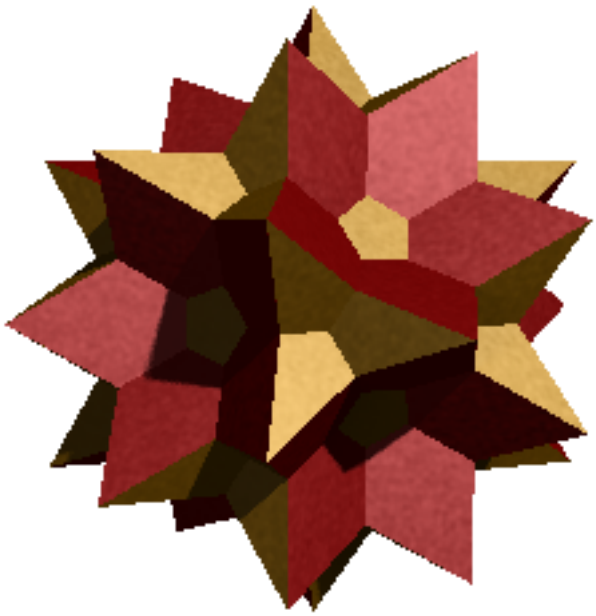
Parallel Computing Toolkit, Version 2



Dr. Roman E. Mäder
MathConsult Dr. R. Mäder
Samstagernstraße 58a
CH−8832 Wollerau

mailto:maeder@mathconsult.ch
http://www.mathconsult.ch

- Parallel computation with *Mathematica*
- Experiences with Version 1
- A powerful primitive for parallelization
- Nondeterminism
- Various new features

Introduction

What is Parallel Computing Toolkit?

Parallel Computing Toolkit (PCT) is a *Mathematica* AddOn that allows you to develop and run parallel computations.

- distributed memory, master/slave parallelism
- machine independent, all written in *Mathematica*
- uses *MathLink* to communicate with remote kernels
- 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
- supports all *Mathematica* data types, including symbolic expressions

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.

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.

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 the main node of an Orion Multisystems DT−12, and the slave kernels on the remaining 11 nodes of this desktop cluster.

PCT 2

Major new features of Version 2:

- `ParallelEvaluate[]` has been extended to become a flexible tool for implementing many structural operations in parallel (such as Map, Apply, Cases, Select, Count, MemberQ, FreeQ, Inner, Outer, and all associative functions).
- `ParallelEvaluate[]`, `ParallelMap[]`, and `ParallelTable[]` now do a single dispatch on each remote kernel, taking relative processor speeds into account for optimal load balancing.
- Delayed definitions for shared variables are now 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.
- If no remote kernels are available, all evaluations happen sequentially in the master kernel.

Enhancements:

- Improved defaults for launching local and remote kernels with a new `Parallel‘Configuration‘` package.
- `Receive[]` can take a *list* of kernels as argument, and waits for one result from each of them, allowing for callbacks, such as shared variables.
- Aborting remote kernels is now possible, provided the MathLink device used for the kernel connection supports aborts. Resetting runaway kernels is now 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.

System Configuration

Local configuration, such as a list of commonly available machines and the operating−system commands needed to launch a kernel there, can be set up in a user’s init.m file.

The template `$RemoteCommand` may contain four slots filled as follows

‘1‘	hostname	first argument of `LaunchSlave[]`
‘2‘	linkname	as returned by `LinkCreate[]`
‘3‘	remote user	`$RemoteUserName`, default is `$UserName`
‘4‘	protocol	link protocol argument

```
(orion) In[1]:= $RemoteCommand

(orion) Out[1]= ssh -f ‘1‘ ’/usr/local/bin/math -mathlink -
                  linkmode Connect -linkname ‘2‘ -noinit >&/dev/null’
```

This is the default ist of available remote machines.

```
(orion) In[2]:= $AvailableMachines

(orion) Out[2]= {n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11}
```

