MATHEMATICA®
PARALLEL COMPUTING TOOLKIT

UNLEASH THE POWER OF PARALLEL COMPUTING

WOLFRAMRESEARCH
www.wolfram.com

January 2005
Intended for use with *Mathematica* 5.0 and higher

Software and manual written by Roman E. Maeder

Product managers: Jeffrey Bryant and Lars Hohmuth
Project manager: Jennifer Peterson
Document quality assurance: Rebecca Bigelow, Julienne Davison, Marcia Krause, and Jan Progen
Software quality assurance: Rachelle Bergmann, Jay Hawkins, and Cindie Strater
Package design: Jeremy Davis, Megan Gillette, and Richard Miske

Published by Wolfram Research, Inc., 100 Trade Center Drive, Champaign, Illinois 61820-7237, USA
phone: +1-217-398-0700; fax: +1-217-398-0747; email: info@wolfram.com; web: www.wolfram.com

T5051 378428 0105.rcm

Table of Contents

Preface

Parallel Computing Toolkit (PCT) brings parallel computation to anyone with access to more than one computer on a network. It implements many parallel programming primitives and includes high-level commands for the parallel execution of operations such as animation, plotting, and matrix manipulation. This toolkit also supports many popular programming approaches such as parallel Monte Carlo simulation, visualization, searching, and optimization. The implementation of all high-level commands is in *Mathematica* source form and can serve as templates for building additional parallel programs.

PCT builds on *Mathematica*'s advanced symbolic programming language. It is written entirely in the *Mathematica* language and uses *Mathematica*'s standard *MathLink*® protocol to communicate between any number of *Mathematica* kernels. The kernels can run under any supported operating system including Unix, Linux, Windows, and Macintosh. Individual machines can be single- or multiprocessor PCs and servers connected through TCP/IP.

PCT supports all common parallel programming paradigms: shared or distributed memory; automatic or explicit scheduling; and concurrency, including synchronization, locking, and latency hiding. It also supports failure recovery. In the event of a network, hardware, or software failure, the affected computation is reassigned.

We gratefully acknowledge the support of Orion Multisystems, whose desktop cluster workstation DT-12 was used to perform most of the evaluations shown in the PCT documentation and examples.

Roman E. Maeder, October 2004

Changes in *Parallel Computing Toolkit* Version 2

New Functionality

`ParallelEvaluate[]` has been extended to become a flexible tool for implementing many structural operations (such as `Map`, `Apply`, `Cases`, `Select`, `Count`, `MemberQ`, `FreeQ`, `Inner`, `Outer`, and all associative functions) in parallel.

A new auxiliary command `ParallelDispatch[]` for sending different commands to different kernels is the basis for the re-implementation of `ParallelEvaluate[]` and `ParallelTable[]`.

Delayed definitions for shared variables are possible and cause the right sides to be evaluated on the remote kernels.

Support for different process queue models has been added, including user-defined ones. Standard queues provided are a FIFO queue (as in Version 1) and a new priority queue, as well as a faster unordered queue.

Processes can be queued with a user-defined priority that the scheduler takes into account when assigning processes to processors.

There is more extensive debugging support and saving of trace output for later analysis.

If no remote kernels are available, all evaluations happen sequentially in the master kernel.

A new command `RemoteNeeds[]` for loading packages on remote kernels has been added.

Enhancements

`ParallelEvaluate[]`, `ParallelMap[]`, and `ParallelTable[]` do a single dispatch on each remote kernel, taking relative processor speeds into account for optimal load balancing.

There are new commands `ParallelSum[]` and `ParallelProduct[]` to complement `ParallelTable[]`.

A new configuration variable `$RemoteUserName` for use in kernel launch command templates, such as `$RemoteCommand`, has been added.

`Send[`*kernel*`, `*cmd*`]` returns *kernel*, so you can use it as an argument of `Receive[]`.

`Receive[]` can take a *list* of kernels as the argument and waits for one result from each kernel while allowing for callbacks, such as shared variables.

`RemoteEvaluate[]` allows callbacks and therefore shared variables.

A proper data type for remote kernels with improved diagnostic print formatting is included.

Aborting remote kernels is possible, provided the *MathLink* device used for the kernel connection supports aborts. Resetting runaway kernels is possible in a wider set of circumstances, in many cases without an actual abort.

Newly launched kernels inherit all previously exported environments, loaded packages, and declared shared variables.

There is a new command `ClearShared[]` to unshare previously shared variables. The variables `$SharedVariables` and `$SharedDownValues` give the lists of currently shared variables and downvalues, respectively.

It is possible to extract a part of a shared variable and transmit only the requested part of the variable to the remote kernel, rather than its whole value.

Process scheduling has much less latency.

`Needs["Parallel`"]` sets up PCT by loading the required parts and autoloading the optional extensions (except for `Parallel`Commands`).

`ParallelEvaluate[Dot[...]]`, `ParallelEvaluate[Inner[...]]`, and `ParallelDot/ParallelInner` also parallelize tensors of rank 1 (vectors).

Obsolete Features

`$TraceLevel` is supported only for backwards compatibility. There is a nicer debugging facility in the `Parallel`Debug`` package.

`CloseSlave[]` has been renamed `Close[]`; the old name is still supported.

Incompatibilities

The symbols `$AvailableMachines`, `$RemoteCommand`, and `RemoteMachine` are no longer in the global context, but in `Parallel`Configuration``. If PCT finds that global symbols with the same names exist and have values, those values are picked up, so old initialization code should still work. (A shadowing warning is emitted in this case.)

`ResetSlaves[]` no longer clears remote definitions. It is intended to clean up after an abort of the master kernel. There is a new function `ClearSlaves[]` that clears all remote definitions and shared variables.

`Queue[]` and `DoneQ[]` no longer call `QueueRun[]`. Explicit scheduling code should be checked for any necessary changes. In most cases, `QueueRun[]` is called implicitly when `Wait[]` or `WaitOne[]` is called, so no changes will be necessary.

Use of `$TraceLevel` requires loading the debugging support package before loading PCT itself.

The default connection type for starting kernels on remote machines is `LinkConnect` rather than `LinkLaunch`. `LinkLaunch` is still used for local kernels.

The default link protocol for remote kernels is TCPIP. Launching remote kernels prior to *Mathematica* 5 requires a `LinkProtocol->"TCP"` setting.

Limitations

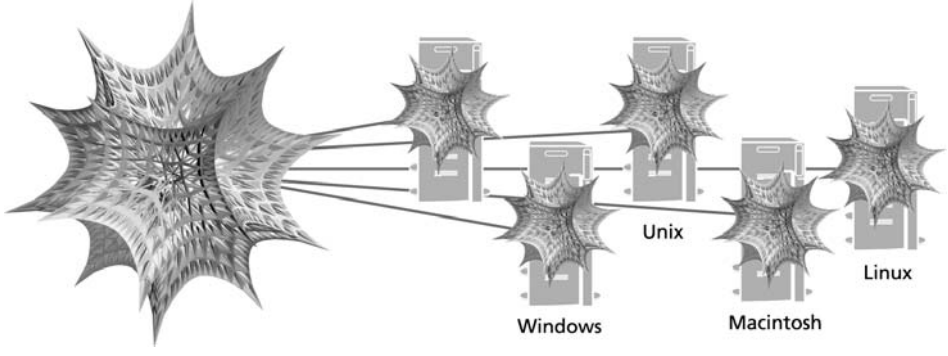
There is no `ParallelDo[]`, because `Do[]` makes sense only in the presence of side effects, which prevent a naive parallelization.

1 Introduction

Parallel Computation with *Mathematica*

The *MathLink* communication protocol can be used to control several *Mathematica* kernel processes from within *Mathematica*. This feature allows the implementation of a distributed-memory environment for parallel programming. Parallel language constructs, such as a parallel version of `Map`, can easily be implemented on top of these primitive operations.

Parallel Computing Toolkit (PCT) is written entirely in *Mathematica* and is therefore machine-independent. It has been tested on Unix, Linux, Windows, and Macintosh platforms. This product can be used in heterogeneous networks. All client and application code is distributed through *MathLink.* No common file system is necessary.



To perform computations in parallel, you need to be able to perform the following tasks:

- start processes and wait for processes to finish
- schedule processes on available processors
- exchange data between processes and synchronize access to common resources

In the *Mathematica* environment, the term *processor* refers to a running *Mathematica* kernel, whereas a *job* or *process* is an expression to be evaluated.

Parallel Computing Toolkit Features

The main features of PCT are

- distributed memory, master/slave parallelism
- written in *Mathematica*
- machine independent
- *MathLink* communication with remote kernels
- exchange of symbolic expressions and programs with remote kernels, not only numbers and arrays
- heterogeneous network, multiprocessor machines, LAN, and WAN
- virtual process scheduling or explicit process distribution to available processors
- virtual shared memory, synchronization, locking
- latency hiding
- parallel functional programming and automatic parallelization support
- failure recovery, automatic reassignment of stranded processes on failed remote computers

Requirements

To use PCT, you need access to a number of remote computers capable of running *Mathematica* or use of a multiprocessor local machine, a suitable network connection between your local computer and the remote machines, and the required number of *Mathematica* licenses. Note that even if a network is set up, there may be security restrictions that limit your ability to start *Mathematica* on remote computers.

To start *Mathematica* on a remote computer, the remote computer must run an `rsh/ssh` daemon or other remote login or execution service. The chapter Starting Remote Kernels contains detailed discussions of the various available options.

An alternative approach that works on any computer equipped with a TCP/IP network, even without an `rsh` daemon, is to manually start the desired kernels on each remote machine and then connect to the waiting kernels from the local machine.

Overview of Remote Execution

The method used to start remote kernels depends on both the operating system of your local computer and the types of remote computers you use. You can start kernels on remote computers that have an operating system different from the one you are using locally.

This section covers typical PCT commands you would use to start remote kernels on Windows, Mac OS X, or Unix systems. The chapter Starting Remote Kernels will describe how to start kernels manually and provide details on the commands presented in this section.

PCT provides a high-level command `LaunchSlave` for connecting to and starting kernels on remote computers. The command has the following general form.

```
LaunchSlave["remotehost", "oscommands", "options"]
```

The variable *remotehost* is the name of the remote computer on which you will start a kernel. On a local network, this can be a simple hostname. On a wide-area network, this would typically be a domain name, such as host.example.com. If you have a multiprocessor and can therefore start kernels on your local machine, use `"localhost"` rather than the computer name.

The *oscommands* argument is passed to the command interpreter on your computer; its form depends on your operating system. This argument can be a series of commands that start a kernel. Certain values are interpolated into the command string to make this feature more general. Typical *oscommands* are:

- `ssh`: The name of your local `ssh` client command. This command is used to establish a secure connection to a remote computer. ssh is provided with most versions of Unix, and it is available as third-party software for Windows.

- `rsh`: The name of your local `rsh` client command. It works in a similar way to `ssh` using a widely supported standard protocol, but provides only minimal security features.
- `winrsh`: A Windows `rsh` client distributed with *Mathematica* for systems lacking `rsh`.
- `math`: The name of the command on the remote computer to start the *Mathematica* kernel. You may have to give a full pathname such as `/usr/local/bin/math`.
- `$mathkernel`: The full pathname of the command used to start a *local* kernel.

A remote host may require your login name before you can establish a connection. In this case *username*, your login name on the remote computer, will be part of the second argument of `LaunchSlave`.

Before running any commands, load PCT into your local *Mathematica* session with the `Needs` command.

```
In[1]:= Needs["Parallel`Debug`"]
        Needs["Parallel`"]

           Parallel Computing Toolkit 2.0 (November 11, 2004)

           Created by Roman E. Maeder
```

Working on a Unix or Macintosh Computer

To connect to a remote computer running Unix or Mac OS X and start a *Mathematica* kernel there, use

```
LaunchSlave["remotehost"]
```

This command uses the value of the variable `$RemoteCommand` as the default *oscommands* argument. The slots `` `1` `` through `` `4` `` are replaced by values such as remote hostname, `linkname`, login name, and `linkprotocol`.

```
In[2]:= $RemoteCommand

Out[2]= ssh -x -f -l `3` `1` math -mathlink -linkmode Connect `4` -linkname '`2`' -noinit
```

If this is not appropriate for a particular remote host, you can supply your own custom command.

```
LaunchSlave["remotehost",
 "rsh `1` /usr/local/bin/math -mathlink -linkmode Connect `4` -linkname '`2`' &"]
```

To connect to your own local machine and start a kernel there (recommended for testing and if you have a multiprocessor machine), use the following command.

```
In[3]:= LaunchSlave["localhost"]

Out[3]= slave1[localhost]
```

Working on Windows

Please note that establishing connections *to* Windows requires third-party software (some of which is available for free) and special installation. Please refer to the detailed discussion in Chapter 2, Starting Remote Kernels. You, however, can easily establish connections *from* your local Windows PC.

To connect to a remote computer with an `rsh` daemon, use

```
LaunchSlave["remotehost"]
```

This command uses the value of the variable `$RemoteCommand` as the default *oscommands* argument. The slots `` `1` `` through `` `4` `` are replaced by values such as remote hostname, `linkname`, login name, and `linkprotocol`.

```
In[2]:= $RemoteCommand

Out[2]= rsh `1` -n -l `3` "math -mathlink -
          linkmode Connect `4` -linkname `2` -noinit >& /dev/null &"
```

If this is not appropriate for a particular remote host, you can supply your own custom command.

```
LaunchSlave["remotehost", $winrsh <>
  " -m -h `1` -l `3` -t 2 'math -mathlink `4` -linkmode Connect -linkname `2`'"]
```

To connect to your own local machine and start a kernel there, use the following command.

```
In[3]:= LaunchSlave["localhost"]

Out[3]= slave2[localhost]
```

Simple Parallel Computations

Once you have successfully started at least one remote kernel, you can begin to use PCT.

First, you can ask each remote kernel to identify itself. The result is a list of each remote kernel's unique ID, the remote host's name, *Mathematica*'s identifier for the remote operating system, the remote kernel's process ID, and the *Mathematica* version running on the remote computer.

```
In[10]:= TableForm[
          RemoteEvaluate[{$ProcessorID, $MachineName, $SystemID, $ProcessID, $Version}],
          TableHeadings -> {None, {"ID", "host", "OS", "process", "Mathematica Version"}}]
```

```
Out[10]//TableForm=
ID      host        OS            process     Mathematica Version
4       prokyon     UltraSPARC    11293       5.0 for Sun Solaris (UltraSPARC) (November 26, 2003)
5       sirius      UltraSPARC    20025       5.0 for Sun Solaris (UltraSPARC) (November 26, 2003)
6       delia       Linux         1304        5.0 for Linux (November 18, 2003)
```

You can try to run the same *Mathematica* command on all remote computers. Normally, all the results returned should agree. Here a definite integration is performed on each of the three remote kernels.

```
In[11]:= RemoteEvaluate[∫_1^∞ x^-2 dx]
```

```
Out[11]= {1, 1, 1}
```

Here four definite integrals with different lower bounds are computed in parallel.

```
In[12]:= ParallelTable[Integrate[Sin[x]/Sqrt[x], {x, i π, ∞}], {i, 0, 4}]
```

$$\text{Out[12]= } \left\{\sqrt{\frac{\pi}{2}},\ \sqrt{\frac{\pi}{2}}-\sqrt{2\pi}\ \text{FresnelS}\left[\sqrt{2}\right],\ \sqrt{\frac{\pi}{2}}-\sqrt{2\pi}\ \text{FresnelS}[2],\right.$$
$$\left.\sqrt{\frac{\pi}{2}}-\sqrt{2\pi}\ \text{FresnelS}\left[\sqrt{6}\right],\ \sqrt{\frac{\pi}{2}}-\sqrt{2\pi}\ \text{FresnelS}\left[2\sqrt{2}\right]\right\}$$

The remaining chapters of the documentation will provide many more examples of typical parallel computations you can perform with the help of PCT.

