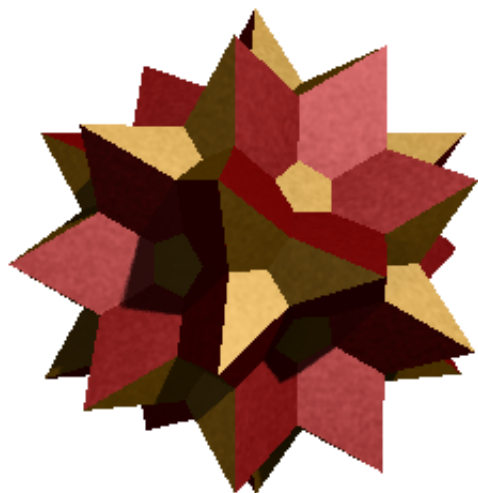


# Parallel Computation with Mathematica

The Parallel Computing Toolkit and gridMathematica



*Roman E. Mäder*  
*MathConsult Dr. R. Mäder*  
<http://www.mathconsult.ch>

on behalf of

*unisoftware plus gmbh*  
<http://www.unisoftwareplus.com>

The **Parallel Computing Toolkit** is a Mathematica AddOn that allows users to develop and run parallel computations. It provides a high-level language for expressing parallel algorithms in a machine-independent way. gridMathematica takes full advantage of the Parallel Computing Toolkit for providing an easy-to-use environment for parallel computations on multi-processor machines or clusters of compute nodes. The presentation discusses the features of the Parallel Computing Toolkit and gives examples of applications.

## **Dr. Roman Mäder**

The speaker is one of the developers of *Mathematica* and author of the Parallel Computing Toolkit, which is the basis for gridMathematica.

Dr. Mäder is the author of several books on *Mathematica*, especially **Programming in Mathematica**.

He is now a consultant on computer-aided mathematics, particularly for finance applications.

© 2004 MathConsult Dr. R. Mäder. All rights reserved.

---

## **Introduction**

Topics include

- Infrastructure for Parallel Computation with Mathematica
- the Parallel Computing Toolkit
- Parallel Programming Methods
- Shared Memory and Synchronization
- gridMathematica
- Examples

---

## Support for Parallel Computing

What is needed to add parallel computing to a programming environment, such as *Mathematica*?

### Parallel computing ( a system programmer's view)

Parallel programming requires support for:

- starting processes, waiting for processes to finish.
- scheduling processes on available processors.
- exchanging data between processes, synchronizing access to common resources

#### ■ In this context

- a *processor* is a running *Mathematica* kernel.

Parallel algorithms are expressed directly in *Mathematica*'s programming language, no compilation is necessary.

- a *process* is an expression to be evaluated.

Such an expression may be a simple calculation or a larger program.

---

## The Parallel Computing Toolkit

The Parallel Computing Toolkit is a *Mathematica* application package that provides everything that is needed for developing and running parallel computations.

### Features of the Parallel Computing Toolkit

- distributed memory, master/slave parallelism
- machine independent, all written in *Mathematica*.  
Tested on Unix/Linux/Windows/Macintosh
- uses *MathLink* protocol to communicate with remote kernels
- data not restricted to numbers and arrays. Can use symbolic expressions and programs.
- works on heterogeneous networks, multi-processor machines, LAN and WAN
- scheduling of virtual processes, or explicit distribution to available processors
- virtual shared memory, synchronization, locking
- failure recovery, stranded processes are automatically reassigned
- latency hiding

### The Application Package

All required support code is contained in an application package, which can simply be imported into a running kernel, the *master kernel*, which controls the parallel computation.

## ■ Needs["Parallel`"]

Parallel Computing Toolkit 2.0beta1 (September 2004)

Created by Roman E. Maeder

Note that the master kernel is normally connected to a *Mathematica* frontend, through which the user interacts with the parallel computation.

The master kernel interacts with any number of remote *slave kernels*, which perform part of a parallel computation. Starting, initialization, monitoring, and stopping slave kernels is done through the master kernel.

## Requirements

All scheduling happens in the master kernel. No extra software is required on the slave kernels. No support system for parallel computing, such as PVM or MPI is required.