Startup

```
(orion) In[3]:= Needs["Parallel‘Debug‘"]
                Needs["Parallel‘"]

        Parallel Computing Toolkit 2.0beta3 (October 2004)

        Created by Roman E. Maeder



(orion) In[6]:= LaunchSlaves[]

(orion) Out[6]= {slave1[n1], slave2[n2], slave3[n3], slave4[n4], slave5[n5], slave6[n6],
                 slave7[n7], slave8[n8], slave9[n9], slave10[n10], slave11[n11]}
```

```
(orion) In[7]:= TableForm[RemoteEvaluate[{$ProcessorID, $MachineName, $SystemID,
                  $Version, TimeUsed[], MaxMemoryUsed[]}], TableHeadings ->
                 {None, {"ID", "host", "OS", "Mathematica Version", "CPU Time", "Memory"}}]
```

```
(orion) Out[7]//TableForm=
ID      host      OS        Mathematica Version                    CPU Time
1       n1        Linux     5.0 for Linux (November 18, 2003)      0.064991
2       n2        Linux     5.0 for Linux (November 18, 2003)      0.06499
3       n3        Linux     5.0 for Linux (November 18, 2003)      0.06599
4       n4        Linux     5.0 for Linux (November 18, 2003)      0.062989
5       n5        Linux     5.0 for Linux (November 18, 2003)      0.06599
6       n6        Linux     5.0 for Linux (November 18, 2003)      0.06899
7       n7        Linux     5.0 for Linux (November 18, 2003)      0.06799
8       n8        Linux     5.0 for Linux (November 18, 2003)      0.064991
9       n9        Linux     5.0 for Linux (November 18, 2003)      0.06199
10      n10       Linux     5.0 for Linux (November 18, 2003)      0.068989
11      n11       Linux     5.0 for Linux (November 18, 2003)      0.068989
```

Parallelizing Commands

Wait and Queue

`Queue[expr]` queues the remote evaluation of expr and returns a process ID (pid).

`Wait[pid]` waits for the given process to finish and returns its result.

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

```
(orion) In[8]:= Wait[Queue[1 + 2]]
```

```
(orion) Out[8]= 3
```

`Wait[{pid1,pid2,...}]` waits for several processes.

```
(orion) In[9]:= Wait[Queue /@ {1 + 1, 2 + 2, 3 + 3}]
```

```
(orion) Out[9]= {2, 4, 6}
```

This example was not very parallel. We need to be careful.

```
(orion) In[10]:= Wait[Queue /@ Unevaluated[{1 + 1, 2 + 2, 3 + 3}]]
```

```
(orion) Out[10]= {2, 4, 6}
```

To convince the skeptic.

```
(orion) In[11]:= Wait[Queue /@ Unevaluated[{$ProcessorID, $ProcessorID}]]
```

```
(orion) Out[11]= {1, 2}
```

```
(orion) In[12]:= Wait[Queue /@ {$ProcessorID, $ProcessorID}]
```

```
(orion) Out[12]= {0, 0}
```

Elegant, but Slow

A recipe for parallelizing functional operations.

```
(orion) In[13]:= Map[f, {a, b, c}]

(orion) Out[13]= {f[a], f[b], f[c]}

(orion) In[14]:= Composition[q, f][x]

(orion) Out[14]= q[f[x]]

(orion) In[15]:= Map[Composition[q, f], {a, b, c}]

(orion) Out[15]= {q[f[a]], q[f[b]], q[f[c]]}

(orion) In[16]:= Map[Composition[Queue, f], {a, b, c}]

(orion) Out[16]= {f[a], f[b], f[c]}

(orion) In[16]:= Wait[%]

(orion) In[17]:= Wait[Table[Queue[{i}, Prime[i]], {i, 1, 10}]]

(orion) Out[17]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
```

A Marketing Problem

In V1, everybody seemed to try out something like this at first.

```
(orion) In[18]:= ParallelTable[Sin[i], {i, 0.0, 100000.0}] // Short

      (orion) Out[18]//Short=
                {0., 0.841471, <<99998>>, 0.0357488}
```

What is the underlying primitive of parallel evaluation of many *Mathematica* commands?

If we know how many remote processors we have, we do not have to send out 100,000 processes of the form `Sin[`i`]`, but *one* process of the form `Table[Sin[i],{i,`i_0`,`i_1`}]` *to each processor*.