Cleaning Up

When you have completed your parallel computations, you should stop all remote kernels before exiting your local *Mathematica* kernel and front end.

```
In[13]:= CloseSlaves[]

Out[13]= {zombie[localhost], zombie[sirius], zombie[delia]}
```

2 Starting Remote Kernels

Configurations of local computing infrastructure vary widely from site to site. Unfortunately, there is no simple way to accommodate all possible setups. This chapter describes the various configurations in detail, so the content is fairly technical. Your local system administrator may be able to help you set up remote connections.

The way to start a remote (slave) kernel depends on the operating systems of the local and remote machines, the properties of the network, and the security measures in effect. Note that you can also start slave kernels on the local machine where the master kernel is running. This is particularly useful for testing and on multiprocessor machines.

MathLink Communication Modes

PCT uses *MathLink* to communicate with remote kernels. Once a connection has been established, it is used for any further communication with the remote kernel. The *MathLink* connection provides a machine-independent channel for *Mathematica* expressions between the controlling (master) kernel and the remote (slave) kernels.

In general, establishing a connection to a remote kernel requires two steps. First, the remote kernel must be started, then it must be instructed to establish a *MathLink* connection to the master kernel. Both of these tasks can be performed with the command `LaunchSlave[]`. Depending on its arguments, it uses various *MathLink* commands to achieve the result.

Active Connection (LinkLaunch)

An active connection is initiated from the master kernel by using the *MathLink* function `LinkLaunch["`*oscommands*`", `*options*`]`. The argument *oscommands* is an operating system command that makes a connection to a remote machine and starts a *Mathematica* kernel on that remote machine.

`LinkLaunch[]` is used by default for launching slave kernels on the local machine in the form `LaunchSlave["localhost",`*options*`]`. The PCT command `LaunchSlave[`"*remotehost*"`,`"*oscommands*"`, ConnectionType→LinkLaunch]` also uses the active connection method.

Callback Connection (LinkCreate)

For kernels on remote machines, it is generally better to establish separate *MathLink* connections than to use the command channel opened by `LinkLaunch`. The master kernel opens a *MathLink* link in Listen mode using `LinkCreate[]`, then the remote kernel is instructed to connect to the listening link.

The following command creates a link to which remote kernels can connect.

```
LaunchSlave["remotehost", "oscommands", options]
```

The *oscommands* is usually a template that may contain the sequences `` `1` ``, `` `2` ``, `` `3` ``, and `` `4` ``, which are replaced by values computed by the code in `LaunchSlave`. `` `1` `` is replaced by the hostname, `` `2` `` by the name of the link created, `` `3` `` by the user name, and `` `4` `` by the *MathLink* `linkprotocol` specification for nondefault protocols. Examples for the use of these placeholders is given in the examples that follow.

Passive Connection

Active or callback connections may not be available because of operating system deficiencies or security measures. If this is the case, another method for establishing a connection is available. You can manually start a kernel on a remote machine and instruct it to open a TCPIP port on which to listen for connection requests. This is usually achieved by providing the command-line arguments `-mathlink -linkcreate` to the command to start the kernel, usually `math`. (Under Windows, add `-linkprotocol TCPIP`.) The started kernel will tell you the ports on which it is listening.

In the master kernel you can make a connection to a listening kernel with the following *MathLink* command

```
LinkConnect["port1@hostname,port2@hostname"]
```

The PCT command `ConnectSlave[`"*port1@hostname,port2@hostname*"`]` will connect to a listening remote kernel.

Link Objects

The result of a successful `LinkLaunch` or `LinkConnect` connection is a *MathLink* link object having the following form.

```
LinkObject[name,number,...]
```

The *name* is taken from the argument of `LinkLaunch` or `LinkConnect` and allows you to identify the object. PCT keeps track of the available remote kernels by maintaining a list of such link objects in the variable `$Slaves`. To obtain the raw link object from a remote kernel object, use `LinkObject[`*kernel*`]`.

Remote Execution Options

To use active or callback connections, you need a way to execute a command to start *Mathematica* on the remote computer.

The available methods for remote command execution depend on the operating system of the master and slave machines. The network applications `rsh` and `ssh` are standard under Unix. Under Windows you can use any `rsh` program that may be provided with the system or available from a number of sources, or use `winrsh`, which is supplied with *Mathematica*.

Your local machine, from which you want to initiate a connection, needs an `ssh` or `rsh` *client program*; the remote machine needs a corresponding *daemon*.

Remote Execution under Unix and Mac OS X

To make a connection from your local Unix or Mac OS X machine you can use the `rsh` or `ssh` programs.

The shell `ssh` is a replacement for `rsh` that offers secure cryptographic authentication and encryption of the communication between the local and remote machines. It is, therefore, usable in situations where the `rsh` security is insufficient such as on the internet. If your site is requiring `ssh`, please contact your system administrator about your local setup. PCT has been tested with Version 2 of `ssh`.

Using ssh

To test whether ssh is configured correctly, the following command can be given in a shell window.

```
ssh remotehost math
```

Here, *remotehost* is the name of the remote machine and math is the command to start a *Mathematica* kernel on the remote machine. If the remote machine is outside of the local area network, then *remotehost* must be a fully qualified domain name. If the math command is not on the search path, the full pathname can be given instead, for example /usr/local/bin/math.

It is a good idea to try to establish a connection in a shell window to see whether everything is set up correctly before trying to use the given remote host in PCT. If everything is fine, the remote kernel should print the familiar In[1]:= prompt. You can then use Quit[] to terminate the remote kernel and the connection to the remote machine.

Once ssh is working, you can use the following command to start a kernel on a remote Unix machine.

```
LaunchSlave["remotehost","ssh -f `1` math -mathlink -linkmode Connect
-linkname '`2`'"]
```

The placeholder `1` is replaced by the remote hostname *remotehost*, `2` is replaced by the link specification of the link created by LinkCreate[]. The resulting command is then executed by the operating system. LaunchSlave[] supports a number of additional placeholders to accommodate more complicated situations, see the section Configuring *Parallel Computing Toolkit*.

If you leave out the second argument, the following default is used.

```
$RemoteCommand

ssh -x -f -l `3` `1` math -mathlink -linkmode Connect `4` -linkname '`2`' -noinit
```

For an active connection that uses standard input and output as *MathLink* transport, use the following command.

```
LaunchSlave["remotehost","ssh `1` math -mathlink",
ConnectionType->LinkLaunch]
```

The placeholder `1` is replaced by the remote host name *remotehost*. The resulting command is then executed by the operating system.

You may have to prefix the remote kernel command `math` with the appropriate pathname on the remote machine, such as `/usr/local/bin/math`.

rsh

The following shell command starts an interactive *Mathematica* kernel on a remote machine.

```
rsh remotehost math
```

Here, *remotehost* is the name of the remote machine and `math` is the command to start a *Mathematica* kernel. If the remote machine is outside of the local area network, then *remotehost* must be a fully qualified domain name. If the `math` command is not on the search path, the full pathname can be given instead, for example `/usr/local/bin/math`.

It is a good idea to try to establish a connection in a shell window to see whether everything is set up correctly before trying to use the given remote host in PCT. If everything is fine, the remote kernel should print the familiar `In[1]:=` prompt. You can then use `Quit[]` to terminate the remote kernel and the connection to the remote machine.

Once `rsh` is working, you can use the following PCT command to start a kernel on a remote Unix machine, using a callback connection.

```
LaunchSlave["remotehost","rsh `1` math -mathlink -linkmode Connect
-linkname '`2`' &"]
```

The placeholder `` `1` `` is replaced by the remote hostname *remotehost*, `` `2` `` is replaced by the link specification of the link created by `LinkCreate[]`. The resulting command is then executed by the operating system.

For an active connection that uses standard input and output as *MathLink* transport, use the following command.

```
LaunchSlave["remotehost","rsh `1` math -mathlink",
ConnectionType->LinkLaunch]
```

The placeholder `` `1` `` is replaced by the remote host name *remotehost*. The resulting command is then executed by the operating system.

You may have to prefix the remote kernel command `math` with the appropriate pathname, such as `/usr/local/bin/math`.

To start a kernel on a remote Windows machine, the remote machine must have an rsh daemon running. You can start a remote kernel on a Windows machine from a Unix host by using commands similar to those explained in the section Remote Execution under Windows. For example, issue the following command on a local Unix machine to start a remote kernel on a remote Windows host.

```
LaunchSlave["remoteWindowshost",
"rsh `1` math -mathlink -linkmode Connect -linkname `2` &"]
```

Security considerations

You can only use rsh if you are allowed to log into the remote machine without a password. For this to work, your local machine must be in the remote machine's /etc/hosts.equiv or ~/.rhosts file. See the Unix Manual for rlogin and rsh for more details and consult your system administrator.

Starting kernels on your local machine

For testing, and if you have a multiprocessor machine available, you can also start kernels on your local machine where you operate the master kernel and the front end.

```
LaunchSlave["localhost"]
```

This command uses the value of the variable $mathkernel as the command to launch a kernel. It should be set up suitably for your *Mathematica* installation.

A typical value for Unix is shown here.

```
$mathkernel

/usr/local/Wolfram/Mathematica/5.0/Executables/math -noinit -mathlink
```

Here is a typical value for Mac OS X.

```
$mathkernel

/Mathematica\ 5.0.app/Contents/MacOS/MathKernel -noinit -mathlink
```

Remote machines running Mac OS X

No math script is installed on Mac OS X. To launch a remote kernel on a Mac OS X machine, you can either give the full pathname of the MathKernel executable or write your own math script; see the support pages support.wolfram.com/applicationpacks/parallel for more information.

The kernel is typically at `$InstallationDirectory/Contents/MacOS/MathKernel`. Because the pathname contains space characters, it needs to be enclosed in double quotes and the space escaped by a backslash. Here is an example of a command to launch a kernel on a Mac OS X machine.

```
LaunchSlave["remotehost","ssh -f `1` \"/Mathematica\\ 5.1.app/Contents/Mac
OS/MathKernel\" -mathlink -linkmode Connect `4` -linkname '`2`'"]
```

Remote Execution under Windows

Under Windows you can use any available `rsh` program or `winrsh.exe`, which is supplied with *Mathematica*, to start remote kernels. The remote kernel can then establish a TCPIP connection back to the local kernel. This is most easily done by creating a *MathLink* object locally to which the remote kernel can establish a callback connection.

On the remote end, the *Mathematica* kernel command-line arguments `-linkmode Connect -linkprotocol TCPIP -linkname` *port1@host,port2@host* instruct the kernel to connect to an open port on your local machine, *host*. PCT will provide the *port1@host,port2@host* argument for you. Use `` `2` `` to interpolate it into the command string.

You need a `rsh` daemon for all remote machines. Note that *Mathematica* does not provide an `rsh` daemon.

If your version of Windows includes `rsh`, you can use these arguments to `LaunchSlave` to make connections to remote hosts.

```
LaunchSlave["remotehost"]
```

This command uses the following template for starting the remote kernel.

```
$RemoteCommand

rsh `1` -n -l `3` "math -mathlink -
  linkmode Connect `4` -linkname `2` -noinit >& /dev/null &"
```

The first placeholder `` `1` `` will be replaced by the hostname *remotehost* as usual; the second placeholder `` `2` `` will be replaced by a link object created on the local machine to which the remote kernel can connect. The placeholder `` `3` `` is replaced by the username. If the remote host does not require a login username, omit the `-l `3`` option. The placeholder `` `4` `` is replaced by the correct `-linkprotocol` setting.

You may have to prefix the remote kernel command `math` with the appropriate pathname, such as `/usr/local/bin/math` for a remote Unix system. Be sure to give the correct user name for connecting to the remote machine.

The winrsh client

Mathematica for Windows ships with an `rsh` client `winrsh.exe` that can be used if no `rsh` command is available. To use this client to make connections to remote hosts, use these arguments to `LaunchSlave`.

```
LaunchSlave["remotehost",
$winrsh<>" -m -h `1` -l `3` -t 2 'math -mathlink `4` -linkmode Connect
-linkname `2`'"]
```

You can make this command the default by setting `$RemoteCommand` like this.

```
$RemoteCommand = $winrsh <>
  " -m -h `1` -l `3` -t 2 'math -mathlink `4` -linkmode Connect -linkname `2`'"
```

The `$winrsh` command opens a window. The connection will not be established and will appear to hang until the window has closed. The option `-t 2` instructs `winrsh` to close the window after two seconds.

The command `winrsh.exe` is found in the `SystemFiles\FrontEnd\Binaries\Windows` folder inside your *Mathematica* installation. PCT provides a variable `$winrsh` that contains the complete pathname appropriate for your installation.

```
$winrsh

"C:\Program Files\Wolfram Research\
  Mathematica\5.0\SystemFiles\FrontEnd\Binaries\Windows\winrsh"
```

In case of problems, you can open an MS-DOS window and try `winrsh` there. You can do something like the following.

```
"C:\Program Files\Wolfram
Research\Mathematica\5.0\SystemFiles\FrontEnd\Binaries\Windows\winrsh.exe
"-h remotehost -l username 'dir'
```

Then `winrsh` should open a window and display a directory listing, `dir`, of the remote host.

Available third-party software

A list of third-party `rsh` daemons and clients can be found at support.wolfram.com/applicationpacks/parallel. (The author, MathConsult Dr. R. Mäder, and Wolfram Research, Inc. do not endorse any of the products listed at that URL. We provide this information in the hope that it may be useful.)

If you have an ssh client installed on Windows, you may be able to connect to a remote Unix or Mac OS X slave using ssh. Neither PCT nor *Mathematica* supply an ssh client for Windows, but there are several available commercially. You can then use the following command to launch remote kernels.

```
LaunchSlave["remotehost","ssh -x -f -l `3` `1` \"math -mathlink -linkmode
Connect `4` -linkname '`2`' </dev/null >&/dev/null &\""]
```

The exact arguments needed may vary with network and machine configurations.

Starting a kernel on a local Windows machine

For testing and in multiprocessor machines, you can conveniently start a kernel on the local machine. Use this command.

```
LaunchSlave["localhost"]
```

This command uses the value of the variable `$mathkernel` as the command to launch a kernel. It should be set up suitably for your *Mathematica* installation.

```
$mathkernel

"C:\Program Files\Wolfram Research\Mathematica\5.0\MathKernel"
```

Remote machines running Mac OS X

No `math` script is installed on Mac OS X. To launch a remote kernel on a Mac OS X machine, you can either give the full pathname of the `MathKernel` executable or write your own `math` script; see the support pages support.wolfram.com/applicationpacks/parallel for more information.

The kernel is typically at `$InstallationDirectory/Contents/MacOS/MathKernel`. Because the pathname contains space characters, it needs to be enclosed in double quotes and the space escaped by a backslash. Here is an example of a command to launch a kernel on a Mac OS X machine.

```
LaunchSlave["remotehost","ssh `1` \"\"/Mathematica\\ 5.1.app/Contents/Mac
OS/MathKernel\" -mathlink -linkmode Connect `4` -linkname ''2`'
</dev/null >&/dev/null &\""]
```

Passive Connections

If your local computer does not provide an `rsh` client or the remote computer does not provide an `rsh` daemon, you have to start the required remote kernels manually on each remote computer.

These command-line options should be given to the kernel command on the remote machines.

```
-mathlink -linkcreate
```

Under Windows, you should add `-linkprotocol TCPIP`.

The kernel will start up and tell you the address or *linkname* where it is listening. Addresses have the form *port@host* or *port1@host,port2@host*, where *port* is a TCP port number (a decimal integer) and *host* is the computer's name.