Slave kernels are normally started once and are then used for any number of computations. The Parallel Computing Toolkit contains its own scheduler and load balancing tools.

## ■ Supported Configurations

Master and slave kernels can run on a multi-processor machine, a tightly coupled network of processors (a cluster), a network of workstations or any remote machines accessible through TCP/IP.



In this demonstration, the frontend runs on my laptop computer, the master kernel runs on a Linux machine at Wolfram Research in Illinois, USA, and the slave kernels run on a cluster of rack-mounted servers, connected through a Gigabit ethernet with the master kernel.

## Starting Remote Kernels

Any connection method supported by *MathLink* can be used to start remote kernels.

- launch a kernel on the local (multi-processor) machine

```
■ LaunchSlave["localhost"]
```

```
slave1[localhost]
```

- use remote access (rsh or ssh) to start a kernel on a remote machine

```
■ LaunchSlave["xserve1",  
"ssh -n `1` '/usr/local/bin/math5 -mathlink -linkmode  
Connect -linkname `2` -noinit >&/dev/null &'"]
```

```
slave2[xserve1]
```

- use TCP tunneling (through ssh or VPN) for secure connection to remote machines

My connection between frontend and master kernel uses this method.

- connect to a listening kernel on another machine

## Simple Calculations

`RemoteEvaluate[ ]` sends the same expression to all slave kernels. Here, we ask them for their unique processor ID.

```
| RemoteEvaluate[$ProcessorID]
```

```
{1, 2}
```

This command displays a number of properties of the remote kernels, such as the machine on which they are running, the *Mathematica* version, and CPU and memory usage so far.

```
| TableForm[RemoteEvaluate[  
  {$ProcessorID, $MachineName, $Version, TimeUsed[], MaxMemoryUsed[]},  
  TableHeadings -> {None, {"ID", "host", "Mathematica Version",  
    "CPU Time", "Memory"}}, TableDepth -> 2]
```

ID	host	Mathematica Version	CPU Time
1	gridhead	5.0 for Linux (November 18, 2003)	0.06299
2	xserve1	5.0 for Mac OS X (November 19, 2003)	0.13

After all computations are done, the slave kernels should be shut down.

```
| CloseSlaves[];
```

## Setup

The normally available slave kernels can be set up as defaults.

```
| $AvailableMachines
```

```
{RemoteMachine[xserve1, LinkHost -> 192.168.1.1],  
RemoteMachine[xserve2, LinkHost -> 192.168.1.1],  
RemoteMachine[xserve3, LinkHost -> 192.168.1.1],  
RemoteMachine[xserve4, LinkHost -> 192.168.1.1],  
RemoteMachine[xserve5, LinkHost -> 192.168.1.1],  
RemoteMachine[xserve6, LinkHost -> 192.168.1.1],  
RemoteMachine[xserve7, LinkHost -> 192.168.1.1],  
RemoteMachine[xserve8, LinkHost -> 192.168.1.1]}
```

Here is the command template needed for connecting to these kernels.

```
| $RemoteCommand
```

```
ssh -n `1` '/usr/local/bin/math5 -mathlink -  
linkmode Connect -linkname `2` -noinit >&/dev/null &'
```

It is now easy to start them up.

## ■ LaunchSlaves[]

```
{slave1 [xserve1], slave2 [xserve2], slave3 [xserve3], slave4 [xserve4],  
slave5 [xserve5], slave6 [xserve6], slave7 [xserve7], slave8 [xserve8]}
```

Here is the list of currently running slave kernels. They run on a cluster of 8 servers at Wolfram Research in Illinois, USA (each one has two CPUs).

```
TableForm[RemoteEvaluate[  
  {$ProcessorID, $MachineName, $Version, TimeUsed[], MaxMemoryUsed[]},  
  TableHeadings -> {None, {"ID", "host", "Mathematica Version",  
    "CPU Time", "Memory"}}, TableDepth -> 2]
```

