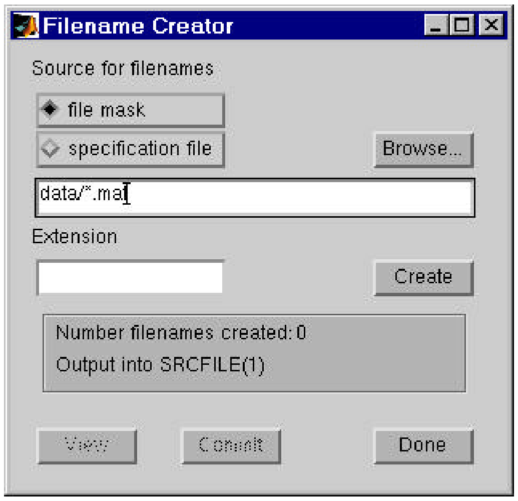
Figure 10. Index creator GUI

The interface can be seen to the right. The first thing to do when creating the indices is to define what kind of data structure the original data comes from and what the size of the sub-matrices (or blocks) should be. The source data could either be a real variable from the user's workspace, or it could be a "virtual matrix", i.e. a phoney matrix created that has the same characteristics as the real source data.

The virtual matrix could either be created as a cell matrix, i.e. it should be indexed with {}, or it could be created as a normal matrix, i.e. one that is indexed with normal parenthesis: (). Any dimension of matrices can be created: 1D, 2D, 3D, etc. When the dimension of the sub-matrices (blocks) has been determined the button *split* can be used to create the indices. When this is done the number of sub-matrix indices created is displayed in the darker box. Pressing the *view* button shows the created indices in the Matlab console. Once the user is content with the indices they can be committed to the field that was originally chosen to be updated or created (see 17). Indices can only be committed to the block-structure if the number of sub-matrices corresponds to the number of blocks defined (see 14). If commit is not pressed the created indices will not be transmitted to the block definition structure of the *pmfun* object. When the user has finished the *done* button can be used to exit. Note that the dialogue box can stay open for the edition of several attributes, by selecting these in *funed* (16). Overlapping blocks cannot be created.

Creating Filenames for Input/Output Arguments

This interface is reached via the block attributes section of the *funed* interface (17). The filenames created with this interface can be used from the *loadfile* or *savefile* command in *pmfun.datain* and *pmfun.dataout* to load and save the input or the output respectively. The filenames created can denote files containing more than one variable, only the desired variables will be read. The interface can be seen on the right side of the page.



data/*.mat

Number filenames created: 0
Output into SRCFILE(1)

Figure 11. Filename creator GUI

The filenames can be created either from an ASCII file containing a filename on each line, or from a search pattern. The former is in the interface denoted "specification file", and the latter "file mask". Using the *browse* button a specification file can be looked for and found easily.

Any text typed in the *extension* field will be added at the end of the filename – before the "`.mat`" extension. This is useful for example when creating output filenames that should differ from the input filenames.

When the *create* button is pressed the current settings are used and the result is temporarily stored in a local variable until the *commit* button is pressed. As soon as the create button has been pressed once the local variable contents can be viewed by pressing the *view* button at the down-most left corner of the interface. The interface can remain open for the creation of several filename attributes. The destination attribute is chosen in *funed* (see 15).

Dispatching

Configuration

The dispatcher has a number of different parameters that can be configured. The dispatcher will "remember" the settings from one call to another as long as it is not cleared from memory (by `clear dispatch`, `clear all`, etc.). The settings can either be set by calling the dispatcher method with a "set configuration" request, or parameters can be passed together with the actual dispatch request. The current settings of the dispatcher can be retrieved as follows:

```
>> conf = dispatch(pmjob,'getconfig')
```

This returns the configuration in a *struct* with the fields described in the following table:

Parameter	**Possible values**	**Description**
`gui`	Boolean (0|1)	Turns on or off the graphic user interface of the dispatcher. This must be on to dynamically change the parameters of the dispatch, e.g. add or delete Matlab processes, turn on and off logging, etc.
`saveinterv`	Double: 1 to Inf	The interval with which the state will be saved. Saving to often may seriously decrease the performance if the files to save are of considerable size.
`statefile`	Valid filename	Make sure that this specifies a file at a location where there is enough space to store the state.
`debug`	Boolean (0|1)	Turns debugging mode on or off, where on means that the commands executed on the Matlab worker processes will display they do.
`logfile`	Valid filename, 'stdout' or 'stderr' or empty	If a filename is chosen, make sure that there is enough space left on the drive where it is written. Specifying stdout or stderr sends the log to the standard output or standard error. Providing an empty array cancels logging.