With this information, you can establish a connection from your local kernel to the remote one with the following command.

```
ConnectSlave["port1@host,port2@host"]
```

Under Windows, you should include the option setting `LinkProtocol->"TCPIP"`.

Alternatively, `ConnectSlave` can take an already established *MathLink* link object as its argument.

Note that you may be able to use a Telnet application to log in to a remote computer, so you can give the commands described in the following section from your local machine. Otherwise, you will have to enter the command at the computer's console.

Unix and Mac OS X Remote Computers

To start a kernel on Unix, give the following command in a shell or Telnet window.

```
math -mathlink -linkcreate
```

Mathematica will output the listening ports on standard output. If `math` is not on your search path, give an absolute pathname, such as `/usr/local/bin/math`.

Once you receive the listening port information, you can connect to the waiting remote kernel from your local computer with the command `ConnectSlave["`*linkname*`",LinkProtocol->"TCPIP"]` (you can omit the option setting on a Unix local computer).

Windows Remote Computer

To start a kernel on Windows, give the following command in an MS-DOS window on the remote computer.

```
math -mathlink -linkcreate -linkprotocol TCPIP
```

Mathematica will open a small panel that displays the ports on which it is listening. You must close this panel before the connection can be used.

Once you have the listening port information, connect to the waiting remote kernel from your local computer with the command `ConnectSlave["`*linkname*`",LinkProtocol->"TCPIP"]`.

Using a Running Kernel

You can also prepare a running kernel as a remote kernel for parallel computations. Start the kernel by double-clicking the MathKernel (not *Mathematica*) icon or starting `math` in a shell window. You can then create the required link from within the *Mathematica* kernel as follows.

1. Launch (double-click) `MathKernel`. Do not launch `Mathematica`! A window with the prompt `In[1]:=` appears.
2. At the `In[1]:=` prompt, give the following command shown here with the expected form of the result.

```
In[1]:= lnk = LinkCreate[LinkProtocol -> "TCPIP", LinkMode -> Listen]

LinkObject[port1@host,port2@host, 2, 2]
```

3. Take note of the port numbers that appears in place of *port*.
4. At the next input prompt, `In[2]:=`, give the following command. No output will be produced.

```
In[2]:= $ParentLink = lnk
```

With this information, you can now connect to the waiting remote kernel from your local computer with the command `ConnectSlave["port1@host,port2@host"]`.

Configuring *Parallel Computing Toolkit*

This section lists the commands available in PCT and shows you how you can prepare a configuration file to automate the task of starting the remote kernels that are usually available to you.

Launching Remote Kernels

To start a remote kernel and add it to the list of available slave processors, use the command `LaunchSlave`.

`LaunchSlave["remotehost", "oscommands"]`
: use the operating system (shell) command *oscommands* to start a kernel on a remote machine named *remotehost* and have it connect to a link created on the local machine

`LaunchSlave["remotehost", "oscommands", ConnectionType->LinkLaunch]`
: use the operating system (shell) command *oscommands* to start a kernel on a remote machine named *remotehost* using `LinkLaunch` (no separate *MathLink* connections)

`LaunchSlave["localhost", "oscommands"]`
: use the operating system (shell) command *oscommands* to start a kernel on the local machine (using `LinkLaunch`)

`LaunchSlave["localhost"]`	use the operating system (shell) command stored in `$mathkernel` to start a kernel on the local machine
`LaunchSlave["`*remotehost*`"]`	use the operating system (shell) command stored in `$RemoteCommand` to start a kernel
`$ProcessorID`	a unique integer assigned to each remote kernel (numbers are assigned starting with 1)

Starting remote kernels.

The argument *oscommands* can be a template containing the character sequences `` `1` `` for connection type `LinkLaunch` and `` `1` `` through `` `4` `` for connection type `LinkConnect`.

placeholder	*meaning*	
`` `1` ``	hostname	the remote hostname, the first argument of `LinkLaunch`
`` `2` ``	link name	the name of the link object created
`` `3` ``	remote user	the username on the remote machine, the value of `$RemoteUserName`, which defaults to `$UserName`
`` `4` ``	protocol	a suitable `-linkprotocol` *proto* setting for the *MathLink* argument list

Placeholders in operating system command templates.

If there is no `LinkProtocol->"`*proto*`"` setting in `LaunchSlave`, the placeholder `` `4` `` expands to the empty string for local connections (to use the native default protocol) and to `-linkprotocol TCPIP` for remote connections. If an explicit `LinkProtocol->"`*proto*`"` setting exists, `` `4` `` expands to `-linkprotocol` *proto*.

To connect to a *Mathematica* 4.2 remote kernel, use `LinkProtocol->"TCP"` in the `LaunchSlave` command.

The following options can be given in LaunchSlave.

option name	default value	
InitCode	$InitCode	a sequence of commands (wrapped inside Hold) to send to each remote kernel upon startup
ConnectionType	Automatic	the *MathLink* connection type, which can be either LinkLaunch or LinkCreate; by default, LinkLaunch is used for local kernels, and LinkCreate is used for remote kernels
LinkProtocol	Automatic	this option is passed on to LinkLaunch or LinkCreate by default, "TCPIP" is used for remote kernels, and no setting is used for local kernels
LinkHost	""	this option is passed on to LinkCreate; it can be used to specify the interface on which the link is listening
ProcessorSpeed	1	an estimate of the relative speed of the remote kernel

Options of LaunchSlave.

The default value of the variable $InitCode is Hold[$DisplayFunction=Identity;].

These options can also be given to ConnectSlave[].

If all or most of your remote hosts can be reached with the same command, you can set $RemoteCommand to a suitable command template that is used by default in LaunchSlave.

For Unix and Mac OS X, the default value is

```
$RemoteCommand="ssh -x -f -l `3` `1` math -mathlink -linkmode Connect
`4` -linkname '`2`' -noinit";
```

One requirement is that the command return quickly, even though *Mathematica* keeps running. If it does not return, you can put the command into the background with a setting like the following.

```
$RemoteCommand="ssh -n -x -f -l `3` `1` math -mathlink -linkmode Connect
`4` -linkname '`2`' -noinit &";
```

Under Windows, `$RemoteCommand` is set by default to

```
$RemoteCommand="rsh `1` -n -l `3` \"math -mathlink -linkmode Connect `4`
-linkname `2` -noinit >& /dev/null &\"";
```

To use `winrsh.exe`, if no `rsh` is available, use

```
$RemoteCommand=$winrsh<>" -m -h `1` -l `3` -t 2 'math -mathlink `4`
-linkmode Connect -linkname `2`'";
```

If `$RemoteCommand` is set up correctly, you can simply use the following commands to start a kernel on remote hosts named, for example, `host1` and `host2`.

```
LaunchSlave["host1"]
LaunchSlave["host2"]
```

To connect to your local machine (recommended for testing and if you have a multiprocessor machine), you should be able to use

```
LaunchSlave["localhost"]
```

You may want to verify that `$mathkernel` contains the appropriate command for invoking a local kernel by evaluating `$mathkernel`.

```
$mathkernel
```

Using Passive Connections

For passive connections, you should manually start the remote kernels with the `-linkcreate` argument as described earlier, note the ports on which the remote kernels are listening, and use `ConnectSlave` for each remote kernel.

`ConnectSlave["`*linkname*`"]`	connect to a listening link on the given computer
`ConnectSlave[`*link*`]`	connect to an existing *MathLink* object

Connecting to listening kernels.

Port numbers will usually be different each time you start a remote kernel; therefore, this method cannot easily be automated.

Preparing a Host Description List

To automate the task of starting remote kernels, you can prepare a list of available machines.

`RemoteMachine["`*remotehost*`", "`*oscommand*`s"]`	a host description for a computer named *remotehost* using the command *oscommands* for connection and using the default connection type defined for `LaunchSlave`
`RemoteMachine["`*remotehost*`"]`	host description for a computer named *remotehost* using the default command `$RemoteCommand` for connection
`"`*remotehost*`"`	simple hostname; shortcut for `RemoteMachine["`*remotehost*`"]`
`$RemoteCommand`	the default *oscommands* to use
`RemoteMachine["`*localhost*`"]`	host description for the local machine
`$mathkernel`	the default command to start a local kernel

Host description entries.

The command `LaunchSlaves[`*list*`]` takes a list of such host descriptions as an argument and tries to establish a connection to each of the hosts listed.

Finally, you can assign a list of host descriptions to the global variable `$AvailableMachines` and use `LaunchSlaves[]` without an argument, which will consult this variable.

Defining a Default Configuration

Note that you can put assignments for `$AvailableMachines` and `$RemoteCommand` into your personal *Mathematica* kernel startup file `init.m`.

You do not need to load PCT to define a default configuration. There is a smaller package `Parallel`Configuration`` that you can load instead.

```
Needs["Parallel`Configuration`"]
```

Alternatively, you can put assignments for `$AvailableMachines` and `$RemoteCommand` into a notebook and evaluate them to set up the connections. You can use one of the samples here as a template. Copy the appropriate cell group into a new notebook and save it under a name such as UnixInit.nb or WindowsInit.nb. Then you can simply open this notebook and evaluate its cells to set up your remote kernels.

Sample configuration for Windows

Some of the input cells in this template have been made inactive (not evaluatable), because they contain commands for optional features. Enable these cells on a case-by-case basis according to your needs. These commands will evaluate properly only if you have access to a Windows machine on which to start a kernel and you substitute valid values for variable arguments.

Load PCT.

```
Needs["Parallel`Debug`"]
Needs["Parallel`"]
```

Enable optional features as desired.

```
Needs["Parallel`Commands`"]
```

Set your default remote command if the default is not suitable.

```
$RemoteCommand = "rsh `1` -n -l `3` \"math -mathlink -
    linkmode Connect `4` -linkname `2` -noinit >& /dev/null &\"";
```

Set the default remote username if it is different from your local username.

```
$RemoteUserName = "name";
```

If you have ssh available, you can set $RemoteCommand to use ssh.

```
$RemoteCommand =
  "ssh `1` -x -f -l `3`  \"math -mathlink -
    linkmode Connect `4` -linkname `2` </dev/null >& /dev/null &\"";
```

Use winrsh.exe if neither rsh nor ssh are available.

```
$RemoteCommand = $winrsh <>
   " -m -h `1` -l `3` -t 2 'math -mathlink `4`-linkmode Connect -linkname `2`'";
```

Set the default initialization for your remote kernels.

```
$InitCode = Hold[$DisplayFunction = Identity;];
```

List any normally available machines, filling in the *hostname* variable in each entry.

```
$AvailableMachines = {
        RemoteMachine["hostname1"],
        RemoteMachine["hostname2"]
    };
```

Now you can try to start a remote kernel on all defined remote machines.

```
LaunchSlaves[$AvailableMachines]
```

You can also put LaunchSlave or ConnectSlave commands for special cases here and evaluate them as needed.

```
LaunchSlave["specialhost", "special template", options]
```

Start a kernel on the local machine.

```
LaunchSlave["localhost"]
```

Connect to a manually started remote kernel.

```
ConnectSlave["port1@host,port2@host", LinkProtocol → "TCPIP"]
```

Now verify that all remote kernels are operating correctly by collecting information about them.

```
TableForm[
 RemoteEvaluate[{$ProcessorID, $MachineName, $SystemID, $ProcessID, $Version}],
 TableHeadings -> {None, {"ID", "host", "OS", "process", "Mathematica Version"}}]
```

After finishing your computations, you should close all connections.

```
CloseSlaves[]
```

Sample configuration for Unix and Mac OS X

Some of the input cells in this template have been made inactive (not evaluatable), because they contain commands for optional features. Enable these cells on a case-by-case basis according to your needs. These commands will evaluate properly only if you have access to a Unix machine on which to start a kernel and you substitute valid values for variable arguments.

Load PCT.

```
Needs["Parallel`Debug`"]
Needs["Parallel`"]
```

Enable optional features as desired.

```
Needs["Parallel`Commands`"]
```

Set your default `$RemoteCommand`.

```
$RemoteCommand = "ssh -x -f -l `3` `1` math -
    mathlink -linkmode Connect `4` -linkname '`2`' -noinit";
```

Set the default remote username if it is different from your local username.

```
$RemoteUserName = "name";
```

Set the default initialization for your remote kernels.

```
$InitCode = Hold[$DisplayFunction = Identity;];
```

List any normally available machines.

```
$AvailableMachines = {
        RemoteMachine["hostname1"],
        RemoteMachine["hostname2"]
    };
```

Now you can try to start a remote kernel on all defined remote machines.

```
LaunchSlaves[$AvailableMachines]
```

You can also put `LaunchSlave` or `ConnectSlave` commands for special cases here and evaluate them as needed.

```
LaunchSlave["specialhost", "special template", options]
```

Start a kernel on the local machine.

```
LaunchSlave["localhost"]
```

Connect to a manually started remote kernel.

```
ConnectSlave["port1@host,port2@host"]
```

Now verify that all remote kernels are operating correctly by collecting information about them.

```
TableForm[
 RemoteEvaluate[{$ProcessorID, $MachineName, $SystemID, $ProcessID, $Version}],
 TableHeadings -> {None, {"ID", "host", "OS", "process", "Mathematica Version"}}]
```

After finishing your computations, close all connections.

```
CloseSlaves[]
```

Kernel Initialization

To prevent the execution of the initialization commands you may have put into your init.m file, add the argument `-noinit` to any kernel invocation command. This is recommended unless you have put specific commands for initializing remote kernels into init.m.

You can put your remote kernel initialization commands into the PCT variable `$InitCode`.

Housekeeping

The list of available remote kernels is given in `$Slaves`. This is a read-only variable that contains the active kernel objects you have previously opened with `LaunchSlaves`, `LaunchSlave`, or `ConnectSlave`.

`Length[$Slaves]` gives you the number of currently connected remote machines or the degree of parallelism.

The Remote Kernel Object

The properties of the remote kernel objects can be obtained with these functions.

`ProcessorID[`*kernel*`]`	a unique integer assigned to each kernel
`ProcessorName[`*kernel*`]`	the name of the machine on which the kernel is running
`ProcessorSpeed[`*kernel*`]`	an estimate of the relative of the remote processor
`LinkObject[`*kernel*`]`	the raw *MathLink* `LinkObject` that connects to the remote kernel

Host description entries.

To get a nicely formatted listing of properties of the remote kernel connections, use this command. The command is followed by output from a sample session.

```
TableForm[
 Through[{Identity, ProcessorID, ProcessorName, ProcessorSpeed, LinkObject}[#]] & /@
  $Slaves,
 TableHeadings -> {None, {"slave", "id",  "name", "speed", "link"}}]
```

slave	id	name	speed	link
slave$_1$[localhost]	1	localhost	1	LinkObject[/home/prokyon/math/Dist/501/Executables/math -noinit -mathlink, 2, 2]
slave$_2$[sirius]	2	sirius	1	LinkObject[44320@denebola,44321@denebola, 3, 3]
slave$_3$[sirius]	3	sirius	1	LinkObject[44323@denebola,44324@denebola, 4, 4]
slave$_4$[prokyon]	4	prokyon	1	LinkObject[44326@denebola,44327@denebola, 5, 5]
slave$_5$[delia]	5	delia	1	LinkObject[44332@denebola,44333@denebola, 6, 6]

Remote Properties

The variable `$ProcessorID` is set on each remote kernel to its own processor ID (pid).