ID	host	Mathematica Version	CPU Time
1	xserve1	5.0 for Mac OS X (November 19, 2003)	0.12
2	xserve2	5.0 for Mac OS X (November 19, 2003)	0.13
3	xserve3	5.0 for Mac OS X (November 19, 2003)	0.12
4	xserve4	5.0 for Mac OS X (November 19, 2003)	0.09
5	xserve5	5.0 for Mac OS X (November 19, 2003)	0.22
6	xserve6	5.0 for Mac OS X (November 19, 2003)	0.17
7	xserve7	5.0 for Mac OS X (November 19, 2003)	0.17
8	xserve8	5.0 for Mac OS X (November 19, 2003)	0.13

---

## ■ Concurrency

The building blocks of concurrency:

- queue processes
- wait for (some of) them to finish
- get results

Note: a process is simply an expression to be evaluated.

### Starting Processes, Waiting for Results

Enter the given expression into queue, return a process id.

```
■ Queue[cmd]
```

Waits for one of the given processes to finish.

```
■ WaitOne[pid_list]
```

Wait for all processes or a single process, return list of results or single result, respectively.

```
■ Wait[pid_list], Wait[pid]
```

### Examples

Queue a process and wait for it to finish.

```
| j1 = Queue[Unevaluated[1 + 1]]
```

```
| pid1
```

```
| Wait[j1]
```

```
| 2
```

Prepare to queue several processes. The construct with `Hold[]` prevents evaluation of the expressions already on the master kernel.

```
| Queue /@ Hold[1 + 1, 2 + 2, 3 + 3, 4 + 4, 5 + 5]
```

```
| Hold[Queue[1 + 1], Queue[2 + 2], Queue[3 + 3], Queue[4 + 4], Queue[5 + 5]]
```

Queue them. The result is a list of process IDs.

```
| pids = List @@ %
```

```
| {pid2, pid3, pid4, pid5, pid6}
```

Nondeterministic waiting for a job and a convenient use of the return value to drain the queue. This command can be repeated as long as the list of remaining pids is nonempty. Each time, one result is returned.

```
| {res, pid, pids} = WaitOne[pids]
```

```
| {6, pid4, {pid5, pid6}}
```

Finally, wait for all remaining processes.

```
| Wait[pids]
```

```
| {8, 10}
```

## Scheduling

Users can implement their own scheduling policy. The PCT contains sample implementation of FIFO and priority queues.

## Automatic Parallel Code Generation

`Wait[Queue[expr]]` is just the evaluation function (or identity).

```
| Wait[Queue[1 + 2]]
```

```
| 3
```

This observation gives us a powerful recipe for parallelizing functional operations. This example develops a parallel version of the *Mathematica* function `Map[]`.

```
Map[f, {a, b, c}]
```

```
{f[a], f[b], f[c]}
```

Here we use a symbolic function `q` to explain the concept of functional composition.

```
Composition[q, f][x]
```

```
q[f[x]]
```

```
Map[Composition[q, f], {a, b, c}]
```

```
{q[f[a]], q[f[b]], q[f[c]]}
```

Now we compose `f` with `Queue[ ]`. The result is a list of process IDs that represent the individual function applications.

```
Map[Composition[Queue, f], {a, b, c}]
```

```
{pid8, pid9, pid10}
```

`Wait[ ]` runs the scheduler and returns with the results, which were computed in parallel on any available slave kernels.

```
Wait[%]
```

```
{f[a], f[b], f[c]}
```

The powerful functional language allows us to derive parallel code automatically.

---

## Parallel Commands

A general mechanism, `ParallelEvaluate[ ]`, allows the distribution of a computation to available processors. It dispatches parts of the computation to remote processors and is flexible to handle a large number of different cases.

### ? ParallelEvaluate

```
ParallelEvaluate[h[exprs...], f, comb] distributes parts of the computation of f[h[exprs]] to all slaves and combines the partial results with comb. The default combinator is h if h is Flat, and Join otherwise. The default function f is the Identity.
```

Here are examples of different kinds of parallel computations.