Types of Parallel Evaluations

```
(orion) In[19]:= {1 + 1, 2 + 2, 3 + 3, 4 + 4}

(orion) Out[19]= {2, 4, 6, 8}

(orion) In[20]:= Map[Prime, {1, 2, 3, 4, 5}]

(orion) Out[20]= {2, 3, 5, 7, 11}

(orion) In[21]:= Select[{1, 2, 3, 4, 5}, EvenQ]

(orion) Out[21]= {2, 4}

(orion) In[22]:= Count[{1, 2, 3, 4, 5}, _?OddQ]

(orion) Out[22]= 3

(orion) In[23]:= {a, b, c}.{x, y, z}

(orion) Out[23]= a x + b y + c z

(orion) In[24]:= Plus[1, 2, 3, 4, 5]

(orion) Out[24]= 15
```

Common theme: iteration over a list (or other long expression).

Efficient parallelization: do not create one process for each element of the list, but work in chunks.

The New ParallelEvaluate

The General Case

The general mechanism to parallelize all such cases:

- Split the input expression into chunks
- Apply a function to each chunk
- Combine the results

Here is `ParallelEvaluate`:

```
(orion) In[25]:= ?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.

(orion) In[26]:= ParallelEvaluate[h[1 + 2, 2 + 3, 3 + 4, 4 + 5, 5 + 6], f, comb]

(orion) Out[26]= comb[f[h[3]], f[h[5]], f[h[7]], f[h[9]], f[h[11]]]
```

Special Case 1: Join

For structure−preserving functions *f* the natural combination function is `Join`.

```
(orion) In[30]:= ParallelEvaluate[{1, 2, 3, 4, 5}, Prime]

(orion) Out[30]= {2, 3, 5, 7, 11}

(orion) In[31]:= ParallelEvaluate[{1, 2, 3, 4, 5}, Function[list, Map[f, list]]]

(orion) Out[31]= {f[1], f[2], f[3], f[4], f[5]}
```

Special Case 2: Identity

The default function to perform is the identity (do nothing).

```
(orion) In[32]:= ParallelEvaluate[{1 + 2, 2 + 3, 3 + 4, 4 + 5, 5 + 6}]

(orion) Out[32]= {3, 5, 7, 9, 11}
```

Special Case 3: Associative Functions

For associative *h*, the natural parallelization of

```
h[a, b, c, d, e]
```

is

```
h[h[a, b, c], h[d, e]]
```

That is, the function is the Identity, but the combiner is *h* itself.

```
(orion) In[33]:= ParallelEvaluate[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8]

(orion) Out[33]= 36
```

How to Distribute a Table?

The structure over which to parallelize Table[*expr*,*iterator*] is the iterator!

```
(orion) In[34]:= Table[i, {i, 1, 7}]

(orion) Out[34]= {1, 2, 3, 4, 5, 6, 7}
```

Of course, we should not generate it explicitly, but figure out suitable subiterators.

```
(orion) In[35]:= Join[Table[i, {i, 1, 4}], Table[i, {i, 5, 7}]]

(orion) Out[35]= {1, 2, 3, 4, 5, 6, 7}
```

Now, parallelization is easy.

```
(orion) In[36]:= ParallelDispatch[Join[Table[i, {i, 1, 4}], Table[i, {i, 5, 7}]]]

(orion) Out[36]= {1, 2, 3, 4, 5, 6, 7}
```

Load Balancing

If we know the relative performance of the remote processors, we can take this information into account when figuring out the chunks. Version 2 will allow you to measure and specify this information. This example uses remote kernels of different CPU speeds.

```
TableForm[{ProcessorID[#], #, ProcessorSpeed[#]} & /@ $Slaves,
 TableHeadings → {None, {"ID", "kernel", "speed setting"}}]
```