To get a nicely formatted listing of this and other standard properties of the remote kernels, use this command. The command is followed by output from a sample session.

```
TableForm[RemoteEvaluate[
  {$ProcessorID, $MachineName, $SystemID, $Version, $CommandLine, $ParentLink}],
 TableHeadings -> {None, {"ID", "host", "OS", "Mathematica Version",
    "CommandLine", "Parent Link"}}, TableDepth → 2]
```

```
ID      host          OS              Mathematica Version
1       denebola      UltraSPARC      5.0 for Sun Solaris (UltraSPARC) (November 26, 2003)
2       sirius        UltraSPARC      5.0 for Sun Solaris (UltraSPARC) (November 26, 2003)
3       sirius        UltraSPARC      5.0 for Sun Solaris (UltraSPARC) (November 26, 2003)
4       prokyon       UltraSPARC      5.0 for Sun Solaris (UltraSPARC) (November 26, 2003)
5       delia         Linux           5.1 for Linux (October 25, 2004)
```

Troubleshooting

If you get an error message and the result `$Failed` when using `LaunchSlave`, the connection could not be established. There are a number of reasons this can happen:

- The remote computer cannot be reached over the network, or you do not have sufficient privileges to execute remote commands on the computer.
- The remote computer does not run an `ssh` or `rsh` daemon. Such daemons are standard under Unix and Mac OS X and available as third-party products under Windows.
- *Mathematica* may not be installed correctly on the remote computer, the `math` command may not be on your search path, or you do not have a sufficient number of *Mathematica* licenses.
- Your remote execution command on Windows has exceeded the low, arbitrary limit on command length that Microsoft imposes on command execution. Please refer to the section Remote Execution under Windows for more details. In most cases, PCT will tell you that running the command has failed with exit code −1.

You can still continue to use any remote kernels that you could launch correctly; failed connections will never be used by PCT.

The variable `$Slaves` gives the current list of remote connections that started up normally. If there is at least one, you can continue to work with PCT. Evaluating the expression `$Slaves` will return the value of this variable.

To diagnose network problems, you can use the `netstat` operating system command in a Unix shell or MS-DOS window. You should try command-line arguments to find which will work on your operating system; most likely it will be one of the following.

```
netstat -a -p tcp
netstat -a -f inet
netstat -a -t
```

The output of `netstat` will list existing TCP connections to remote computers. Each remote kernel will occupy one or two such TCP connections.

Tracing *MathLink* Commands

With debugging and tracing enabled, `LinkLaunch[]` will show you which *MathLink* commands it runs to establish a connection with a remote kernel. To use these features, you have to load the debugging package before loading PCT.

```
In[1]:= Needs["Parallel`Debug`"]

In[2]:= Needs["Parallel`"]

        Parallel Computing Toolkit 2.0 (November 11, 2004)

        Created by Roman E. Maeder
```

Now you can enable `MathLink` tracing.

```
In[3]:= SetOptions[$DebugObject, Trace → {MathLink}]

Out[3]= {Trace → {MathLink}}
```

Sample trace of a callback connection

This sample output shows how a default connection to a remote host is established. The output shown is

- the `LinkCreate[]` command used to establish a listening link on the local machine
- the resulting link created
- the command run to start the remote kernel with all placeholders filled in
- the exit code of this command (should be 0)
- the remote kernel object of the new kernel

```
In[4]:= LaunchSlave["sirius"]

        MathLink: Creating listening link with
           LinkCreate[Sequence[LinkProtocol → TCPIP, LinkHost → , LinkHost → ]]

           MathLink: Link created as LinkObject[34946@prokyon,34947@prokyon, 2, 2]

        MathLink: Running command "ssh -x -f -l maeder sirius math -mathlink -linkmode
           Connect -linkprotocol TCPIP -linkname '34946@prokyon,34947@prokyon' -noinit"

        MathLink: Command returned 0

Out[4]= slave1[sirius]
```

Sample trace of a LinkLaunch connection

This sample output shows how a default connection to the local host is established. The output shown is

- the `LinkLaunch[]` command used to start the kernel, containing the operating system command to start the kernel itself and the `LinkLaunch[]` options used
- the resulting link created
- the remote kernel object of the new kernel

```
In[5]:= LaunchSlave["localhost"]

          MathLink: Launching kernel on localhost with LinkLaunch[
              Sequence[/home/prokyon/math/Dist/501/Executables/math -noinit -mathlink,
               LinkProtocol → Automatic, LinkHost → , LinkProtocol → Automatic, LinkHost → ]]

          MathLink: Link launched as LinkObject[
              /home/prokyon/math/Dist/501/Executables/math -noinit -mathlink, 3, 3]

Out[5]= slave2[localhost]
```

To turn off tracing when you are done, use the following.

```
In[6]:= SetOptions[$DebugObject, Trace → {}]

Out[6]= {Trace → {}}
```

Resetting and Terminating Remote Kernels

Resetting Kernels

After aborting the master kernel during a parallel computation and if remote kernels do not respond, you can try to reset them and bring them back into a usable state.

`Abort[`*kernel*`]`	aborts a kernel (interrupt any running evaluations)
`ResetSlaves[]`	discards any processes in the queue and aborts all running evaluations

Aborting and resetting kernels.

`ResetSlaves` can be used after a parallel computation has been aborted with the menu command Kernel ▹ Abort Evaluation, or CMD-..

For some remote kernel connections, notably for kernels started on remote machines using `LinkLaunch`, there may be no way to interrupt them. If a remote evaluation takes too long or is in an infinite loop, you must terminate the remote kernel process using the appropriate operating system command.

Clearing Kernels

Between different parallel computations, you may want to make sure that all remote kernels delete any variable definitions that may have been set. Rather than terminating and restarting all kernels, you can use `ClearSlaves`.

`ClearSlaves[]`	clears all variables in the remote kernel's `Global`` context and forgets any shared variables and exported environments

Clearing definitions.

Definitions for symbols in contexts other than `Global`` are not cleared.

Any definitions of global symbols exported with `ExportEnvironment` will become unavailable. Any shared global variables will become unshared.

Terminating Kernels

When you are done with your parallel computations, close any open remote kernel connections. This frees the resources occupied on the remote machines and closes the open network connections.

`Close[`*kernel*`]`	closes the given connection link and removes it from `$Slaves`
`CloseSlaves[]`	terminates all open connections

Terminating kernels.

Note that exiting the local master kernel may or may not close the open connections cleanly. Always use `CloseSlaves[]` before exiting the master kernel.

3 Parallel Evaluation

Sending Commands to Remote Kernels

Recall that connections to remote kernels, as opened by `LaunchSlave`, are represented as kernel objects. See the chapter Starting Remote Kernels for details. The commands in this section take such kernel objects as arguments and send or receive *Mathematica* expressions to remote kernels using these links. In the following, the variable *link* is a kernel returned by `LinkLaunch` or `LinkConnect`.

Low-Level Send and Receive

`Send[`*link*`, `*cmd*`]`	sends *cmd* for evaluation to the remote kernel connected to *link*; returns *link*
`Send[{`*links…*`}, `*cmd*`]`	sends *cmd* for evaluation to all remote kernels listed
`ReceiveIfReady[`*link*`]`	returns a result waiting on the given link; returns `$NotReady` if no result is waiting
`ReceiveIfReady[`*link, h*`]`	wraps the result in *h*`[…]` before returning it
`Receive[`*link*`]`	waits for a result on the given *link* and returns it
`Receive[{`*links…*`}]`	waits for a result on each of the *links* and returns them in a list
`Receive[`*links, h*`]`	wraps the results in *h*`[…]` before returning them
`RemoteEvaluate[`*cmd*`, `*link*`]`	sends *cmd* for evaluation to the remote kernel connected to *link*, then waits for the result and returns it
`RemoteEvaluate[`*cmd*`, {`*links…*`}]`	sends *cmd* for evaluation to the remote kernels given, then waits for the results and returns them
`RemoteEvaluate[`*cmd*`]`	sends *cmd* for evaluation to all remote kernels and returns the list of results; equivalent to `RemoteEvaluate[`*cmd*`, $Slaves]`

Sending and receiving commands to and from remote kernels.

`Send` has the attribute `HoldRest` so that the given command is not evaluated before it is sent to the remote kernel. `RemoteEvaluate` has the attribute `HoldFirst`.

`RemoteEvaluate[`*cmd*`, `*link*`]` is equivalent to the combination `Send[`*link*`, `*cmd*`]; Receive[`*link*`]`.

With `ReceiveIfReady[`*link*`]`, you can poll several open links for results.

You cannot use `RemoteEvaluate` while a concurrent computation involving `Queue` or `Wait` is in progress. See the chapter Concurrency: Managing Parallel Processes for details.

Values of Variables

Values of variables defined on the local master kernel are usually not available to remote kernels. If a command you send for evaluation refers to a variable, it usually will not work as expected. The following piece of code will return `False` because the symbol `a` will most likely not have any value at all on the remote kernel.

```
a = 2;
RemoteEvaluate[a === 2, link]
```

A convenient way to insert variable values into unevaluated commands is to use `With`, as demonstrated in the following command.

```
With[{a = 2}, RemoteEvaluate[a === 2, link] ]
```

The symbol `a` is replaced by `2`, then the expression `2 === 2` is sent to the remote kernel where it evaluates to `True`.

If you need variable values and definitions carried over to the remote kernels, use `ExportEnvironment` or the package `Parallel`VirtualShared``, which is part of PCT.

Iterators, such as `Table` and `Do`, work in the same way with respect to the iterator variable. Therefore, a statement like the following will not do the expected thing. The variable `i` will not have a value on the remote kernel.

```
Table[RemoteEvaluate[i^2, link], {i, 1, 10}]
```

You can use the following command to accomplish the intended iteration on the remote kernel. This substitutes the value of `i` into the argument of `RemoteEvaluate`.

```
Table[With[{i = i}, RemoteEvaluate[i^2, link]], {i, 1, 10}]
```

Pattern variables, constants, and pure function variables will work as expected on the remote kernel. Each of the following three examples will produce the expected result.

```
Function[i, RemoteEvaluate[i^2]][5]

f[i_] := RemoteEvaluate[i^2]

With[{i = 5}, RemoteEvaluate[i^2]]
```

Parallel Evaluation of Expressions

Dispatching Evaluations to Remote Kernels

`ParallelDispatch[`$h[e_1, e_2, \ldots, e_n]$`, {`$k_1, k_2, \ldots, k_m$`}]`
: evaluates e_i on kernel k_i and returns $h[r_1, r_2, \ldots r_n]$, where r_i is the result of evaluating e_i; the default list of kernels is `$Slaves`

`ParallelEvaluate[`$h[e_1, e_2, \ldots, e_n]$`, `f`, `$comb$`]`
: evaluates $f[h[e_1, e_2, \ldots, e_n]]$ in parallel by distributing chunks $f[h[e_i, e_{i+1}, \ldots, e_{i+k}]]$ to all kernels and combining the results with $comb[]$

`ParallelEvaluate[`$h[e_1, e_2, \ldots, e_n]$`, `f`]`
: the default combiner $comb$ is h, if h has attribute `Flat`, and `Join` otherwise

`ParallelEvaluate[`$h[e_1, e_2, \ldots, e_n]$`]`
: the default function f is `Identity`

Basic parallel dispatch of evaluations.

In ParallelDispatch[$h[e_1, e_2, \ldots, e_n], \{k_1, k_2, \ldots, k_m\}$], the number m of kernels must be at least as large as the number n of expressions. ParallelDispatch has the attribute HoldFirst so that $h[e_1, e_2, \ldots, e_n]$ is not evaluated on the master kernel before the parallelization.

ParallelDispatch[$\{e_1, e_2, \ldots, e_n\}, \{k_1, k_2, \ldots, k_n\}$] is equivalent to Receive[Inner[Send, $\{k_1, k_2, \ldots, k_n\}, \{e_1, e_2, \ldots, e_n\}$]].

ParallelEvaluate[$h[e_1, e_2, \ldots, e_n], f, comb$] breaks $h[e_1, e_2, \ldots, e_n]$ into as many pieces $h[e_i, e_{i+1}, \ldots, e_{i+k}]$ as there are remote kernels, evaluates $f[h[e_i, e_{i+1}, \ldots, e_{i+k}]]$ in parallel (using ParallelDispatch[]), then combines the results r_i using $comb[r_1, r_2, \ldots, r_m]$. ParallelEvaluate has the attribute HoldFirst so that $h[e_1, e_2, \ldots, e_n]$ is not evaluated on the master kernel before the parallelization.

The size of the pieces of the input expression is chosen to be proportional to the remote processor speed estimates for optimal load balancing.

ParallelEvaluate

ParallelEvaluate is a general and powerful command with default values for its arguments that are suitable for evaluating elements of containers such as lists and associative functions.

Evaluating List-like Containers

If the result of applying the function f to a list is again a list, ParallelEvaluate[$\{e_1, e_2, \ldots, e_n\}, f$] simply applies f to pieces of the input list and joins the partial results together.

```
In[1]:= ParallelEvaluate[{1, 2, 3, 4, 5, 6, 7, 8, 9}, Prime]

Out[1]= {2, 3, 5, 7, 11, 13, 17, 19, 23}
```

The result is the same as that of Prime[{1,2,3,4,5,6,7,8,9}], but the computation is done in parallel.

The default function is Identity, therefore, ParallelEvaluate[$\{e_1, e_2, \ldots, e_n\}$] simply evaluates the elements e_i in parallel.

```
In[2]:= ParallelEvaluate[{1 + 2, 2 + 3, 3 + 4, 4 + 5, 5 + 6}]

Out[2]= {3, 5, 7, 9, 11}
```

If the result of applying the function f to a list is *not* a list, a custom combiner has to be chosen.

The function `Function[li,Count[li,_?OddQ]` counts the number of odd elements in a list. To find the total number of odd elements, add the partial results together.

```
In[3]:= ParallelEvaluate[{1, 2, 3, 4, 5, 6, 7, 8, 9}, Function[li, Count[li, _?OddQ]], Plus]

Out[3]= 5
```

Evaluating Associative Operations

If the operation h in $h[e_1, e_2, \ldots, e_n]$ is associative (has attribute `Flat`), the identity

$$h[e_1, e_2, \ldots, e_n] = h[h[e_1, e_2, \ldots, e_i], h[e_{i+1}, e_{i+2}, \ldots, e_n]]$$

holds; with the default combiner being h itself, the operation is parallelized in a natural way. Here all numbers are added in parallel.

```
In[4]:= ParallelEvaluate[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9]

Out[4]= 45

In[5]:= ParallelEvaluate[GCD[4, 6, 8, 10]]

Out[5]= 2
```

Parallel Mapping and Iterators

The commands in this section are fundamental to parallel programming in *Mathematica*.

`ParallelEvaluate[h[`e_1`,`e_2`,...]]`	evaluates the elements e_i in parallel and returns `h[`r_1`,`r_2`, ...]`, where r_i is the result of evaluating e_i
`ParallelMap[`f`, h[`e_1`,`e_2`, ...]]`	evaluates `h[`$f[e_1]$`,`$f[e_2]$`, ...]` in parallel.
`ParallelTable[`$expr$`,{`i`,`i_0`,`i_1`,`di`},{`j`,`j_0`,`j_1`,`dj`},...]`	builds `Table[`$expr$`,{`i`,`i_0`,`i_1`,`di`},{`j`,`j_0`,`j_1`,`dj`},...]` in parallel; parallelization occurs along the first (outermost) iterator `{`i`,`i_0`,`i_1`,`di`}`
`ParallelSum[...]`, `ParallelProduct[...]`	computes sums and products in parallel

Parallel evaluation, mapping, and tables.

`ParallelEvaluate` has attribute `HoldFirst`, so that its argument is not evaluated on the local kernel. After receiving the results, r_i, the expression `h[`r_1`, `r_2`, ...]` is further evaluated normally on the local kernel. The symbol `List` is often used as the head `h`.

`ParallelMap[`f`, h[`e_1`, `e_2`, ...]]` is a parallel version of f `/@ h[`e_1`, `e_2`, ...]` evaluating the individual f`[`e_i`]` in parallel rather than sequentially.