- parallel evaluation of list elements

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

```
{3, 5, 7, 9, 11}
```

- function application: testing numbers for primality

```
| ParallelEvaluate[{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}, PrimeQ]
```

```
| {False, True, True, False, True, False, True, False, False, False, True}
```

- reduction: summation of a long sum in parallel

```
| ParallelEvaluate[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10]
```

```
| 55
```

- parallel pattern matching: finding all odd elements of a list

```
| ParallelEvaluate[Cases[{1, 2, 3, 4, 5, 6, 7, 8}, _?OddQ]]
```

```
| {1, 3, 5, 7}
```

- parallel inner products

```
| ParallelEvaluate[{a, b, c} . {x, y, z}]
```

```
| a x + b y + c z
```

---

## Shared Memory

How can processes communicate with each other?

- Message Passing
- Shared Memory

*Mathematica* kernels do not share memory, even if they run on the same machine. Our computation model is always a distributed memory one.

**Virtual Shared Memory:** implementing shared memory on distributed memory machines with a software layer that handles all variable accesses, using message passing.

When a process needs the value of a global (shared) variable or wants to set its value, it sends a request back to the scheduler. The scheduler maintains the variables, updates their values, and sends back the requested values.

### Examples

We will work with two slave kernels.

```
| s1 = $Slaves[[1]]; s2 = $Slaves[[2]];
```

Declare shared variables.

```
| SharedVariables[{x}]
```

Basic tests. One slave kernel reads the variable, another one increments its value.

```
| x = 0;
```

```
| RemoteEvaluate[x = x + 1, s1]
```

```
| 1
```

```
| RemoteEvaluate[x, s2]
```

```
| 1
```

The new value is stored globally.

```
| x
```

```
| 1
```

Increment the value in one indivisible step. This operation is important for synchronization.

```
| RemoteEvaluate[x++, s2]
```

```
| 1
```

```
| RemoteEvaluate[x, s1]
```

```
| 2
```

No special syntax is required to use shared variables. Just do things the normal way. If a variable is shared, all accesses are transparently sent to the master process.

```
| ClearShared[x];
```

## Synchronization

Exclusive access to a variable or other resource can be controlled by a *lock*. An update of a shared variable implements a lock and allows semaphores and critical sections.

Here is the typical code that implements a *critical section* with a lock. Each of the five parallel processes accesses the same shared variable *y*, first reading it, then writing a new value. Before doing this, it acquires a lock, which is released after the critical section. In this way no conflicting updates happen, as can be seen by the output, in which all values from 1 to 5 occur.

```
| SharedVariables[{y, lock}]
```

```
| y = 0;  
| ParallelMap[Module[{a},  
|     Pause[0.2 Random[]];  
|     While[TestAndSet[lock, #] != #, Pause[0.1 Random[]]];  
|     a = y;  
|     Pause[Random[]];  
|     y = a + 1;  
|     lock = Null;  
|     a + 1] &,  
|     Range[5]]
```

```
| {3, 4, 1, 2, 5}
```

```
ClearSlaves[];
```