ID	kernel	speed setting
1	slave$_1$[localhost]	1
2	slave$_2$[sirius]	1
3	slave$_3$[sirius]	1
4	slave$_4$[vvv]	1

```
timings = First /@ RemoteEvaluate[Timing[N[Cos[1], 10000];;] /. Second → 1

{2.83, 1.59, 1.59, 1.46}

speeds = 1 / timings / Min[1 / timings]

{1., 1.77987, 1.77987, 1.93836}

Round[200 %]

{200, 356, 356, 388}

MapIndexed[(ProcessorSpeed[Extract[$Slaves, #2]] = #1) &, speeds]

{1., 1.77987, 1.77987, 1.93836}
```

Parallel Commands by the Dozen

General Idea

Examine each *Mathematica* command that operates on a list or other long expression.

```
f[list, arg2, arg3] ⟶
 ParallelEvaluate[list, Function[li, f[li, arg2, arg3]]]
```

Examples

```
(orion) In[37]:= ParallelEvaluate[{1 + 1, 2 + 1, 3 + 1, 4 + 1, 5 + 1},
                  Function[li, Select[li, EvenQ]]]

(orion) Out[37]= {2, 4, 6}

(orion) In[38]:= ParallelEvaluate[{1 + 1, 2 + 1, 3 + 1, 4 + 1, 5 + 1},
                  Function[li, Cases[li, _?OddQ]]]

(orion) Out[38]= {3, 5}

(orion) In[39]:= With[{n = 5},
                  mata = Table[Random[Real, {-1, 1}], {n}, {n}];
                  matb = Table[Random[Real, {-1, 1}], {n}, {n}];
                 ]

(orion) In[40]:= With[{mata = mata, matb = matb},
                  par = ParallelEvaluate[mata, Dot[#, matb] &]
                 ];
                 MatrixForm[par]

(orion) Out[41]//MatrixForm=
                 ⎛ -0.413189  1.97472  -0.764574  0.398395   0.390085 ⎞
                 ⎜  0.340144  1.05595   0.140281   1.83771    0.39134 ⎟
                 ⎜ -0.020901  1.99898  -0.127458 0.0551463  -0.427745 ⎟
                 ⎜  2.14046  0.732813   1.41867  0.783665   0.345912 ⎟
                 ⎝ 0.0672538 0.127011  0.799246 -0.172638  -1.12883  ⎠
```

Less Obvious

For functions that do not preserve the structure, a custom combiner needs to be chosen.

```
(orion) In[42]:= ParallelEvaluate[{1 + 1, 2 + 1, 3 + 1, 4 + 1, 5 + 1}, Count[#, _?OddQ] &, Plus]

(orion) Out[42]= 2

(orion) In[43]:= ParallelEvaluate[{1, 2, 3, 4, 5, 6, 7}, MemberQ[#, 5] &, Or]

(orion) Out[43]= True

(orion) In[44]:= ParallelEvaluate[{1, 2, 3, 4, 5, 6, 7}, MemberQ[#, 9] &, Or]

(orion) Out[44]= False

(orion) In[45]:= ParallelEvaluate[{1, 2, 3, 4, 5, 6, 7}, FreeQ[#, 5] &, And]

(orion) Out[45]= False

(orion) In[46]:= ParallelEvaluate[{1, 2, 3, 4, 5, 6, 7}, FreeQ[#, 9] &, And]

(orion) Out[46]= True
```

Associative Functions

```
(orion) In[47]:= ParallelEvaluate[1 + 2 + 3 + 4 + 5 + 6 + 7 + 8]

(orion) Out[47]= 36
```

```
(orion) In[48]:= ParallelEvaluate[Max[3, 5, 7, 3]]

(orion) Out[48]= 7

(orion) In[49]:= ParallelEvaluate[GCD[2, 4, 6, 8]]

(orion) Out[49]= 2

(orion) In[50]:= ParallelEvaluate[StringJoin["parallel ", "computation ", "is ", "fun."]]

(orion) Out[50]= parallel computation is fun.
```

It's Always the Same!