`ParallelEvaluate` and related commands use all available remote kernels on the list `$Slaves`.

Side Effects

Unless you use shared variables, the parallel evaluations performed are completely independent and cannot influence each other. Furthermore, any side effects, such as assignments to variables, that happen as part of evaluations will be lost. The only effect of a parallel evaluation is that its result is returned at the end.

Examples

First, load the package and then start several remote kernels.

```
In[1]:= Needs["Parallel`"]

          Parallel Computing Toolkit 2.0 (November 11, 2004)

          Created by Roman E. Maeder
```

The sine function is applied to the given arguments. Each computation takes place on a remote kernel.

```
In[2]:= ParallelMap[Sin, {0, π, 1.0}]

Out[2]= {0, 0, 0.841471}
```

This particular computation is almost certainly too trivial to benefit from parallel evaluation. The overhead required to send the expressions `Sin[0]`, `Sin[`π`]`, and so on to the remote kernels and to collect the results will be larger than the gain obtained from parallelization.

Factoring integers of the form 111…1 takes more time, so this computation can benefit from parallelization.

```
In[3]:= ParallelMap[FactorInteger, (10^Range[20, 30] - 1)/9]

Out[3]= {{{11, 1}, {41, 1}, {101, 1}, {271, 1}, {3541, 1}, {9091, 1}, {27961, 1}},
         {{3, 1}, {37, 1}, {43, 1}, {239, 1}, {1933, 1}, {4649, 1}, {10838689, 1}},
         {{11, 2}, {23, 1}, {4093, 1}, {8779, 1}, {21649, 1}, {513239, 1}},
         {{11111111111111111111, 1}}, {{3, 1}, {7, 1}, {11, 1}, {13, 1},
          {37, 1}, {73, 1}, {101, 1}, {137, 1}, {9901, 1}, {99990001, 1}},
         {{41, 1}, {271, 1}, {21401, 1}, {25601, 1}, {182521213001, 1}},
         {{11, 1}, {53, 1}, {79, 1}, {859, 1}, {265371653, 1}, {1058313049, 1}},
         {{3, 3}, {37, 1}, {757, 1}, {333667, 1}, {440334654777631, 1}}, {{11, 1},
          {29, 1}, {101, 1}, {239, 1}, {281, 1}, {4649, 1}, {909091, 1}, {121499449, 1}},
         {{3191, 1}, {16763, 1}, {43037, 1}, {62003, 1}, {77843839397, 1}},
         {{3, 1}, {7, 1}, {11, 1}, {13, 1}, {31, 1}, {37, 1}, {41, 1},
          {211, 1}, {241, 1}, {271, 1}, {2161, 1}, {9091, 1}, {2906161, 1}}}
```

Alternatively, you can use `ParallelTable`. A list of the number of factors in $\underbrace{11\ldots1}_{i}$ is generated here.

```
In[4]:= ParallelTable[{i, Plus @@ (#[[2]] &) /@ FactorInteger[(10^i - 1)/9]}, {i, 20, 30}] //
          TableForm

Out[4]//TableForm=
20      7
21      7
22      7
23      1
24      10
25      5
26      6
27      7
28      8
29      5
30      13
```

Automatic Parallelization

`ParallelEvaluate[`*cmd*`[`*list*`,`*arguments*`...]]` recognizes if *cmd* is a *Mathematica* function that operates on a list or other long expression in a way that can be easily parallelized and performs the parallelization automatically. You do not need to figure out suitable *f* and *comb* for `ParallelEvaluate[`*list*`,` *f*`,` *comb*`]`. The list of commands that can be parallelized is kept in the variable `$ParallelCommands`.

```
In[5]:= $ParallelCommands

Out[5]= {Cases, Select, Count, MemberQ, FreeQ,
         Map, Apply, Outer, Inner, Dot, Table, Sum, Product}

In[6]:= ParallelEvaluate[Count[{1, 2, 3, 4, 5, 6, 7}, _?PrimeQ]]

Out[6]= 4

In[7]:= ParallelEvaluate[Map[f, {a, b, c, d, e, f}]]

Out[7]= {f[a], f[b], f[c], f[d], f[e], f[f]}

In[8]:= ParallelEvaluate[{a, b, c, d}.{{w1, w2}, {x1, x2}, {y1, y2}, {z1, z2}}]

Out[8]= {a w1 + b x1 + c y1 + d z1, a w2 + b x2 + c y2 + d z2}
```

Not all uses of these commands can be parallelized. A message is generated and the evaluation is performed sequentially on the master kernel if necessary.

```
In[9]:= ParallelEvaluate[Apply[f, g[a, b, c, d]]]

        ParallelEvaluate::nopar1 :
         f @@ g[a, b, c, d] cannot be parallelized; evaluating it sequentially.

Out[9]= f[a, b, c, d]
```

4 Concurrency: Managing Parallel Processes

Processes and Processors

A process is simply a *Mathematica* expression being evaluated. A processor is a remote kernel that performs such evaluations.

The command `RemoteEvaluate` discussed in the chapter Parallel Evaluation will send an evaluation to an explicitly given processor, requiring you to keep track of available processors and processes yourself. The scheduling functions discussed in this chapter perform these functions for you. You can create any number of processes, many more than the number of available processors. If more processes are created than there are processors, the remaining processes will be queued and serviced when a processor becomes available.

Starting and Waiting for Processes

The two basic commands are `Queue[`*expr*`]` to line up an expression for evaluation on any available processor, and `Wait[`*pid*`]` to wait until a given process has finished.

Each process in the queue is identified by its unique pid.

`Queue[`*cmd*`]`	submits *cmd* for evaluation on a remote kernel and returns the queued job's pid
`Queue[{`*vars…*`}, `*cmd*`]`	builds a closure for the local values of the given variables before sending *cmd* to a remote kernel
`QueueRun[]`	checks all remote kernels for available results and submits waiting jobs to available remote kernels
`DoneQ[`*pid*`]`	returns `True` if the given process has finished
`Wait[`*pid*`]`	waits for the given process to finish and returns its result
`Wait[{`pid_1`, `pid_2`, …}]`	waits for all given processes and returns the list of results
`Wait[`*expr*`]`	waits for all pids contained in *expr* to finish and replaces them by the results of the respective process
`WaitOne[{`pid_1`, `pid_2`, …}]`	waits for one of the given processes to finish; it returns {*res*, *id*, *ids*}, where *id* is the pid of the finished process, *res* is its result, and *ids* is the list of remaining pids

Queuing processes.

`WaitOne` is nondeterministic. It returns an arbitrary process that has finished. If no process has finished, it calls `QueueRun` until a result is available. The third element of the resulting list is suitable as an argument of another call to `WaitOne`.

The functions `Queue` and `Wait` implement *concurrency*. You can start arbitrarily many processes, and they will all be evaluated eventually on any available remote processors. When you need a particular result, you can wait for any particular pid, or you can wait for all results using repeated calls to `WaitOne`.

`QueueRun[]` returns `True` if at least one job was submitted to a remote kernel or one result received from a kernel, and `False` otherwise. Normally, you should not have to run `QueueRun` yourself. It is called at appropriate places inside `Wait` and other functions. You need it only if you implement your own *main loop* for a concurrent program.

`Load[]`	gives the sum of the lengths of the input queues of all remote kernels
`Load[`*kernel*`]`	gives the length of the input queue of a remote kernel
`$QueueLength`	gives the length of the input queue of commands submitted with `Queue` but not yet assigned to an available remote kernel
`ResetQueues[]`	waits for all running processes and abandons any queued processes

System load and input queue size.

Basic Usage

To try the examples here, load PCT and then start a few local or remote kernels as described in the Introduction chapter.

```
In[1]:= Needs["Parallel`"]
```

Queue the evaluation `1+1` for processing on a remote kernel. Note that `Queue` has the attribute `HoldAll` to prevent evaluation of the expression before queueing. The value returned by `Queue` is the pid of the queued process.

```
In[2]:= j1 = Queue[1 + 1]

Out[2]= pid1
```

After queuing a process, you can perform other calculations and eventually collect the result. If the result is not yet available, `Wait` will wait for it.

```
In[3]:= Wait[j1]

Out[3]= 2
```

You can queue several processes. Here the expression 1^2, 2^2,…, 5^2 is queued for evaluation.

```
In[4]:= pids = Function[i, Queue[i^2]] /@ {1, 2, 3, 4, 5}

Out[4]= {pid2, pid3, pid4, pid5, pid6}
```

Next, wait for any one process to finish.

```
In[5]:= {res, pid, pids} = WaitOne[pids]

Out[5]= {1, pid2, {pid3, pid4, pid5, pid6}}
```

Note the reassignment of `pids` to the list of remaining pids. Repeating the previous evaluation until the `pids` list becomes empty allows you to drain the queue.

```
In[6]:= {res, pid, pids} = WaitOne[pids]

Out[6]= {4, pid3, {pid4, pid5, pid6}}
```

You can also wait for all of the remaining processes to finish.

```
In[7]:= Wait[pids]

Out[7]= {9, 16, 25}
```

A Note on the Use of Variables

If an expression *e* in `Queue[e]` involves variables with assigned values, care must be taken to ensure that the remote kernels have the same variable values defined. Unless you use `ExportEnvironment` or shared variables, locally defined variables will not be available to remote kernels. See the section Values of Variables in the chapter Parallel Evaluation for more information.

Here are a few common cases where there may be problems.

This assigns the value `2` to the variable `a` in the local master kernel.

```
In[8]:= a = 2;
```

You want to evaluate the expression `Head[a]` on a remote kernel. The result is not `Integer`, as it is on the local kernel, because on the remote kernel the symbol `a` does not have a value.

```
In[9]:= Head[a]

Out[9]= Integer

In[10]:= Wait[Queue[Head[a]]]

Out[10]= Symbol
```

You can use a local constant, and even name it `a`, to textually insert the value of `a` into the argument of `Queue`.

```
In[11]:= With[{a = a}, Wait[Queue[Head[a]]]]

Out[11]= Integer
```

To make this frequent case simpler, you can use an optional first argument in `Queue` to declare the variables. The variables will then be inserted into the expression in the second argument.

```
In[12]:= Wait[Queue[{a}, Head[a]]]

Out[12]= Integer
```

The syntax `Queue[{`*vars*`...}, `*expr*`]` is by design similar to `Function[{`*vars*`...}, `*expr*`]`. Both form *closures* where the variables are bound to their values.

Iterator variables behave in the same way. In the following two outputs, the parallel evaluation does not give the correct result.

```
In[13]:= Table[i^2, {i, 1, 10}]

Out[13]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}

In[14]:= Wait[Table[Queue[i^2], {i, 1, 10}]]

Out[14]= {i^2, i^2, i^2, i^2, i^2, i^2, i^2, i^2, i^2, i^2}
```

Insert the iterator variable as a constant or declare a closure to ensure that you are getting correct results, as is done with the following command.

```
In[15]:= Wait[Table[Queue[{i}, i^2], {i, 1, 10}]]

Out[15]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Note that ParallelTable[] treats the iterator variable correctly.

```
In[16]:= ParallelTable[i^2, {i, 1, 10}]

Out[16]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Working with Process IDs

Wait[{pid_1, pid_2, …}] is merely a simple form of a general mechanism to parallelize computations. Wait can take any expression containing pids in its arguments and will wait for all associated processes to finish. The pids will then be replaced by the results of their processes.

You can view Wait as the inverse of Queue; that is, Wait[Queue[*expr*]] gives *expr*, evaluated on a remote kernel, just as *expr* itself is evaluated locally. Further, Wait[…Queue[e_1] … Queue[e_n] …] is equivalent to … e_1 … e_n …, where each e_i is evaluated in parallel. Here the ellipses represent an arbitrary surrounding computation.

The pids generated by an instance of Queue should be left intact and should neither be destroyed nor duplicated before Wait performs its task. The reason is that each of them represents an ongoing parallel computation whose result should be collected exactly once.

Examples of expressions that leave pids intact follow.

- pids in a list are safe, because the list operation does not do anything to its arguments; it merely keeps them together. Nested lists are also safe for the same reason.

  Wait[{Queue[e_1], …, Queue[e_n]}]

- pids are symbolic objects that are not affected by Plus. They may be reordered, which is irrelevant. Most arithmetic operations are safe.

  Wait[Queue[e_1] + … + Queue[e_n]]

- Mapping a function involving Queue onto a list is safe because the result will contain the list of the pids.

  Wait[Map[Queue[… # …] &, {e_1, …, e_n}]]

- `Table` returns lists of pids and is, therefore, safe.

```
Wait[Table[Queue[{i}, expr], {i, 1, 10}]]
```

Examples of expressions that are not safe include the following.

- The `Head` operation will destroy the symbolic pid, as will other structural operations such as `Length`, `ByteCount`, and so on.

```
Wait[Head[Queue[e]]]
```

- Multiplying a pid by `0` will destroy it.

```
Wait[0 * Queue[e]]
```

- `Do` does not return anything, so all pids are lost. A similar case is `Scan`.

```
Wait[Do[Queue[{i}, expr], {i, 1, 10}]]
```

To recover from a computation where pids were destroyed or duplicated, use the command `ResetQueues[]` or `ResetSlaves[]`.

`ProcessID[`*pid*`]`	a unique integer identifying the process
`Process[`*pid*`]`	the expression representing the process
`Scheduling[`*pid*`]`	the priority assigned to the process
`ProcessState[`*pid*`]`	the state of the process: queued, running, finished

Properties of pids.

Here several processes are queued.

```
In[1]:= ids = Queue[2 + #, Scheduling → #] & /@ Range[6]

Out[1]= {pid1, pid2, pid3, pid4, pid5, pid6}
```

Because the scheduler has not yet been running, all the processes are still queued for evaluation.

```
In[2]:= TableForm[
         Function[pid,
           Through[{Identity, ProcessID, Process, Scheduling, ProcessState}[pid]]] /@ ids,
         TableHeadings → {None, {"pid", "ID", "expr", "prio", "state"}}]
```

```
Out[2]//TableForm=
       pid       ID      expr      prio      state
       pid1      1       2 + 1     1         queued
       pid2      2       2 + 2     2         queued
       pid3      3       2 + 3     3         queued
       pid4      4       2 + 4     4         queued
       pid5      5       2 + 5     5         queued
       pid6      6       2 + 6     6         queued
```

To demonstrate how it works, the scheduler is invoked by hand.

```
In[3]:= QueueRun[]; QueueRun[]

Out[3]= False
```

Now some processes are running on the available processors; some may already have finished.

```
In[4]:= TableForm[
         Function[pid,
           Through[{Identity, ProcessID, Process, Scheduling, ProcessState}[pid]]] /@ ids,
         TableHeadings → {None, {"pid", "ID", "expr", "prio", "state"}}]
```

```
Out[4]//TableForm=
       pid       ID      expr      prio      state
       pid1      1       2 + 1     1         finished[3]
       pid2      2       2 + 2     2         finished[4]
       pid3      3       2 + 3     3         running[slave2[n1]]
       pid4      4       2 + 4     4         running[slave3[n2]]
       pid5      5       2 + 5     5         queued
       pid6      6       2 + 6     6         queued
```

`Wait[]` invokes the scheduler until all processes are finished and returns their results. Note that the priorities are not used with the default queue type, see the section The Scheduler, later in this chapter.

```
In[5]:= Wait[ids]

Out[5]= {3, 4, 5, 6, 7, 8}
```

Latency Hiding

The *latency* is the communication overhead, the time period between the completion of one request and the start of servicing a new one.

MathLink is a buffered stream. You can send additional requests before receiving all outstanding results. If you can keep the buffers full at all times, there will be no latency; each remote processor will always be busy.