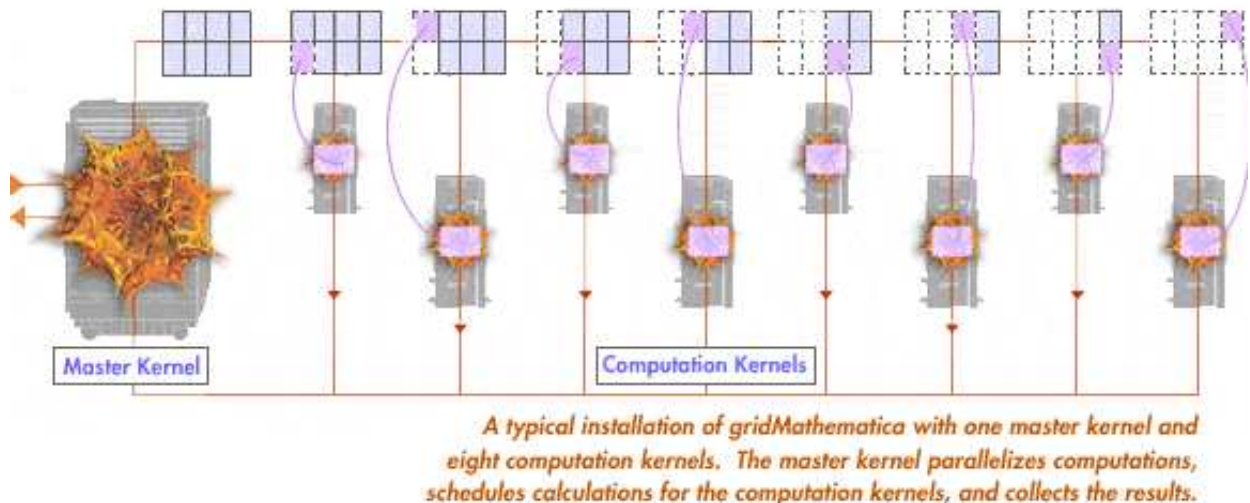
---

## gridMathematica

gridMathematica provides a quick way to set up and run large calculations by bundling *Mathematica* kernel licenses and the Parallel Computing Toolkit in a way that is easy to install on parallel computers.

A typical installation of gridMathematica involves a *master kernel*, a license manager, and one *slave kernel* per available node. The master kernel handles all input, output, and scheduling. It can be controlled from any *Mathematica* front end or via batch files, either locally or via a remote connection. Once the remote kernels are launched, they are ready to receive commands from the master machine.

gridMathematica is platform independent and can be used on dedicated multiprocessor machines as well as on homogeneous and heterogeneous clusters. The only technical requirement, apart from the ability to run *Mathematica*, is a TCP/IP connection between the individual computing nodes. This connection allows customers to run the same code on any available machines without any porting work. It also makes it easy to build ad-hoc clusters out of underutilized computers or to take advantage of low-use periods.



## Comparison with other Environments

gridMathematica does not need any load-balancing or resource allocation software or parallel libraries, such as PVM or MPI.

All parallel code is written in *Mathematica*. No compilation is necessary and no common file system is required. Parallel programs can be developed and debugged interactively, and run interactively or in batch mode.

---

## Example 1: Parallel Search

We want to find an irreducible polynomial modulo a prime  $p$ . Such polynomials are important in signal processing, data storage, and communication.

This generates a random polynomial of a given degree mod  $p$ .

```
randPoly[n_Integer?Positive, x_, p_] :=  
  Random[Integer, {1, p - 1}] * x^n + Sum[  
    Random[Integer, {0, p - 1}] * x^i, {i, 1, n - 1}] + Random[Integer, {1, p - 1}]
```

It is irreducible if it does not have any factors other than a numerical content.

```

| irreducibleQ[poly_, p_] :=
  With[{fl = FactorList[poly, Modulus -> p]}, Length[fl] < 3 && fl[[-1, 2]] == 1]

```

Here is an example polynomial modulo 2.

```

| poly = randPoly[10, x, 2]

```

```

| 1 + x2 + x3 + x5 + x6 + x8 + x9 + x10

```

We test whether it is irreducible.

```

| irreducibleQ[poly, 2]

```

```

| False

```

This loop generates new polynomials until it finds an irreducible one.

```

| findOne[n_, p_] := Module[{poly},
  While[! irreducibleQ[poly = randPoly[n, x, p], p], Null];
  poly]

```

Here is an irreducible polynomial of degree 100 mod 2.

```

| findOne[100, 2]

```

```

| 1 + x2 + x3 + x5 + x6 + x13 + x14 + x19 + x20 + x21 + x22 + x23 + x24 + x30 + x35 + x39 + x40 +
  x41 + x42 + x43 + x44 + x45 + x47 + x51 + x53 + x56 + x58 + x60 + x61 + x63 + x65 + x68 + x70 +
  x72 + x73 + x76 + x77 + x78 + x79 + x80 + x81 + x87 + x88 + x89 + x90 + x94 + x96 + x97 + x100

```

We run such a loop on each slave kernel and wait until one of them returns a result. The remaining computations are then aborted gracefully.