Table 8. Parameters and settings for the dispatcher.

In the same way, the parameters can be set by providing the dispatcher with a Matlab *struct* with the above fields succeeding the argument 'setconfig' to the dispatcher. This will set the parameters and return immediately from the dispatch method. Additionally, each parameter can be set individually by listing parameter names, each followed a parameter value, for example:

```
>> dispatch(pmjob,'setconfig','gui',0,'debug',1)
```

This can also be done directly when executing the actual dispatch, for example:

```
>> dispatch(my_pmjob,[],'statefile','/tmp/my_state.mat')
```

The empty array ([]) means that we are not starting from a saved state.

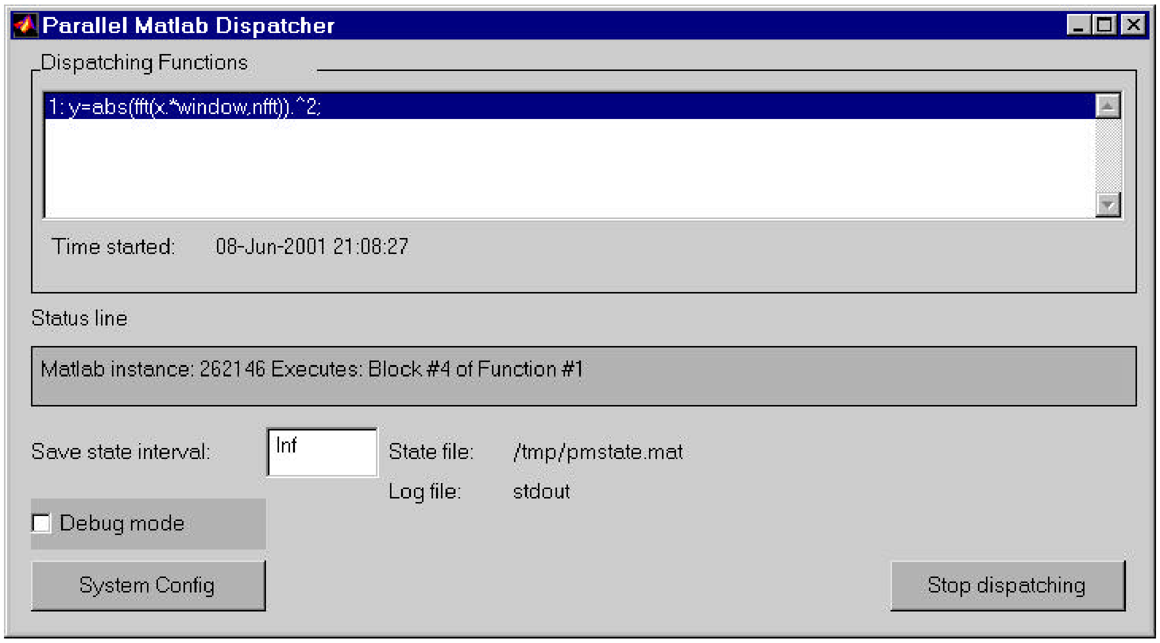
Information about Ongoing Dispatching through Log and State Files

If no graphic user interface is used to visualise and control the dispatching, the current state can still be retrieved by examining the log file. The Unix command *tail* is particularly useful for following the changes to the log file (see the Unix man file for how to use it). The log file begins with information about the function(s) that are to be dispatched. This is followed by the different events in the dispatcher, such as what part of the problem is sent to which Matlab process. Any logged information about the parallel Matlab system, such as dead or reinstated Matlab processes, is preceded by "sys" which is useful for quickly finding such information in a larger log file (e.g. with the Unix utility *grep*).