As we saw, to parallelize

```
(orion) In[51]:= Count[{1, 2, 3, 4, 5, 6, 7}, _?PrimeQ]

(orion) Out[51]= 4
```

you have to say

```
(orion) In[52]:= ParallelEvaluate[{1, 2, 3, 4, 5, 6, 7}, Count[#, _?PrimeQ] &, Plus]

(orion) Out[52]= 4
```

The idea is always the same. Wouldn't this be nice:

```
(orion) In[53]:= ParallelEvaluate[Count[{1, 2, 3, 4, 5, 6, 7}, _?PrimeQ]]

(orion) Out[53]= 4
```

It is quite simple:

```
Count /: ParallelEvaluate[Count[list_, params___]] :=
 ParallelEvaluate[list, Count[#, params] &, Plus]
```

All of these have been overloaded.

```
(orion) In[54]:= $ParallelCommands

(orion) Out[54]= {Cases, Select, Count, MemberQ, FreeQ,
                  Map, Apply, Outer, Inner, Dot, Table, Sum, Product}
```

The Ultimate Goal

It is still a long way from

```
ParallelEvaluate[your program goes here ...]
```

Nondeterminism

Waiting for a Result

Idea: try out different methods to solve a problem. The first one to succeed wins, the others are then aborted. We can also allow for one method to return `$Failed` to indicate that it did not succeed. Waiting then continues until one result is found that is other than `$Failed`.

It is already supported in V1. Queue each evaluation as a process, then use `WaitOne[]`. This returns with one of the results. You can then abort the others (this part does not work well in V1).

```
(orion) In[55]:= ids = {Queue[1 + 1], Queue[2 + 2]}

(orion) Out[55]= {pid25, pid26}

(orion) In[56]:= {res, id, ids} = WaitOne[ids]

(orion) Out[56]= {2, pid25, {pid26}}

(orion) In[57]:= ResetSlaves[];

(orion) In[58]:= res

(orion) Out[58]= 2
```

Parallel Try

Here is a function that combines these steps.

The function `ParallelTry[{exprs...}]` takes a list of expressions as argument and submits them to remote kernels. The first result other than `$Failed` is returned, and any remaining computations are aborted.

```
ParallelTry[exprs_] :=
 Module[{res = $Failed, id, ids},
  ids = List @@ Queue /@ Unevaluated[exprs];
  While[res === $Failed && Length[ids] > 0, (* try one more *)
   {res, id, ids} = WaitOne[ids]
  ];
  ResetSlaves[];
  res
 ]
```

Finding Irreducible Polynomials

We want to find an irreducible polynomial modulo n.

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

```
p = 2 (*the modulus *)

2

randPoly[n_Integer?Positive, x_, p_] :=
 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]

poly = randPoly[10, x, p]

1 + x^3 + x^8 + x^10

irreducibleQ[poly, p]

False
```

The Loop

This loops keeps trying until it finds an irreducible one.

```
While[! irreducibleQ[poly = randPoly[100, x, p], p], Null]; poly
```

$$1 + x + x^2 + x^7 + x^8 + x^{10} + x^{12} + x^{14} + x^{15} + x^{16} + x^{17} + x^{19} + x^{20} + x^{22} + x^{24} + x^{26} + x^{28} + x^{29} + x^{33} + x^{34} + x^{35} + x^{36} + x^{37} + x^{39} + x^{43} + x^{44} + x^{45} + x^{46} + x^{47} + x^{48} + x^{49} + x^{50} + x^{51} + x^{52} + x^{56} + x^{57} + x^{58} + x^{59} + x^{61} + x^{62} + x^{63} + x^{68} + x^{69} + x^{70} + x^{72} + x^{74} + x^{76} + x^{78} + x^{80} + x^{82} + x^{86} + x^{87} + x^{90} + x^{91} + x^{92} + x^{94} + x^{95} + x^{97} + x^{100}$$

Here is the function that combines these steps.

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

findOne[100, p]
```

$$1 + x^2 + x^4 + x^5 + x^7 + x^9 + x^{10} + x^{11} + x^{12} + x^{15} + x^{17} + x^{18} + x^{21} + x^{22} + x^{25} + x^{26} + x^{27} + x^{33} + x^{34} + x^{36} + x^{37} + x^{38} + x^{39} + x^{43} + x^{45} + x^{46} + x^{47} + x^{48} + x^{54} + x^{56} + x^{60} + x^{61} + x^{62} + x^{64} + x^{66} + x^{71} + x^{72} + x^{75} + x^{76} + x^{77} + x^{78} + x^{81} + x^{83} + x^{84} + x^{91} + x^{92} + x^{93} + x^{94} + x^{95} + x^{96} + x^{97} + x^{98} + x^{100}$$

In Parallel

```
ExportEnvironment[randPoly, irreducibleQ, findOne]
```

Generate list of identical commands, one for each remote kernel.