Let us line up more workers; each of the 8 machines has two CPUs so we can launch another kernel on each one.

```

| LaunchSlaves[]

```

```

| {slave9[xserve1], slave10[xserve2], slave11[xserve3], slave12[xserve4],
  slave13[xserve5], slave14[xserve6], slave15[xserve7], slave16[xserve8]}

```

Here is the number of available remote kernels.

```

| Length[$Slaves]

```

```

| 16

```

```

| ExportEnvironment[randPoly, irreducibleQ, findOne]

```

```

| With[{n = 512, p = 2},
  Module[{id, ids},
    ids = Table[Queue[findOne[n, p]], {Length[$Slaves]}];
    res = $Failed;
    While[res == $Failed && Length[ids] > 0, (* try one more *)
      {res, id, ids} = WaitOne[ids];
    ];
    ResetSlaves[];
  ]
]

```

Here is the result.

```
| res
```

---

## Example 2: Monte–Carlo Simulation

### Components

Monte–Carlo simulation is used to approximate values for which no closed–form solution exists. It works by generating a large number of samples (or scenarios) and averaging the results.

- A generator for random scenarios
- A valuation function for a scenario
- An averaging method over all scenarios

In this example we want to find the theoretical value of a path–dependent financial derivative instrument. A scenario is a possible stock–price movement over a number of days.

### Stock Price Movement

A common assumption is that the returns of a stock are log–normally distributed. A scenario consists of the daily stock prices under this assumption.

```
| Needs["Statistics`ContinuousDistributions`"]
```

This function generates a list of  $n$  log–normally distributed values with given mean and standard deviation. These are the simulated returns of the stock.

```
| RandomLogNormal[μ_, σ_, n_] := RandomArray[LogNormalDistribution[μ, σ], n]
```

```
| RandomLogNormal[0.01, 0.1, 10]
```

```
{1.06082, 0.99825, 0.830601, 1.15185,  
 0.869831, 0.928707, 1.03128, 1.02607, 1.01515, 1.03158}
```

The stock prices themselves are then obtained by successive multiplication.

```
| FoldList[Times, 1, %]
```

```
{1, 1.06082, 1.05897, 0.879579, 1.01315, 0.881267,  
 0.818439, 0.844039, 0.866046, 0.879169, 0.906933}
```

This function generates such a sequence of stock prices.

```
| Scenario[μ_, σ_, n_] :=  
  Rest[FoldList[Times, 1, RandomLogNormal[ $\frac{\mu}{n}$ ,  $\frac{\sigma}{\sqrt{n}}$ , n]]]
```

```
| sample = Scenario[0.01, 0.1, 10]
```

```
{0.99736, 1.00879, 0.965867, 0.973915,  
 0.936141, 0.997953, 1.04455, 1.06715, 1.06947, 1.05014}
```

## Valuation

The payoff of an arithmetic–mean call option depends on the mean of the stock price over the lifetime of the option. Here,  $x$  represents the strike price.

```
| avgPayoff[prices_, x_] := Max[Mean[prices] - x, 0]
```

```
| avgPayoff[sample, 1.0]
```

```
| 0.0111333
```

## Averaging

To find the value of the derivative instrument, we perform many simulations and take the average of the results.

```
| MonteCarloValuation[payoff_,  $\sigma$ _, r_, n_, trials_] :=  
| Sum[payoff[Scenario[r -  $\sigma^2$  / 2,  $\sigma$ , n]], {trials}] / trials
```

```
| MonteCarloValuation[avgPayoff[#, 1.0] &, 0.2, Log[1.1], 360, 1000]
```

```
| 0.0784224
```

Each time we evaluate the valuation function we get a different result.

```
| MonteCarloValuation[avgPayoff[#, 1.0] &, 0.2, Log[1.1], 360, 1000]
```

```
| 0.0827051
```

## Parallelization

Averaging is a linear operation, so we can simply perform parts of it in parallel and combine the results.

We can export all the code and then use `MonteCarloValuation` on the slave kernels.

```
| RemoteEvaluate[Needs["Statistics`ContinuousDistributions`"]];
```

```
| ExportEnvironment [  
| {MonteCarloValuation, avgPayoff, Scenario, RandomLogNormal}];
```

## Running the Simulation

This is the number of kernels available.

```
| Length[$Slaves]
```

```
| 16
```

To perform the same calculation on all slave kernels, we can use `RemoteEvaluate[]`.

```
res = RemoteEvaluate[
  MonteCarloValuation[avgPayoff[#, 1.0] &, 0.2, Log[1.1], 360, 2000]]
```

```
{0.0747837, 0.0774568, 0.0756655, 0.0735534, 0.0725372,
 0.0785634, 0.0746813, 0.0759387, 0.0734202, 0.0767386,
 0.0757416, 0.0761098, 0.0731437, 0.07783, 0.0797182, 0.0773893}
```

Because all slave kernels used the same number of simulations, we can simply average the results.

```
Mean[res]
```

```
0.0758295
```

---

## Example 3: Global Interval Minimization

### Introduction

Minimization finds the minimum value (and where it occurs) of a function of several variables. It is used in optimization problems.