To turn on latency hiding, set `$LoadFactor` to a value larger than `1`, such as `2` through `5`. `$LoadFactor` determines the maximum number of computations sent to a slave kernel before at least one of the results is read. Here are two example commands that set latency hiding.

```
$LoadFactor = 3; (* enable latency hiding *)

$LoadFactor = 1; (* disable latency hiding *)
```

To benefit from latency hiding, you should plan to create many more processes than there are remote kernels. You can create more processes using `Queue` or an appropriate parallel mapping or table construct. If the number of processes is larger than `$LoadFactor*Length[$Slaves]`, automatic *load balancing* is an added benefit of this scheme. The faster processors will automatically service more processes.

Do not use latency hiding if you plan to create exactly one process for each remote kernel.

You should develop your program with the default `$LoadFactor = 1` for easier debugging. Once your program runs correctly, you can try to increase `$LoadFactor` and measure the effect on the elapsed time of your computation. The smaller the remote computations are, the greater the benefit from latency hiding.

Latency hiding is incompatible with virtual shared memory. When you load `Parallel`Virtual`Shared`, the value of `$LoadFactor` will be permanently set to `1`.

Examples

Before evaluating these examples, load the package and then start several remote kernels.

```
In[1]:= Needs["Parallel`Parallel`"]
```

An Infinite Calculation

If you want to verify that the polynomials $\sum_{i=1}^{n+1} i\, x^{i-1}$ for $n = 1, 2, \ldots$ are all irreducible (an open conjecture), factor the polynomials and then check that the length of the list of factors is `2`, one factor being the overall numerical factor.

This computation will continue forever. To stop it, abort the local evaluation by pressing CMD+. or choosing Kernel ▹ Abort Evaluation. After the abort, collect any waiting results, as follows.

Here is the definition of the polynomial in x with degree n.

```
In[2]:= poly[n_, x_] := Sum[i x^(i-1), {i, 1, n+1}]
```

You then make this definition known to all remote kernels.

```
In[3]:= ExportEnvironment[{poly}];
```

For better performance, you turn on latency hiding by setting `$LoadFactor` to a value larger than `1`.

```
In[4]:= $LoadFactor = 2;
```

Now you can start the computation, requiring that it print each result as it goes on. To stop the computation, abort it by choosing Kernel ▹ Abort Evaluation or pressing CMD+.. The explicit call of `QueueRun[]` is necessary in such examples where you program your own scheduling details.

```
In[5]:= d = 100; ids = {};
        While[True,
         While[$QueueLength == 0,
              AppendTo[ids, Queue[{d}, {d, Length[FactorList[poly[d, x]]]}]];
           d++;
          If[! QueueRun[], Break[]];
         ];
         If[Length[ids] > 0,
          {res, id, ids} = WaitOne[ids];
          Print[res]
         ];
        ]

           {101, 2}

           {103, 2}

           {100, 2}

           {102, 2}

Out[5]= $Aborted
```

Do not forget to collect the orphaned processes after an interrupt.

```
In[6]:= Wait[ids]

Out[6]= {{104, 2}, {105, 2}, {106, 2}, {107, 2}, {108, 2}}
```

Automatic Process Generation

A general way to parallelize many kinds of computation is to replace a function *g* occurring in a functional operation by `Composition[Queue, g]`. This new operation will cause all instances of calls of the function *g* to be queued for parallel evaluation. The result of this composition is a pid that will appear inside the structure constructed by the outer computation where *g* occurred. To put back the results of the computation of *g*, wrap the whole expression in `Wait`. `Wait` will replace any pid inside its expression by the result returned by the corresponding process.

Here are a few examples of such functional compositions.

Parallel mapping

A parallel version of `Map` is easy to develop. The sequential `Map` wraps a function `f` around all elements in a list.

```
In[1]:= Map[f, {a, b, c, d}]

Out[1]= {f[2], f[b], f[c], f[109]}
```

Simply use `Composition[Queue, f]` instead of `f` to schedule all mappings for parallel execution. The result is a list of pids.

```
In[2]:= Map[Composition[Queue, f], {a, b, c, d}]

Out[2]= {pid16, pid17, pid18, pid19}
```

Finally, simply wait for the processes to finish. Every pid will be replaced with the result of its associated process.

```
In[3]:= Wait[%]

Out[3]= {f[2], f[b], f[c], f[109]}
```

Parallel inner products

To see how this works for symbolic inner products, assume you want a generalized inner product where `d` is the common last dimension of `a` and first dimension of `b`. Think of `p` as `Plus` and `t` as `Times`.

$$(a.b)_{i_1\, i_2 \ldots i_{n-1}\, k_2\, k_3 \ldots k_m} = p[t[a_{i_1\, i_2 \ldots i_{n-1}\, 1},\ b_{1\, k_2\, k_3 \ldots k_m}],\ \ldots,\ t[a_{i_1\, i_2 \ldots i_{n-1}\, d},\ b_{d k_2\, k_3 \ldots k_m}]]$$

Here is an example with `d = 2`.

```
In[4]:= Inner[t, Array[a, {2, 2}], Array[b, {2, 2}], p]

Out[4]= {{p[t[2[1, 1], b[1, 1]], t[2[1, 2], b[2, 1]]],
           p[t[2[1, 1], b[1, 2]], t[2[1, 2], b[2, 2]]]},
          {p[t[2[2, 1], b[1, 1]], t[2[2, 2], b[2, 1]]],
           p[t[2[2, 1], b[1, 2]], t[2[2, 2], b[2, 2]]]}}
```

You can use `Composition[Queue,p]` in place of `p` in the previous expression to queue all calculations of `p` for concurrent execution. The result is a tensor of pids.

```
In[5]:= Inner[t, Array[a, {2, 2}], Array[b, {2, 2}], Composition[Queue, p]]

Out[5]= {{pid20, pid21}, {pid22, pid23}}
```

Now, simply wait for all processes in this expression.

```
In[6]:= Wait[%]

Out[6]= {{p[t[2[1, 1], b[1, 1]], t[2[1, 2], b[2, 1]]],
          p[t[2[1, 1], b[1, 2]], t[2[1, 2], b[2, 2]]]},
         {p[t[2[2, 1], b[1, 1]], t[2[2, 2], b[2, 1]]],
          p[t[2[2, 1], b[1, 2]], t[2[2, 2], b[2, 2]]]}}
```

This scheduling is not optimal in terms of communication. One better possibility is to parallelize only along the dimensions of `a` and send the tensor `b` once to each processor before scheduling the jobs. The command `ParallelInner` in the `Parallel`Commands`` package uses this strategy.

Parallel tables and sums

This code generates a 5×5 matrix of random numbers, where each row is computed in parallel.

```
In[7]:= Wait[Table[Queue[Table[Random[], {5}]], {5}]]

Out[7]= {{0.126768, 0.775646, 0.120027, 0.657164, 0.091348},
         {0.617092, 0.954231, 0.80132, 0.237532, 0.856428},
         {0.665655, 0.57283, 0.952782, 0.314881, 0.0194191},
         {0.073742, 0.46934, 0.633294, 0.648699, 0.241248},
         {0.265634, 0.675514, 0.239893, 0.628718, 0.0933279}}
```

Here is a sum whose elements are computed in parallel. Each element of the sum is a numerical integration.

```
In[8]:= Wait[Sum[Queue[{k}, NIntegrate[x^(-1 - 1 / k), {x, 1, ∞}]], {k, 1, 8}]]

Out[8]= 36.
```

Here is the corresponding table of symbolic results. The value of $\int_1^\infty \frac{1}{x^{1+1/k}}\, dx$ is k.

```
In[9]:= TableForm[
         Wait[Table[Queue[{k}, {k, Integrate[x^(-1 - 1 / k), {x, 1, ∞}]}], {k, 1, 8}]],
         TableDepth -> 2]
```

```
Out[9]//TableForm=
1      1
2      2
3      3
4      4
5      5
6      6
7      7
8      8
```

Comparison with ParallelEvaluate

Parallel mapping, tables, and inner products were already introduced in the chapter Parallel Evaluation. Those functions perform a single dispatch of part of the computation on each remote processors. The functions in this chapter generate one process for each subproblem.

If all subproblems take the same amount of time, the functions such as `ParallelMap[]`, `ParallelTable[]` are faster. However, if the computation times of the subproblems are different, and not easy to estimate in advance, it can be better to use `Wait[… Queue[] … ]` as described in this section. If the number of processes generated is larger than the number of remote kernels, this method performs automatic load balancing, because jobs are assigned to a kernel as soon as the previous job is done, and all kernels are kept busy all the time.

Tracing

To observe how processes are scheduled, you can use tracing. To use these features, you have to load the debugging package before loading PCT and starting several kernels.

```
In[1]:= Needs["Parallel`Debug`"]
```

```
In[2]:= Needs["Parallel`"]
```

Now you can enable `Queueing` tracing.

```
In[3]:= SetOptions[$DebugObject, Trace → {Queueing}]

Out[3]= {Trace → {Queueing}}
```

Here several processes are queued showing how the queue grows in size.

```
In[4]:= ids = Queue[2 + #] & /@ Range[4]

          Queueing: pid1 queued (0)

          Queueing: pid2 queued (1)

          Queueing: pid3 queued (2)

          Queueing: pid4 queued (3)

Out[4]= {pid1, pid2, pid3, pid4}
```

`Wait[]` invokes the scheduler, which sends queued jobs to idle processors, collects results, and hands them back to the application.

```
In[5]:= Wait[ids]

          Queueing: pid1 on slave1[n1]

          Queueing: pid2 on slave2[n2]

          Queueing: pid1 done

          Queueing: pid2 done

          Queueing: pid3 on slave1[n1]

          Queueing: pid4 on slave2[n2]

          Queueing: pid1 dequeued

          Queueing: pid2 dequeued

          Queueing: pid3 done

          Queueing: pid4 done

          Queueing: pid3 dequeued

          Queueing: pid4 dequeued

Out[5]= {3, 4, 5, 6}
```

To turn off tracing when you are done, use the following.

```
In[6]:= SetOptions[$DebugObject, Trace → {}]

Out[6]= {Trace → {}}
```

The Scheduler

Whenever `Queue[]` is called, a process is entered into a queue on the master kernel. The scheduler `Queue:Run[]` selects the first process in the queue as soon as a remote kernel is available. By choosing different queue implementations, you can specify which process should be considered the first one. PCT provides a number of queues that you can use, or you can write your own. Queues are implemented in packages that you can load as needed.

`Parallel`Queue`FIFO`	`FIFOQueue`: first in, first out
`Parallel`Queue`Priority`	`priorityQueue`: user-definable priorities
`Parallel`Queue`LIFO`	`LIFOQueue`: last in, first out (reverse)
`Parallel`Queue`Interface`	interface definitions for all queues

Packages and queue types.

`$Queue`	the queue of waiting processes
`$QueueLength`	the number of processes in the queue
`$QueueType`	gives the current queue type
`SetQueueType[`*queue*`]`	changes the queue type to *queue*

Manipulating the queue.

This is the current queue type (the default).

```
In[1]:= oldType = $QueueType

Out[1]= Parallel`Queue`FIFO`FIFOQueue
```

To use priority queues, for example, load the corresponding package.

```
In[2]:= Needs["Parallel`Queue`Priority`"]

In[3]:= ?priorityQueue

          priorityQueue[] creates an empty queue.
```

Then change the queue type.

```
In[4]:= SetQueueType[priorityQueue]

Out[4]= priorityQueue
```

Now you can generate processes with optional priorities.

`Queue[`*expr*`, Scheduling->`*p*`]`	queue *expr* with priority *p*

Specifying priorities.

```
In[5]:= ids = Queue[2 #, Scheduling → #] & /@ Range[6]

Out[5]= {pid1, pid2, pid3, pid4, pid5, pid6}
```

Each queue type implements the method `Normal[]` to give a list of the queue's contents.

```
In[6]:= Normal[$Queue]

Out[6]= {pid6, pid5, pid4, pid3, pid2, pid1}
```

The scheduler will schedule jobs with higher priority first. The order of the results is not affected.

```
In[7]:= Wait[ids]

Out[7]= {2, 4, 6, 8, 10, 12}
```

This resets the queue type to the default.

```
In[8]:= SetQueueType[oldType]

Out[8]= Parallel`Queue`FIFO`FIFOQueue
```

5 Remote Definitions

Remote kernels do not have access to the values of variables defined in the local master kernel, nor do they have access to locally defined functions.

PCT contains a command `ExportEnvironment` that makes it easy to transport local variables and definitions to all remote kernels. The main advantage of this method is that PCT does not need to be installed on the remote kernels. All definitions are sent through the existing connection to the remote kernels.

Exporting Definitions

`ExportEnvironment[{`s_1`, `s_2`, ... }]`	exports all definitions for symbols s_i to all remote kernels
`ExportEnvironment[`s_1`, `s_2`, ... ]`	the same as `ExportEnvironment[{`s_1`, `s_2`, ...}]`
`ExportEnvironment["`*Context*`"]`	exports all definitions for all symbols in the given context

Exporting definitions.

`ExportEnvironment` has the attribute `HoldAll` to prevent the evaluation of the symbols.

`ExportEnvironment` exports the following kinds of definitions: `OwnValues`, `DownValues`, `SubValues`, `UpValues`, `DefaultValues`, `NValues`.

`ExportEnvironment` sets the attributes of the remote symbols equal to the locally defined attributes, except for attributes such as `Protected` and `Locked`.

Any old definitions existing on the remote side are cleared before the new definitions are made.

Examples

Before evaluating these examples, load PCT and then start several remote kernels.

```
In[1]:= Needs["Parallel`Debug`"]
        Needs["Parallel`"]
```

Define a variable with a value and a function with attributes.

```
In[2]:= var = 5;
```

```
In[3]:= SetAttributes[func, Listable]
        func[r_] := r^2
```

The remote kernels do not know the symbol `var` as demonstrated with the next evaluation.

```
In[5]:= RemoteEvaluate[Names["var"]]

Out[5]= {{}, {}}
```

Here is a subtle point. The following remote evaluation seems to work, even though the symbols are not defined on the remote side.

```
In[6]:= RemoteEvaluate[func[var]]

Out[6]= {25, 25}
```

The reason is that the remote kernels return the unevaluated expression `func[var]`, because the function and variable are not defined on the remote kernel. The master kernel evaluates the returned results further, but it does so sequentially.

You can easily produce an example where the difference between remote and local evaluation becomes apparent.

On the local kernel, the symbol `var` evaluates to `5`, and the `Head` of `5` is `Integer`.

```
In[7]:= Head[var]

Out[7]= Integer
```

On the remote kernels, `var` stays a symbol, and its head is `Symbol`.

```
In[8]:= RemoteEvaluate[Head[var]]

Out[8]= {Symbol, Symbol}
```

You can export the local definitions to the remote kernels.

```
In[9]:= ExportEnvironment[var, func]
```

Now the remote evaluation gives the same result as the local one.

```
In[10]:= RemoteEvaluate[Head[var]]

Out[10]= {Integer, Integer}
```

Exporting Contexts

`ExportEnvironment["`*Context*`"]` exports all definitions for all symbols in the given context. Thus, you can use the following to make all your interactively entered definitions known to the remote kernels.

```
In[11]:= ExportEnvironment["Global`"]

Out[11]= {Global`}
```

Exporting the context of a package you have loaded may not have the same effect on the remote kernels as loading the package on each remote kernel. The reason is that loading a package may perform certain initializations and it may also define auxiliary functions in other contexts (such as a private context). Also, a package may load additional auxiliary packages that establish their own contexts.

`ExportEnvironment["`*Context*`"]` is useful for exporting contexts for definitions that you have explicitly set up to be used on remote kernels. There is a separate command `RemoteNeeds[]` for remote loading of packages.

`ClearSlaves[]` clears any definitions for symbols in the `Global`` context on remote kernels.