```
With[{n = 512, p = 2},
 Join @@ Table[Hold[findOne[n, p]], {Length[$Slaves]}]
]
```

```
Hold[findOne[1000, 2], findOne[1000, 2],
 findOne[1000, 2], findOne[1000, 2], findOne[1000, 2]]
```

Let them loose.

```
ParallelTry[%]
```

$$1 + x^2 + x^3 + x^5 + x^{11} + x^{14} + x^{15} + x^{17} + x^{20} + x^{22} + x^{23} + x^{25} + x^{28} + x^{29} + x^{34} + x^{39} + x^{40} + x^{41} + x^{43} + x^{44} + x^{45} + x^{47} + x^{49} + x^{50} + x^{53} + x^{54} + x^{56} + x^{57} + x^{58} + x^{59} + x^{62} + x^{64} + x^{65} + x^{68} + x^{69} + x^{83} + x^{86} + x^{87} + x^{98} + x^{101} + x^{102} + x^{105} + x^{109} + x^{110} + x^{111} + x^{112} + x^{113} + x^{114} + x^{115} + x^{116} + x^{118} + x^{119} + x^{121} + x^{125} + x^{126} + x^{127} + x^{128} + x^{130} + x^{131} + x^{134} + x^{136} + x^{139} + x^{141} + x^{143} + x^{145} + x^{147} + x^{150} + x^{151} + x^{154} + x^{155} + x^{158} + x^{162} + x^{164} + x^{165} + x^{167} + x^{168} + x^{170} + x^{171} + x^{173} + x^{177} + x^{179} + x^{180} + x^{181} + x^{182} + x^{184} + x^{185} + x^{190} + x^{192} + x^{195} + x^{197} + x^{198} + x^{200} + x^{201} + x^{203} + x^{204} + x^{205} + x^{207} + x^{208} + x^{211} + x^{213} + x^{214} + x^{215} + x^{219} + x^{220} + x^{224} + x^{226} + x^{230} + x^{232} + x^{233} + x^{234} + x^{235} + x^{238} + x^{239} + x^{240} + x^{242} + x^{244} + x^{245} + x^{248} + x^{249} + x^{250} + x^{252} + x^{255} + x^{256} + x^{260} + x^{261} + x^{262} + x^{264} + x^{265} + x^{266} + x^{267} + x^{269} + x^{271} + x^{274} + x^{275} + x^{277} + x^{279} + x^{280} + x^{283} + x^{284} + x^{285} + x^{288} + x^{290} + x^{291} + x^{294} + x^{297} + x^{298} + x^{299} + x^{300} + x^{301} + x^{302} + x^{304} + x^{305} + x^{306} + x^{308} + x^{312} + x^{314} + x^{318} + x^{319} + x^{322} + x^{326} + x^{329} + x^{330} + x^{333} + x^{335} + x^{336} + x^{338} + x^{341} + x^{342} + x^{344} + x^{345} + x^{346} + x^{348} + x^{349} + x^{352} + x^{353} + x^{354} + x^{355} + x^{357} + x^{358} + x^{362} + x^{364} + x^{366} + x^{368} + x^{371} + x^{372} + x^{374} + x^{377} + x^{378} + x^{379} + x^{380} + x^{382} + x^{383} + x^{384} + x^{385} + x^{387} + x^{392} + x^{394} + x^{397} + x^{399} + x^{400} + x^{403} + x^{404} + x^{406} + x^{409} + x^{414} + x^{415} + x^{418} + x^{419} + x^{420} + x^{421} + x^{427} + x^{439} + x^{440} + x^{441} + x^{442} + x^{443} + x^{447} + x^{448} + x^{451} + x^{452} + x^{453} + x^{455} + x^{456} + x^{459} + x^{462} + x^{464} + x^{465} + x^{466} + x^{467} + x^{468} + x^{473} + x^{474} + x^{475} + x^{479} + x^{480} + x^{481} + x^{482} + x^{483} + x^{485} + x^{490} + x^{493} + x^{494} + x^{496} + x^{497} + x^{498} + x^{499} + x^{500} + x^{501} + x^{502} + x^{503} + x^{505} + x^{506} + x^{509} + x^{510} + x^{512} + x^{514} + x^{515} + x^{517} + x^{518} + x^