#### Local minima

- make certain assumptions about differentiability of the objective function
- return a local minimum
- efficient

Examples: `FindMinimum[ ]`

#### Global techniques

- guarantee to find the global minimum, depending on certain continuity conditions
- evaluate the objective function in many places and use adaptive methods for subdivision of interesting regions
- compute-intensive

Examples:

`NMinimize[ ]`

*Global Optimization* by Loehle Enterprises

*LGO* by János Pintér

#### Interval methods

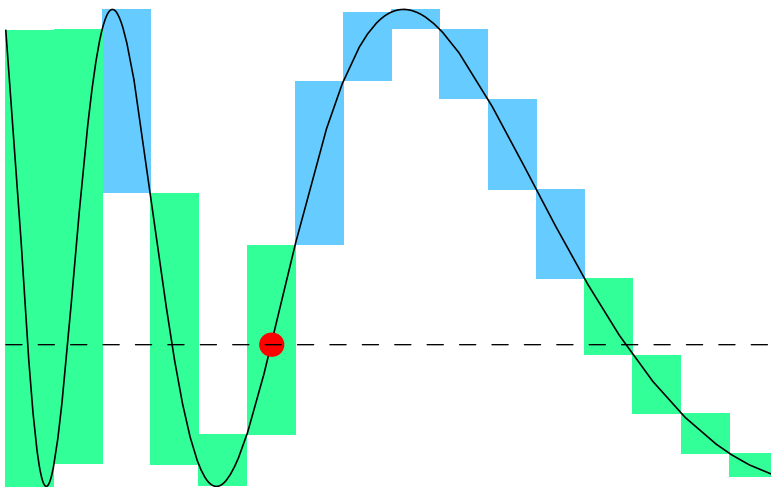
- make no assumptions about the objective function (other than that it can be evaluated for intervals).
- return a set of rectangles guaranteed to contain the position(s) of the global minimum or minima, and a range interval guaranteed to include the value of the global minimum.
- even more compute-intensive
- requires interval arithmetic (not supported in many systems besides Mathematica)

## Algorithm

The domain rectangle is subdivided into  $2^d$  intervals ( $d$  is the dimension of the problem) until either

- the size of the rectangle falls below some threshold (essentially giving up searching further).
- the size of the range interval falls below a threshold (having zoomed in on the possible values of the objective function over this rectangle, when the function is *flat*).

Exclude a rectangle from further consideration if it can be established that it cannot contain a global minimum. This is the case if the value of the objective function over the rectangle is larger than a known function value at a point lying in some other interval (a *feasible point*, shown in red). These rectangles are colored blue in this illustration.



## Parallelization

The list of remaining rectangles and the best minimum so far are global values that need to be shared among all processors working on parts of the problem. The communication and synchronization overhead can be decreased by keeping local queues and synchronizing only occasionally.