Partial results will be stored in the state file, if saving of temporary states is enabled (defining the `saveinterv` parameter of the dispatcher to another value than Inf). The state file contains a Matlab *struct* variable named *state* which contains a task queue (`state.taskq`) defining which functions and blocks that have *not yet been entirely processed*. The task queue is a *k*x2 matrix where *k* is the number of non-terminated tasks. The first column contains indices into a vector of *pmjob* objects to dispatch. The second column describes the index into the *pmblock* of the particular evaluation scheduled. For example, if a dispatch is being executed on a single *pmjob* object (or a *pmfun*) the first column will only contain ones, and the second will contain the remaining *pmblocks* to evaluate for the function. The variable *state* also contains information on possible errors that may have occurred so far, as well as the execution time so far. In addition to this there is of course also the partial output that has been calculated so far.

Using the Graphic User Interface of the Dispatcher

The dispatcher can be executed with an accompanying graphic user interface that allows the user to monitor the dispatching, as well as dynamically update the system by adding or deleting Matlab processes and changing dispatch parameters. It is turned on by turning on the `gui` parameter of the dispatcher configuration either prior to or when starting the dispatcher. The graphic user interface will automatically close once the dispatching has terminated. In addition to the dispatcher GUI there is also a progress in a separate window during the dispatching.



1: y=abs(fft(x.*window,nfft)).^2;

Time started: 08-Jun-2001 21:08:27

Matlab instance: 262146 Executes: Block #4 of Function #1

Save state interval: Inf

State file: /tmp/pmstate.mat

Log file: stdout

Figure 12. The parallel Matlab dispatcher GUI.

- The status line shows the current action of the dispatcher. Every time a sub-problem is sent to a slave it will be shown here.
- The listbox with the title "Dispatching Functions" shows the *expr* field of all the *pmfun* functions scheduled for dispatching.

- A number between 1 and Inf can be entered in the “Save state interval” to set how often the state should be saved.
- Debug mode can be turned on and off to let the slave processes display the expressions they are evaluating or not.
- The stop dispatching button is evidently used for discontinuing the dispatching. If it is pressed the dispatching can still resume from the last saved state.
- The Parallel Matlab System graphical user interface is accessible through the *System Config* button. This allows the user to dynamically update the system during dispatch. Note that the system configuration window can also be started prior to dispatching and used for dynamically changing the system, even without using the dispatcher graphical user interface.

Dynamically Modifying the System

The current system can be modified at any time by, from the dispatcher graphical user interface, pressing the *system configuration* button. This invokes the same user interface as the command `pmcfg`, which is described in the section on the parallel Matlab session.

If the dispatching is done without the graphical user interface, the parallel Matlab system can still be modified. This is done by starting a new Matlab process from the Unix prompt. Then typing `pmopen([],[],[])` at the new Matlab process’ prompt will make it join the current parallel Matlab system and `pmcfg` or other previously methods can be used to update the system. The newly started Matlab process does not belong to any virtual machine by default, but can be made to join the desired virtual machine so as to be used as a slave process.

Interrupting the Execution

If the graphic user interface is running while dispatching the “stop dispatch” button can be used to discontinue the dispatching. Another way would be to press control-c in the Matlab console window. This is however not recommended since it has been known to spontaneously cause crashes of the PVM session, in which case the complete Matlab and PVM session must be closed and started over.

Fault Tolerance

Errors During Dispatch

If a parts of a distributed calculation generates errors it is not for sure that all calculations distributed will. An error may for example be generated because a certain Matlab process was not successfully initialised to use some function in a local library. Errors may also occur if a dispatched evaluation is provided with indices that exceed the source matrix limits. Errors are logged into the log file so as to be easily detected. The error message is also stored in the state file together with information on which task (i.e. which function and block) and Matlab process that generated it.

Timing Out Evaluations

Each scheduled evaluation can be set to be interrupted and rescheduled after a certain time. This is done by specifying a timeout value in the *pmblock* for that particular evaluation. A convenient way to set the timeout value is to use the `setattr` method of the *pmblock*.