Loading Packages on Remote Kernels

`RemoteNeeds["`*Context*`"]`	calls `Needs["`*Context*`"]` in all remote kernels

Loading packages.

`RemoteNeeds["`*Context*`"]` is essentially equivalent to `RemoteEvaluate[Needs["`*Context*`"]]`, but it is remembered and any newly launched remote kernels will be initialized as well.

Exporting the context of a package that you have loaded may not have the same effect on the remote kernels as loading the package on each remote kernel with `RemoteNeeds[]`. The reason is that loading a package may perform certain initializations and it may also define auxiliary functions in other contexts (such as a private context). Also, a package may load additional auxiliary packages that establish their own contexts.

The next two commands load the standard package `Geometry`Rotations`` on the master kernel and all remote kernels.

```
In[12]:= Needs["Geometry`Rotations`"]
```

```
In[13]:= RemoteNeeds["Geometry`Rotations`"];
```

Note that packages other than the standard packages available to the master kernel may not be available on remote kernels.

Example: Eigenvalues of Matrices

Definitions

The parameter `prec` gives the desired precision for the computation of the eigenvalues of a random $n \times n$ matrix.

```
In[1]:= prec = 18;
```

The function `mat` generates a random $n \times n$ matrix with numeric entries.

```
In[2]:= mat[n_] := Table[Random[Real, {-1, 1}, prec], {n}, {n}]
```

The function `tf` measures the time it takes to find the eigenvalues.

```
In[3]:= tf[n_] := Timing[Eigenvalues[mat[n]]][[1]]
```

Make all these definitions known to all slave processors with the following command.

```
In[4]:= ExportEnvironment[{prec, tf, mat}];
```

Clear the local value of `prec`, since it is no longer needed. Then check that the definitions have been carried over to the slave processors.

```
In[5]:= Clear[prec];

In[6]:= RemoteEvaluate[prec]

Out[6]= {18, 18, 18}
```

A sample run

Here you measure the time it takes to find the eigenvalues of 5×5 to 25×25 matrices. Because the computations happen on remote computers that differ in their processor speeds, the results do not necessarily form an increasing sequence.

```
In[7]:= ParallelMap[tf, {5, 10, 15, 20, 25}]

Out[7]= {0.03 Second, 0.01 Second, 0.04 Second, 0.173974 Second, 0.288956 Second}
```

Alternatively, you can perform the same computation on each slave processor to measure their relative speeds. Here you find the speed of calculating the eigenvalues of a 20 × 20 matrix on each of three slave processors.

```
In[8]:= RemoteEvaluate[tf[20]]

Out[8]= {0.43 Second, 0.09 Second, 0.164975 Second}
```

6 Virtual Shared Memory

Shared Memory versus Distributed Memory

Special-purpose multiprocessing hardware comes in two types, *shared memory* and *distributed memory*. In a shared-memory machine, all processors have access to a common main memory. In a distributed-memory machine, each processor has its own main memory, and the processors are connected through a sophisticated network. A collection of networked PCs is also a kind of distributed-memory parallel machine.

Communication between processors is an important prerequisite for all but the most trivial parallel processing tasks. In a shared-memory machine, a processor can simply write a value into a particular memory location, and all other processors can read this value. In a distributed-memory machine, exchanging values of variables involves explicit communication over the network.

Virtual Shared Memory

Virtual shared memory is a programming model that allows processors on a distributed-memory machine to be programmed as if they had shared memory. A software layer takes care of the necessary communication in a transparent way.

PCT uses independent *Mathematica* kernels as parallel processors. It is clear that these kernels do not share a common memory, even if they happen to reside on the same machine. The package `Parallel`VirtualShared`, which is part of PCT, implements virtual shared memory for these remote kernels.

The package is normally set up to be autoloaded the first time you declare a shared variable. To load it explicitly, use

```
Needs["Parallel`VirtualShared`"]
```

The result is a simple programming model. If a variable *a* is shared, any kernel that reads the variable (simply by evaluating it), reads a common value that is maintained by the master kernel. Any kernel that

changes the value of *a*, for example by assigning it with *a* = *val*, will modify the one global copy of the variable *a*, so that all other kernels that subsequently read the variable will see its new value.

The drawback of a shared variable is that every access for read or write requires communication over the network, so it is slower than access to a local unshared variable.

Declaring Shared Variables

`SharedVariables[{`s_1`, `s_2`, ...}]`	declares the symbols s_i as shared variables
`SharedVariables[`s_1`, `s_2`, ...]`	same as `SharedVariables[{`s_1`, `s_2`, ...}]`
`SharedDownValues[{`f_1`, `f_2`, ...}]`	declares the symvols f_i as shared functions or data types
`SharedDownValues[`f_1`, `f_2`, ...]`	same as `SharedDownValues[{`f_1`, `f_2`, ...}]`

Declaring shared variables and downvalues.

The command `SharedVariables` has the attribute `HoldAll` to prevent evaluation of the given variables, which usually have values.

The effect of `SharedVariables` or `SharedDownValues` is that all currently connected and newly launched remote kernels will perform all accesses to the shared variables through the master kernel.

`$SharedVariables`	the list of currently shared variables (wrapped in `Hold[]`)
`$SharedDownValues`	the list of currently shared downvalues (wrapped in `Hold[]`)
`ClearShared[`s_1`, `s_2`, ...]`	unshares the given variables or downvalues
`ClearShared[]`	unshares all variables and downvalues

Manipulating the set of shared variables and downvalues.

Clearing kernels with `ClearSlaves[]` will also clear any shared variables and downvalues.

SharedVariables

A variable *s* that has been declared shared with `SharedVariables[s]` exists only in the master (local) kernel. The following operations on a remote kernel are redefined so that they have the described effect.

s	evaluation of the variable will consult the master kernel for the variable's current value
s = *e*, *s* := *e*	assigning a value to *s* will perform the assignment in the master kernel
s++, *s*--, ++*s*, --*s*	the increment/decrement operation is performed in the master kernel (this operation is atomic and can be used for synchronization)
`TestAndSet[`*s*`,`*e*`]`	if *s* has no value or its value is `Null`, set the value to *e*; otherwise, do nothing and return the current value of *s* (this operation is atomic and can be used for synchronization)
`Part[Unevaluated[`*s*`],`*i*`]`	extracts a part of *s*; the operation will transmit only the requested part over the *MathLink* connection, not the whole value of *s*
s`[[`*i*`]]` = *e*	replaces the specified part of the variable with a new value; the old value of *s* must have the necessary structure to permit the part assignment

Operations on shared variables.

For technical reasons, every shared variable must have a value. If the variable in the master kernel does not have a value, it is set to `Null`.

Note that other forms of assignments, such as conditional assignments involving side conditions, are not supported.

The customary form of part extraction, *s*`[[`*i*`]]`, will transmit the whole value of *s* to the slave kernels. Use `Part[Unevaluated[`*s*`],`*i*`]` to transmit only the *i*th component.

If a variable is `Protected` at the time you declare it as shared, remote kernels can only access the variable, but not change its value.

SharedDownValues

A symbol *f* that has been declared shared with `SharedDownValues[f]` exists only in the master (local) kernel. The following operations on a remote kernel are redefined so that they have the described effect.

`f[i]`, `f[i, j]`, …	evaluation of the function or array element `f[i]`, etc., will consult the master kernel for the symbol's current downvalue
`f[i] = e`, `f[i, j] = e`, `f[i] := e`, …	defining a value for `f[i]`, etc., will perform the definition in the master kernel
`f[[i]]++`, `f[[i]]--`, `++f[[i]]`, `--f[[i]]`	performs the increment/decrement operation in the master kernel (this operation is atomic and can be used for synchronization)
`TestAndSet[f[i], e]`	if `f[i]` has no value or its value is `Null`, sets the value to *e*; otherwise, does nothing and returns the current value of `f[i]` (this operation is atomic and can be used for synchronization)

Operations on shared functions.

For technical reasons, every expression of the form `f[…]` must have a value. If the expression `f[…]` in the master kernel does not evaluate, the result is set to `Null`.

Note that other forms of assignments, such as conditional assignments involving side conditions, are not supported.

You can define shared functions, as in the following. Be sure that the symbol *x* does not have a value in either the remote kernels or in the master kernel. The symbol *x* should not be a shared variable.

```
f[x_] := x^2
```

If you make a delayed assignment on a remote kernel, the right side of the definition will be evaluated on the remote kernel when you use the function. In an immediate assignment, it is evaluated on the master kernel.

```
f[x_] = x^2
```

If you make a delayed assignment on the master kernel, the right side of the definition will be evaluated on the master kernel when you use the function. To cause the right side to be evaluated on the remote kernel nevertheless, use `SendBack[]`.

```
f[x_] := SendBack[x^2]
```

You can implement indexed variables or arrays using shared downvalues of the form `x[1]`, `x[2]`, and so forth.

If a function is `Protected` when you declare it as shared, remote kernels can only use it, but not change its definition.

Basic Example

Load PCT, then start a few local or remote kernels.

```
In[1]:= Needs["Parallel`Debug`"]
        Needs["Parallel`"]
```

Assign the initial value 17 to *x* and declare *x* as a shared variable.

```
In[2]:= x = 17;
        SharedVariables[x]
```

At least two remote kernels should be running. Assign them to two variables for easier use.

```
In[4]:= r1 = $Slaves[[1]]; r2 = $Slaves[[2]];
```

The kernel *r1* now has access to the common value of *x*.

```
In[5]:= RemoteEvaluate[x, r1]

Out[5]= 17
```

Kernel *r2* can change the value of *x* to 18.

```
In[6]:= RemoteEvaluate[x = 18, r2]

Out[6]= 18
```

The local copy of *x* on the master kernel has been changed as well.

```
In[7]:= x

Out[7]= 18
```

Kernel *r1* sees the new value, too.

```
In[8]:= RemoteEvaluate[x, r1]

Out[8]= 18
```

Synchronization

In a situation where several concurrently running remote kernels access the same shared variable for reading and writing, there is no guarantee that the value of a variable is not changed by another process between the time you read a value and write a new value. Any other new value that another process wrote in the meantime would get overwritten.

Example: Critical Sections

This classic example of uncontrolled access to a shared variable illustrates the problem. To try out this example, you should have between two and 10 remote kernels running.

The code inside the first argument of `ParallelMap` is the client code that is executed independently on the available remote kernels. The code reads the shared variable `y`, stores its value in a local variable `a`, performs some computations (here simulated with `Pause`), and then wants to increment the value of `y` by setting it to `a + 1`. But by that time, the value of `y` is most likely no longer equal to `a`, because another process will have changed it.

```
In[9]:= SharedVariables[y];

In[10]:= y = 0;
         ParallelMap[(
            Pause[0.4 Random[]];
            (* begin critical section *)
            a = y;
            Pause[Random[]];
            y = a + 1
            (* end critical section *)
           ) &,
          Range[10]]

Out[11]= {1, 2, 3, 1, 2, 2, 1, 2, 1, 2}
```

If this code were run sequentially (by changing `ParallelMap` into `Map`), the final value of `y` would be 10, but with enough parallel processes, it will most likely be lower.

```
In[12]:= y

Out[12]= 3
```

The code between reading the variable `y` and setting it to a new value is called a *critical section*. During its execution, no other process should read or write `y`. To reserve a critical section, a process can acquire an exclusive lock before entering the critical section and release the lock after leaving the critical section.

PCT provides the operation `TestAndSet[`*lck*`,`*e*`]` to acquire a lock. The argument *lck* must be a shared variable; once a process has set *lck* to a unique value, no other process should set *lck*. To release the lock, the process that acquired it simply sets *lck* to `Null`.

Here is the previous example with the additional code to implement locking. The processes use the unique integer function parameter `#` as the value of the lock. This value is guaranteed to be different for each process. To acquire the lock, a process performs `TestAndSet[`*lck*`, #]` and then checks whether the result is equal to `#`. If it is, the locking was successful. If it is not, some other process currently holds the lock. The returned value will even tell you which process holds the lock.

If locking fails, you have no choice but to try again until you eventually succeed. Note that between attempts to acquire the lock (inside `While`) the process waits for a while. Otherwise, processes waiting to acquire a lock that is reserved for another process will put a heavy load on the master kernel.

```
In[13]:= SharedVariables[{y, lck}];

In[14]:= y = 0;
         ParallelMap[(
            Pause[0.4 Random[]];
            While[TestAndSet[lck, #] =!= #, Pause[0.2]]; (* acquire lock *)
            a = y;
            Pause[Random[]];
            y = a + 1;
            lck = Null; (* release lock *)
            a + 1) &,
          Range[10]]

Out[15]= {1, 5, 6, 2, 8, 9, 7, 10, 3, 4}

In[16]:= y

Out[16]= 10
```

Locking slows down a computation because remote processes may have to wait for one another. In this example, the result is essentially a sequential execution. You should keep critical sections as short as possible. If a process sets a lock but never releases it, a *deadlock* may occur in which any other process waiting to acquire the lock will wait forever.

You can also use other atomic operations such as *lck*++ for locking purposes.

Tracing the Computation of This Example

For debugging shared variable operations, you can enable tracing provided you loaded the debug package before PCT.

```
In[1]:= Needs["Parallel`Debug`"]

In[2]:= Needs["Parallel`"]

In[3]:= SharedVariables[{y, lck}];
```

Now you can enable `SharedMemory` tracing.

```
In[4]:= SetOptions[$DebugObject, Trace → {SharedMemory}]

Out[4]= {Trace → {SharedMemory}}

In[5]:= y = 0;
        ParallelMap[(
           Pause[0.4 Random[]];
           While[TestAndSet[lck, #] =!= #, Pause[0.2]];
           a = y;
           Pause[Random[]];
           y = a + 1;
           lck = Null;
           a + 1) &,
         Range[4]]

           SharedMemory: slave6[n2]: TestAndSet[lck, 2] ⟶ 2
           SharedMemory: slave6[n2]: y ⟶ 0
           SharedMemory: slave7[n3]: TestAndSet[lck, 3] ⟶ 2
           SharedMemory: slave8[n4]: TestAndSet[lck, 4] ⟶ 2
           SharedMemory: slave7[n3]: TestAndSet[lck, 3] ⟶ 2
           SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 2
           SharedMemory: slave8[n4]: TestAndSet[lck, 4] ⟶ 2
           SharedMemory: slave7[n3]: TestAndSet[lck, 3] ⟶ 2
           SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 2
           SharedMemory: slave8[n4]: TestAndSet[lck, 4] ⟶ 2
           SharedMemory: slave7[n3]: TestAndSet[lck, 3] ⟶ 2
           SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 2
           SharedMemory: slave6[n2]: y = 1 ⟶ 1
           SharedMemory: slave6[n2]: lck = Null ⟶ Null
           SharedMemory: slave8[n4]: TestAndSet[lck, 4] ⟶ 4
           SharedMemory: slave8[n4]: y ⟶ 1
           SharedMemory: slave7[n3]: TestAndSet[lck, 3] ⟶ 4
```

```
SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 4
SharedMemory: slave7[n3]: TestAndSet[lck, 3] ⟶ 4
SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 4
SharedMemory: slave7[n3]: TestAndSet[lck, 3] ⟶ 4
SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 4
SharedMemory: slave7[n3]: TestAndSet[lck, 3] ⟶ 4
SharedMemory: slave8[n4]: y = 2 ⟶ 2
SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 4
SharedMemory: slave8[n4]: lck = Null ⟶ Null
SharedMemory: slave7[n3]: TestAndSet[lck, 3] ⟶ 3
SharedMemory: slave7[n3]: y ⟶ 2
SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 3
SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 3
SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 3
SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 3
SharedMemory: slave7[n3]: y = 3 ⟶ 3
SharedMemory: slave7[n3]: lck = Null ⟶ Null
SharedMemory: slave5[n1]: TestAndSet[lck, 1] ⟶ 1
SharedMemory: slave5[n1]: y ⟶ 3
SharedMemory: slave5[n1]: y = 4 ⟶ 4
SharedMemory: slave5[n1]: lck = Null ⟶ Null
```

Out[6]= {4, 1, 3, 2}

In[7]:=
```
SetOptions[$DebugObject, Trace → {}];
```

7 Failure Recovery, Tracing, and Debugging

Failure of Remote Kernels

A remote kernel in use may fail at any time, due to hardware, network, or software problems. A failure of a remote kernel will be noticed the next time PCT tries to send a command to the kernel or tries to read a result from it. The error message `Parallel::rdead` is used to notify you of a failed remote kernel.