{519} + x^{521} + x^{522} + x^{526} + x^{531} + x^{532} + x^{533} + x^{534} + x^{536} + x^{537} + x^{538} + x^{541} + x^{545} + x^{546} + x^{547} + x^{549} + x^{550} + x^{551} + x^{552} + x^{554} + x^{555} + x^{556} + x^{557} + x^{558} + x^{561} + x^{566} + x^{567} + x^{570} + x^{571} + x^{572} + x^{574} + x^{575} + x^{577} + x^{584} + x^{585} + x^{586} + x^{588} + x^{589} + x^{593} + x^{594} + x^{600} + x^{603} + x^{604} + x^{605} + x^{607} + x^{609} + x^{612} + x^{613} + x^{614} + x^{615} + x^{618} + x^{620} + x^{623} + x^{625} + x^{630} + x^{632} + x^{633} + x^{634} + x^{638} + x^{642} + x^{644} + x^{646} + x^{648} + x^{649} + x^{652} + x^{655} + x^{656} + x^{659} + x^{660} + x^{661} + x^{662} + x^{663} + x^{664} + x^{665} + x^{666} + x^{667} + x^{668} + x^{669} + x^{670} + x^{671} + x^{672} + x^{673} + x^{674} + x^{679} + x^{681} + x^{682} + x^{683} + x^{684} + x^{685} + x^{687} + x^{688} + x^{689} + x^{692} + x^{695} + x^{697} + x^{702} + x^{703} + x^{704} + x^{706} + x^{707} + x^{708} + x^{710} + x^{714} + x^{719} + x^{720} + x^{724} + x^{725} + x^{731} + x^{732} + x^{734} + x^{741} + x^{748} + x^{749} + x^{751} + x^{752} + x^{753} + x^{754} + x^{755} + x^{759} + x^{766} + x^{767} + x^{769} + x^{771} + x^{773} + x^{775} + x^{777} + x^{782} + x^{783} + x^{784} + x^{785} + x^{790} + x^{791} + x^{792} + x^{793} + x^{795} + x^{797} + x^{798} + x^{799} + x^{800} + x^{802} + x^{803} + x^{804} + x^{805} + x^{806} + x^{808} + x^{810} + x^{811} + x^{812} + x^{813} + x^{815} + x^{816} + x^{817} + x^{818} + x^{819} + x^{821} + x^{822} + x^{823} + x^{828} + x^{829} + x^{830} + x^{831} + x^{833} + x^{835} + x^{836} + x^{837} + x^{840} + x^{842} + x^{843} + x^{844} + x^{846} + x^{847} + x^{848} + x^{850} + x^{853} + x^{854} + x^{855} + x^{856} + x^{858} + x^{863} + x^{865} + x^{866} + x^{867} + x^{868} + x^{869} + x^{870} + x^{871} + x^{872} + x^{873} + x^{877} + x^{878} + x^{879} + x^{884} + x^{887} + x^{888} + x^{890} + x^{891} + x^{892} + x^{893} + x^{896} + x^{898} + x^{900} + x^{901} + x^{902} + x^{903} + x^{905} + x^{907} + x^{909} + x^{910} + x^{911} + x^{913} + x^{916} + x^{917} + x^{919} + x^{923} + x^{925} + x^{926} + x^{928} + x^{929} + x^{931} + x^{932} + x^{933} + x^{934} + x^{935} + x^{937} + x^{939} + x^{940} + x^{943} + x^{947} + x^{949} + x^{952} + x^{957} + x^{961} + x^{962} + x^{963} + x^{965} + x^{968} + x^{969} + x^{970} + x^{972} + x^{973} + x^{978} + x^{979} + x^{980} + x^{983} + x^{984} + x^{985} + x^{986} + x^{988} + x^{989} + x^{990} + x^{992} + x^{996} + x^{998} + x^{1000}$$

(In this example, 853 polynomials were examined.)

Comparing Algorithms

This example from the new collection of sample applications shows how to compare the different methods for numerical minimization available in *Mathematica* V5.