The Death of Matlab Slave Processes

If a Matlab process should die while executing it will be restarted if possible, and the processed sub-problem will be rescheduled. The newly started Matlab process will automatically belong to the same virtual machine(s) as the one it is replacing, it will also have the same working directory as the one it is replacing had when it was created. At its startup it will evaluate the *try* with *catch* expression that its predecessor used at its startup. The common data will be sent to the newly created Matlab process and the pre-processing expression will be evaluated.

Resuming an Interrupted Execution

If the Matlab process running the dispatcher program should die, the dispatch has to be manually restarted from its last saved state. The name of the state file is given as a configuration parameter to the dispatcher, as is the interval at which the state will be saved. The saved state file contains a variable called “`state`” that can be given to the dispatcher. An example of setting up the dispatcher to save its dispatch state at every 100 evaluations to the file “/tmp/strain_test.mat” follows:

```
>> dispatch(pmjob,[],’statefile’,’/tmp/strain_test.mat’,...
’saveinterv’, 100);
```

This setting will be kept in memory as long as the dispatcher method is not cleared from memory. Resuming the dispatching from a previous state can be done as follows:

```
>> load /tmp/strain_test.mat
>> dispatch(my_job, state);
```

The variable *my_job* must of course contain the same input data as it did when it was originally dispatched.

The Parallel Matlab Toolbox and PVM

Matlab and PVM

The parallel Matlab toolbox is based on the PVM message passing libraries. These libraries are written in C and are accessed in Matlab through a MEX file. The MEX file is a compiled C file accessible from Matlab. For each PVM function in Matlab there is a Matlab function that does nothing but transmit the request to the MEX-file. The user is free to use the PVM functions for any purpose, but note that by doing so in an unforeseen way the functionality of the dispatcher may be harmed. Should this happen, just close and restart the complete session. To completely close the PVM session, should it still be open after Matlab has been closed, start PVM from the unix prompt by typing PVM. Now, typing the command `halt` will kill any remaining PVM processes in the system. To find out more about the PVM commands type `help pvm` at the Matlab prompt.

PVM Error Codes

At some occasions cryptic error codes may be given as a result of an operation. These are generated by the underlying PVM libraries with the following meanings.

Error message	**Error code**	**Meaning**
PvmOk	0	Success
PvmBadParam	-2	Bad parameter
PvmMismatch	-3	Parameter mismatch
PvmOverflow	-4	Value too large
PvmNoData	-5	End of buffer
PvmNoHost	-6	No such host
PvmNoFile	-7	No such file
PvmDenied	-8	Permission denied
PvmNoMem	-10	Malloc failed
PvmBadMsg	-12	Can't decode message
PvmSysErr	-14	Can't contact local daemon
PvmNoBuf	-15	No current buffer
PvmNoSuchBuf	-16	No such buffer
PvmNullGroup	-17	Null group name
PvmDupGroup	-18	Already in group
PvmNoGroup	-19	No such group
PvmNotInGroup	-20	Not in group
PvmNoInst	-21	No such instance
PvmHostFail	-22	Host failed
PvmNoParent	-23	No parent task
PvmNotImpl	-24	Not implemented
PvmDSysErr	-25	Pvmd system error
PvmBadVersion	-26	Version mismatch
PvmOutOfRes	-27	Out of resources
PvmDupHost	-28	Duplicate host
PvmCantStart	-29	Can't start pvmd
PvmAlready	-30	Already in progress
PvmNoTask	-31	No such task
PvmNotFound	-32	Not Found
PvmExists	-33	Already exists
PvmHostrNMstr	-34	Hoster run on non-master host
PvmParentNotSet	-35	Spawning parent set PvmNoSpawnParent
PvmIPLoopback	-36	Master Host's IP is Loopback

Table 9. PVM error codes and meanings.

Examples

In this section a number of different examples are given to illustrate some possible uses of the dispatcher. The code can also be found in the distribution in the *examples* directory. The main applications for which the toolbox has been developed use data structures and objects as provided by the biomedical department at *École Polytechnique de Montréal*. For general understanding they are not shown in this user guide, which is why these examples are primarily for illustration purposes on how to define dispatchable functions using the parallel Matlab toolbox objects. These examples are not primarily intended for showing that their parallel implementations are more effective than the sequential one.