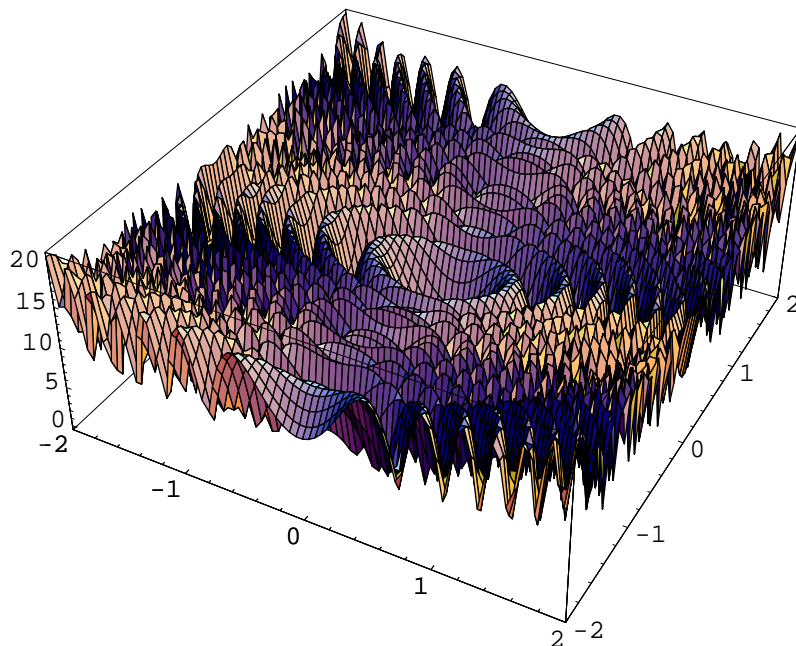
The resulting code is quite sophisticated and is several pages long. Here is an example of its performance.

## An example proposed by János Pintér

Here is the function of two variables whose global minimum we want to find.

```
f = Function[{x, y},  
  x^2 + y^2 + Log[1 + 20 Abs[Sin[x^2 + y^2]]] + 10 Abs[Sin[5 x^2 + 15 y^2]]];
```

```
Plot3D[f[x, y], {x, -2, 2}, {y, -2, 2}, PlotPoints -> 80];
```



This is the domain in which we search for the minimum.

```
domain = N[Interval /@ {{-4, 10}, {-4, 10}}]
```

```
{Interval[{-4., 10.}], Interval[{-4., 10.}]}
```

Here is the precision with which we want to find the minimum.

```
tol = 1.0*^-6;
```

This command initializes the global shared variables and starts the parallel computation. When it finishes it reports performance statistics.

```
pq = priorityQueue[];
enQ[#, f@@#] & /@ initials;
minmax = Infinity; nbusy = Length[$Slaves];
With[{f = f, tol = tol, ming = 2 * Length[$Slaves] + 1},
  fin = RemoteEvaluate[driver[f, tol, ming]]
];
```

```
7 finished, 121 intervals processed, 3 in output, 0 times idle.
8 finished, 86 intervals processed, 11 in output, 1 times idle.
1 finished, 94 intervals processed, 0 in output, 1 times idle.
6 finished, 122 intervals processed, 4 in output, 1 times idle.
2 finished, 112 intervals processed, 1 in output, 1 times idle.
5 finished, 142 intervals processed, 4 in output, 1 times idle.
4 finished, 111 intervals processed, 2 in output, 1 times idle.
3 finished, 108 intervals processed, 0 in output, 1 times idle.
```

Here is the value of the minimum.

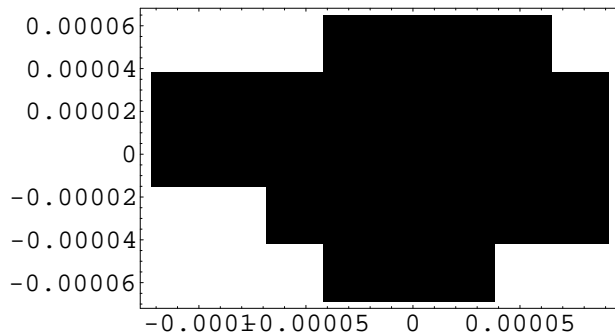
```
minmax
```

```
3.52156 × 10-7
```

This is the collection of intervals in which the minimum is guaranteed to lie.

```
result
```

```
Show[rg, AspectRatio -> Automatic, Frame -> True];
```



---

## Conclusions

```
CloseSlaves[]
```

*Mathematica* and the *Parallel Computing Toolkit* offer a unique environment for the development, testing, and use of parallel programs. It brings parallel computation within reach of anybody with access to more than one processor.

I am glad to see that CERN has obtained a larger number of *Mathematica* licenses and I encourage everyone with access to *Mathematica* to explore the possibilities.

Please see me at unisoftware's booth for more examples and live demonstrations.