If the failed kernel had any processes assigned to it, these processes will be lost. If you are using `Wait` for one of these processes, your program will never terminate because the process will never return.

Because PCT keeps track of the commands submitted to remote kernels, it can reassign these commands to another available remote kernel if a remote kernel fails. Alternatively, it may simply terminate the waiting processes with the result `$Failed`, which indicates failure. The chosen behavior is determined by the value of the variable `$RecoveryMode`.

`$RecoveryMode`	gives the current setting of the failure recovery mode
`$RecoveryMode = None`	does not perform any failure recovery
`$RecoveryMode = Abandon`	lets processes assigned to a failed kernel return with the result `$Failed` (default)
`$RecoveryMode = ReQueue`	reassigns processes on the failed kernel to another kernel

Possible failure recovery modes.

The `ReQueue` recovery mode lets you finish a computation as long as at least one kernel remains usable. However, it may give wrong results if the remote computations produce side effects or your computation depends on a certain number of available remote kernels. Side effects are usually present if you use virtual

shared memory. There is also the possibility of a deadlock if a process on a failed kernel acquired, but never released, a shared resource.

You can use the `Abandon` recovery mode to implement your own failure recovery method.

Failure recovery affects only processes started with `Queue[]` and collected with `Wait[]`. Other parallel commands, such as `ParallelEvaluate[]`, cannot handle a failed remote kernel and always return `$Failed` in such cases.

Tracing and Debugging

Debugging concurrent programs can be tricky. PCT offers a tracing facility that lets you monitor the progress of your computation. To use these features, you have to load the debugging package before loading PCT.

```
In[1]:= Needs["Parallel`Debug`"]

In[2]:= Needs["Parallel`"]
```

`SetOptions[$DebugObject,` *opts…*`]`	sets debug options of PCT
`Options[$DebugObject]`	gives the current debug option settings
`Trace→{`*tracers…*`}`	sets trace events
`TraceHandler→`*handler*	specifies how trace events should be handled; possible values include `Print` and `Save`
`TraceList[]`	gives the current list of trace events
`newTraceList[]`	initializes the trace list

Debugging functions.

`OptionValues[Trace]`	gives the list of possible tracers
`MathLink`	traces *MathLink* events
`SendReceive`	traces `Send/Receive` operations
`Queueing`	traces process scheduling (`Queue/Wait`)
`SharedMemory`	traces shared variable access (available only if the `Virtual Shared` package has been loaded)

Tracers.

Tracing Events

To see certain events, specify the desired class of events in `SetOptions[$DebugObject,Trace ->{`*tracers*`...}]`. From then on everytime one of the selected events occurs, a message is printed.

```
In[3]:= SetOptions[$DebugObject, Trace → {SendReceive}]

Out[3]= {Trace → {SendReceive}}
```

`ParallelEvalute` uses `Send` and `Receive` internally, so you can see how the computation is divided into parts that are sent to remote kernels.

```
In[4]:= ParallelEvaluate[{1, 2, 3, 4, 5}, Prime]

          SendReceive: Sending to slave9[localhost]: Prime[{1, 2, 3}] (q=1)

          SendReceive: Sending to slave10[localhost]: Prime[{4, 5}] (q=1)

          SendReceive: Receiving from slave9[localhost]: {2, 3, 5} (q=0)

          SendReceive: Receiving from slave10[localhost]: {7, 11} (q=0)

Out[4]= {2, 3, 5, 7, 11}
```

To turn off tracing, specify the empty list as tracers.

```
In[5]:= SetOptions[$DebugObject, Trace → {}]

Out[5]= {Trace → {}}
```

Saving Trace Events

Instead of printing trace events, PCT can save them in a list for later analysis. First, configure the tracing system to save the events and initialize the trace list.

```
In[6]:= SetOptions[$DebugObject, TraceHandler → Save]

Out[6]= {TraceHandler → Save}

In[7]:= newTraceList[]
```

Now specify which events to trace as before.

```
In[8]:= SetOptions[$DebugObject, Trace → {SendReceive}]

Out[8]= {Trace → {SendReceive}}
```

Run your computation.

```
In[9]:= ParallelEvaluate[{1, 2, 3, 4, 5}, Prime]

Out[9]= {2, 3, 5, 7, 11}
```

The list of events is now available in `TraceList[]`, which is best viewed in `TableForm`.

```
In[10]:= TableForm[TraceList[], TableDepth → 2,
           TableHeadings → {Automatic, {"Trigger", "Event"}}]

  Out[10]//TableForm=
                Trigger          Event
          1     SendReceive      Sending to slave9[localhost]: Prime[{1, 2, 3}] (q=1)
          2     SendReceive      Sending to slave10[localhost]: Prime[{4, 5}] (q=1)
          3     SendReceive      Receiving from slave9[localhost]: {2, 3, 5} (q=0)
          4     SendReceive      Receiving from slave10[localhost]: {7, 11} (q=0)
```

To reset the list, use `newTraceList[]` and to end tracing, turn it off as before.

```
In[11]:= SetOptions[$DebugObject, Trace → {}]

Out[11]= {Trace → {}}
```

To switch back to printing trace events, use the following.

```
In[12]:= SetOptions[$DebugObject, TraceHandler → Print]

Out[12]= {TraceHandler → Print}
```

The Format of Trace Events

MathLink

MathLink trace messages are described in the chapter Starting Remote Kernels.

SendReceive

A `SendReceive` trace message has the format

`Sending to` *slave* `:` *expr* `(q =` *n*`)`

or

`Receiving from` *slave* `:` *expr* `(q =` *n*`)`

where *slave* is the kernel involved, *expr* is the expression sent or received, and *n* is the size of the kernel's queue.

Queueing

A `Queueing` trace message has one of these formats (*pid* is a process ID, *slave* a remote kernel).

- A process is queued (with `Queue[]`). *n* is the length of the queue.

  pid `queued (`*n*`)`

- A process is sent to a remote kernel.

  pid `on` *slave*

- A process has finished and has been received from a remote kernel.

  pid `done`

- A process has been returned to the application (inside `Wait[]` or `WaitOne[]`).

  pid `dequeued`

SharedMemory

A `SharedMemory` trace message has the format

slave : *access*

where *slave* is the kernel that accessed the shared variable, and *access* describes how the variable was accessed.

- The value of the variable *var* was requested and *val* was returned.

  var → *val*

- The remote kernel asked to change the variable *var* to *val*. The new value *val* was returned.

  (*var* = *val*) → *val*

- The value of a part of the variable *var* was requested and *val* was returned.

  var[[*spec*]] → *val*

- The remote kernel asked to change a part of the variable *var* to *val*.

  (*var*[[*spec*]] = *val*) → *val*

- The remote kernel asked for exclusive access to the variable *var*, setting it to *val*. The request was granted because *var* was currently unused.

  `TestAndSet[`*var*`, `*val*`]` → *val*

- The remote kernel asked for exclusive access to the variable *var*, setting it to *val*. The request was denied because *var* already had the different value *old* set by another process.

  `TestAndSet[`*var*`, `*val*`]` → *old*

- The remote kernel released exclusive access to the variable *var*.

  (*var* = `Null`) → `Null`

For shared downvalues, the expression *var* in the preceding examples will be a normal expression whose head is the shared downvalue, such as `f[…]`.

Aborting Parallel Programs

You can interrupt and abort the local (master) kernel during a concurrent computation. Any evaluations already on remote kernels will continue to run. After an abort, wait for any processes still in the queues using `Wait`, abandon them with `ResetQueues`, or abort the remote kernels with `ResetSlaves[]`.

If you abort any other operation such as `ParallelEvaluate[]`, you should follow it by `ResetSlaves[]`.

`ResetQueues[]`	waits for any running processes to finish and clears all queues
`ResetSlaves[]`	aborts all remote kernels and makes them available again
`CloseSlaves[]`	closes the *MathLink* connections to all remote kernels

Recovering from interrupts and resetting remote kernels.

There is not always a reliable way to interrupt a remote kernel. A reliable way to interrupt a remote kernel is not always available. `ResetQueues[]` waits for any running computations to finish normally to avoid an interrupt. If this takes too long, try to abort the master kernel again and then use `ResetSlaves[]`.

`ResetSlaves[]` tries to abort any remote kernels that are not responding. Kernels that fail to react are closed.

If you quit the local kernel while a remote one is still doing a computation, the remote kernel may continue running and should be aborted or eventually killed using the appropriate operating system command.

8 Sample Parallel *Mathematica* Commands

PCT provides parallel implementations of several *Mathematica* commands. These are part of the `Parallel`Commands`` package. Load this package with `Needs["Parallel`Commands`"]`, after loading PCT.

Most parallel versions use `ParallelMap` or `ParallelTable` to distribute part of the computational work to all available remote kernels. Others use `Queue` and `Wait` to create their own concurrent processes.

The basic parallel commands `ParallelMap` and `ParallelTable` are described in the chapter Parallel Evaluation.

The implementations in this chapter are of a general nature and do not try to optimize the computation for a particular number of kernels. Depending on the relation between the amount of parallel computation and the size of the expressions that needs to be sent back and forth to the remote kernels, you may or may not observe a speed increase over the standard sequential *Mathematica* commands.

The `Parallel`Commands`` package is distributed in source form. You are welcome to look at it and gain insight for possible parallelization of your own algorithms. This package is covered under the PCT copyright.

The new `ParallelEvaluate[]` in Version 2 of PCT made many of the implementations in this package either trivial or obsolete. The package is provided mainly for backward compatibility.

Parallel Animation and Plotting

`ParallelAnimate[`*command*`, `*iterator*`, `*options*`]`
parallel version of `Graphics`Animation`˙˙.Animate[`*command*`, `*iterator*`, `*options*`]`

`ParallelPlot3D[`*expr*`, {`*x*`, `*xmin*`, `*xmax*`}, {`*y*`, `*min*`, `*ymax*`}, `*options*`]`
parallel version of `Plot3D[`*expr*`, {`*x*`, `*xmin*`, `*xmax*`}, {`*y*`, `*min*`, `*ymax*`}, `*options*`]`

`ParallelDensityPlot[`*expr*`, {`*x*`, `*xmin*`, `*xmax*`}, {`*y*`, `*min*`, `*ymax*`}, `*options*`]`
parallel version of `DensityPlot[`*expr*`, {`*x*`, `*xmin*`, `*xmax*`}, {`*y*`, `*min*`, `*ymax*`}, `*options*`]`

`ParallelContourPlot[`*expr*`, {`*x*`, `*xmin*`, `*xmax*`}, {`*y*`, `*min*`, `*ymax*`}, `*options*`]`
parallel version of `ContourPlot[`*expr*`, {`*x*`, `*xmin*`, `*xmax*`}, {`*y*`, `*min*`, `*ymax*`}, `*options*`]`

`ParallelParametricPlot3D[{`*x*`, `*y*`, `*z*`}, {`*u*`, `*u0*`, `*u1*`, `(*du*)`}, {`*v*`, `*v0*`, `*v1*`, `(*dv*)`}, `*options...*`]`
parallel version of `ParametricPlot3D[{`*x*`, `*y*`, `*z*`}, {`*u*`, `*u*0`, `*u*1`, `(*du*)`}, {`*v*`, `*v*0`, `*v*1`, `(*dv*)`}, `*options*`]`

Parallel animation and plotting.

Note that even though a graphic is computed in parallel, it is still rendered sequentially in the master kernel and front end.

Parallel Inner Products

`ParallelInner[`*f*`, `*list1*`, `*list2*`, `*g*`]`	parallel version of `Inner[`*f*`, `*list1*`, `*list2*`, `*g*`]`
`ParallelDot[`*m1*`, `*m2*`]`	parallel version of `Dot[`*m1*`, `*m2*`]` or *m1, m2*; equivalent to `ParallelInner[`*Times*`, `*m1*`, `*m2*`, `*Plus*`]`

Parallel inner products.

Parallelization happens along all dimensions of the first matrix or tensor. All computations involving *g* or `Plus` with all elements of the second matrix or tensor are performed on the same remote processor.

Parallel computations cannot be nested.

Examples

Load PCT and the `Parallel`Commands`` packages and start a few remote kernels.

```
In[1]:= Needs["Parallel`"]
        Needs["Parallel`Commands`"]

           Parallel Computing Toolkit 2.0 (November 11, 2004)

           Created by Roman E. Maeder
```

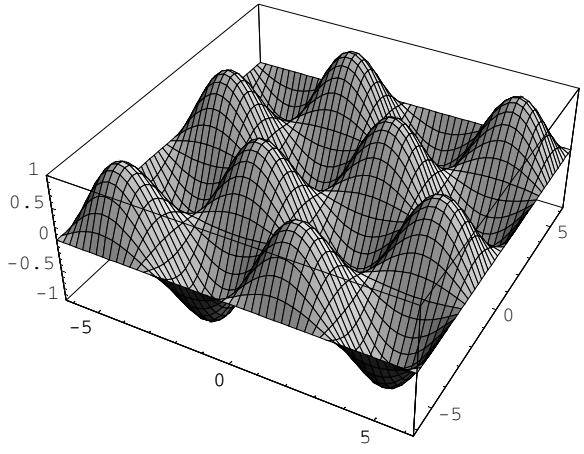
An Animation

Load the `Graphics`Animation`` package.

```
In[3]:= Needs["Graphics`Animation`"]
```

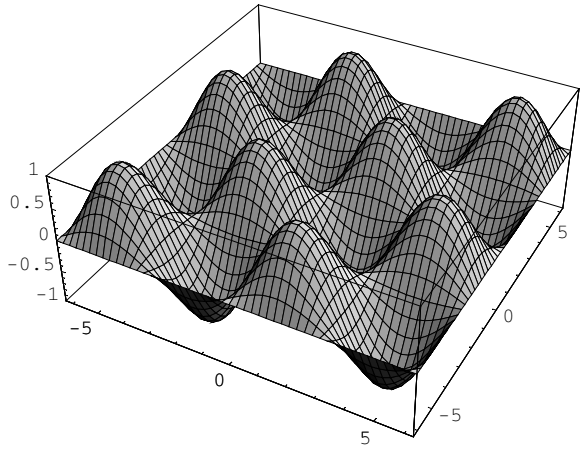
Try out the animation sequentially with a small number of frames.

```
In[4]:= Animate[Plot3D[Sin[x + t] Sin[y + t / 2], {x, -2 π, 2 π}, {y, -2 π, 2 π},
            PlotPoints -> 50, PlotRange -> {-1, 1}], {t, 0, 4 π}, Frames -> 4, Closed -> True];
```



Now compute it in parallel with a larger number of frames.

```
In[5]:= ParallelAnimate[Plot3D[Sin[x + t] Sin[y + t / 2], {x, -2 π, 2 π}, {y, -2 π, 2 π},
            PlotPoints -> 50, PlotRange -> {-1, 1}], {t, 0, 4 π}, Frames -> 24, Closed -> True];
```



Note that the rendering of the frames has to happen sequentially on the master kernel and the front end; only their computation can be parallelized.

Index