Example 1 – Filtering of Large Images

Problem Description

This example shows the most straightforward, and easiest application of the dispatcher. Namely, to execute an algorithm on a large amount of data stored in files. In this simple, yet illustrative, example an averaging filter described by a matrix B is applied to different images using the Matlab `conv2` function. These images need to be stored with the same name in different binary Matlab files (.mat). All Matlab slaves will receive the filter matrix B as a common variable, i.e. only once. Then, each Matlab slave will load different images upon request and apply the filter. The resulting images will be stored in files with the same name as the input image files, but with an added extension so as to be distinguished from the input files. Since the Matlab convolution is very fast, there is no direct need to use the dispatcher for just a few files. However, if a more complex algorithm is used, or if a large amount of images should be treated, the dispatcher is very useful.

Parallel Matlab Code – Filtering of Large Images

```
% A number of images stored in files will be loaded and an
% avereging filter will be applied.

% Make sure that all matlab processes are in the same directory
pmeval(pmothers,'cd /home/argon/ersva/matlab/test')

% This is a file containing the filenames. Each of the files
indicated
% therein should contain a 2D intensity matrix with the name
'img'.
infiles = 'infiles.txt';

% The filter to apply:
B = repmat(1/100,10,10);

w = pmfun;
w.expr='img2=conv2(B,img);'; % what each slave should evaluate.
w.argin = {'img'};          % the input file should contain this
variable
w.argout = {'img2'};         % the output will be saved with this
name
w.datain = {'LOADFILE(1)'}; % Let slave process load its data.
w.dataout= {'SAVEFILE(1)'}; % and then save the result.
w.comarg = {'B'};           % the filter is same for all images.
w.comdata = {'INPUT(1)'};   % and will be given as input to the
dispatcher.

% create cell arrays containing the different filenames of the
input
innames = createfnames(1,infiles);
% create cell arrays containing the different filenames of the
output
outnames = createfnames(1,infiles,'_modified');

% create a pmblock structure describing the filenames in the
pmfun: w.
w.blocks = pmblock('srcfile',innames,'dstfile',outnames);

% Now, to dispatch the filtering of all images:
err = dispatch(w,1,{B},[],[],'debug',1);
```

Example 2 – Welch’s Power Spectral Density

Problem Description

This is an illustrative example of how to program a parallel function. This function calculates the power spectral density according to Welch’s method. Subparts of a signal is multiplied by a window, then the discrete Fourier transform is obtained by using the Matlab `fft` function. The final power spectral density is then obtained by adding the squared results together and normalising. Since the actual calculation time is relatively small compared to the transfer time of the data, the chances are small that this routine gains anything from being parallel. However, it is still included among the examples for two important reasons. The first is of course to illustrate how to make a parallel function. The second is to show that the user should not bother to try to make everything parallel. Matlab is already highly effective for this kind of calculation.

Sequential Matlab Code – Welch’s Power Spectral Density

```
function [y] = welch(x,nfft,window,noverlap)

x = x(:);
window = window(:);
nwin = length(window);
nx= length(x);
if nx < nwin
   x(nwin) = 0; % pad with zeros!
   nx = nwin;
end
nseg = fix((nx-noverlap)/(nwin-noverlap));

% calculate the power spectral density
y = zeros(nfft,1);
ind = 1:nwin;
for n = 1:nseg
   xw = window.*x(ind);
   ind = ind + (nwin - noverlap);
   y = y + abs(fft(xw,nfft)).^2;
end
y=y/(2*pi*nseg*norm(window)^2); % normalise
```

Parallel Matlab Code – Welch's Power Spectral Density

```
function y = pmwelch(x,nfft,window,noverlap)

x = x(:);
window = window(:);
nwin = length(window);
nx= length(x);
if nx < nwin
   x(nwin) = 0; % pad with zeros!
   nx = nwin;
end
nseg = fix((nx-noverlap)/(nwin-noverlap));

w = pmjob;
w.expr = 'y=abs(fft(x.*window,nfft)).^2;';
w.argin = {'x'};
w.argout = {'y'};
w.datain = {'GETBLOC(1)'};
w.dataout = {'SETBLOC(1)'};
w.comarg = {'nfft' 'window'};
w.comdata = {nfft window};

xover = zeros(nfft,nseg);
ind = 1:nwin;
tic
for n = 1:nseg
   xover(:,n) = x(ind);
   ind = ind + (nwin - noverlap);
end
toc

inds = createinds(xover,[nfft 1]);
w.blocks = pmblock('src',inds,'dst',inds);

w.input{1} = xover;
clear xover
err = dispatch(w,[],'gui',0,'saveinterv',Inf);

y = sum(w.output{1},2);
y = y/(2*pi*nseg*norm(window)^2); % normalise
```

Example 3 – Integration

Problem Description

This third problem addresses integration of a function over an interval where the function behaves in a drastically different way at one end of the interval and the other. An example is $sin(x^3)$ where the function fluctuates exponentially quicker for larger x. This makes methods such as Simpson's slow over the complete interval. By dividing the integration into a sum of sub-integrations the calculation can be made parallel.

Sequential Matlab Code – Integration

```
a = 0;
b = 12;
quadl('sin(x.^3)',a,b)
```

Parallel Matlab Code – Integration

This code takes advantages of the flexibility of the *pmfun*/*pmjob* objects by using the *userdata* field. This shows that with some thought the *pmfun* and *pmjob* objects can be used to the user's convenience for particular special needs of an algorithm.

```
%% define dispatch function
w = pmjob;
w.expr = 'x=quad(''sin(x.^3)'',a,b);';
w.argin = {'a','b'};
w.argout = {'x'};
w.datain = {'USERDATA(1)' 'USERDATA(2)'};
w.dataout= {'SETBLOC(1)'};

%% integrate on intervall a->b
a = 0;
b = 12;
numseg = 10;
d = (b-a)/numseg; % step size

w.blocks = pmblock('dst',createinds(ones(numseg,1),[1 1]));
c = b;
for n=1:numseg,
  w.blocks(n).userdata = {(c-d) c}; % store intervals in USERDATA
  c = c - d;
end
w.output = {};

err=dispatch(w,[],'gui',0,'debug',0,'saveinterv',inf,'logerr',…
             0,'logdisp',0,'logsys',0,'logtimeout',0);

integral = sum(w.output{1})
```

Installation Manual

Configuration

There are some files that should be copied or pasted into already existing files in the system. The first one is `cshrc.stub` that should be pasted into `.cshrc` in the users home directory. If this does not exist simply rename the file to `.cshrc`. The file contains a lot of settings accompanied by comments explaining what may need to be changed for the current system. The user must verify that the contents of this file corresponds to the user's system and Matlab version, etc. If this file is not set up correctly the user risk errors at compilation as well as when running the toolbox. Once the `.cshrc` is properly updated it can be executed using "`source ~/.cshrc`", or a new xterm can be started.

The files `startup_stub.m` and `finish_stub.m` should be pasted into the user's possibly already existing `startup.m` and `finish.m` files respectively. These *must be stored in the user's Matlab path*. If these files do not exist simply rename the "stub" files to the respective target filenames and make sure you store them somewhere in the user's Matlab path. If the user has a directory called "*matlab*" in his home directory this will automatically be a part of the user's Matlab path and could be a good directory to place these files.

The last file needed is the `pvmdefhosts.m`. This file defines which hosts that are part of the parallel Matlab system and available for running Matlab processes. The settings and options for this file can be found in the documentation for PVM. A basic setting contains a specification of PVM executable files (`*  ep=$PATH`), followed by one or several lines with one host name on each line:

```
ep=$PATH
hostname1
…
hostnameN
```

Compiling

Before running the toolbox it must be compiled. Make sure that the `.cshrc` has updated the environment variables (either by running in a newly started *xterm* or by using *source*). *From the pm directory* type “`make`” at the UNIX prompt. Some warnings may occur at compilation but do not worry about these. If no fatal errors occur the toolbox is ready to be started. It is normal that there are errors because of directories that already exist, these are however ignored by the make utility (this is displayed by a message such as “error ignored” from the make utility). “`make clean`” can be used to remove resting files after compilation to be able to compile again, e.g. for compiling on another platform.