```
f[x_, y_] := ℯ^Sin[50 x] + Sin[60 ℯ^y] +
  Sin[70 Sin[x]] + Sin[Sin[80 y]] - Sin[10 (x + y)] + 1/4 (x^2 + y^2)
```

We can run all methods in parallel to compare the results.

```
TableForm[ParallelMap[tryMethod, methods], TableDepth → 2,
 TableHeadings → {None, {Method, Timing, Minimum, Position}}]
```

Method	Timing	Minimum	Position
DifferentialEvolution	0.51	0.998004	{x → -31.9783, y → -31.97
NelderMead	0.46	16.4409	{x → -15.9634, y → 15.963
RandomSearch	1.11	0.998004	{x → -31.9783, y → -31.97
SimulatedAnnealing	1.77	0.998004	{x → -31.9783, y → -31.97
Automatic	0.38	15.5038	{x → -31.9317, y → 15.965

Here we use `ParallelTry` to find the fastest method and abort all other calculations. There is no point in including the `Automatic` method setting, because it implies on of the other methods.

```
trials = tryMethod /@ Hold @@ Complement[methods, {Automatic}];

ParallelTry @@ trials
```

```
{NelderMead, 0.38, 16.4409, {x → -15.9634, y → 15.9634}}
```

Shared Memory

Communication

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.

```
(orion) In[59]:= s1 = $Slaves[[1]]; s2 = $Slaves[[2]];
```

Declare shared variables.

```
(orion) In[60]:= SharedVariables[{x}]
```

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

```
(orion) In[61]:= x = 0;

(orion) In[62]:= RemoteEvaluate[x = x + 1, s1]

(orion) Out[62]= 1

(orion) In[63]:= RemoteEvaluate[x, s2]

(orion) Out[63]= 1
```

The new value is stored globally.

```
(orion) In[64]:= x

(orion) Out[64]= 1
```

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

```
(orion) In[65]:= RemoteEvaluate[x++, s2]

(orion) Out[65]= 1

(orion) In[66]:= RemoteEvaluate[x, s1]

(orion) Out[66]= 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.

```
(orion) In[67]:= 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.

```
(orion) In[68]:= SharedVariables[{y, lock}]

(orion) In[69]:= 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]]

(orion) Out[70]= {5, 4, 3, 2, 1}

(orion) In[71]:= ClearSlaves[];
```

Shared Downvalues

Delayed definitions (downvalues) of shared variables can be evaluated on the master kernel or the remote kernel that asked for it. Shared downvalues can be used to access shared resources such as a global queue.

Debugging News

Tracing Events

```
(orion) In[72]:= Options[$DebugObject]

(orion) Out[72]= {Trace → {}, TraceHandler → Print}
```

```
(orion) In[73]:= OptionValues[Trace]
(orion) Out[73]= {MathLink, Queueing, SendReceive, SharedMemory}
(orion) In[74]:= SetOptions[$DebugObject, Trace → {SendReceive, Queueing, SharedMemory}]
(orion) Out[74]= {Trace → {Queueing, SendReceive, SharedMemory}}
(orion) In[75]:= SetOptions[$DebugObject, TraceHandler → Save]
(orion) Out[75]= {TraceHandler → Save}
```

Running A Computation with Tracing

```
(orion) In[76]:= SharedVariables[x]; x = 5;
(orion) In[77]:= Wait[{Queue[x = x + 1], Queue[x = x + 1]}]
(orion) Out[77]= {6, 6}
```

Analyzing the Trace

```
(orion) In[78]:= TableForm[TraceList[], TableDepth → 2,
                  TableHeadings → {Automatic, {"Trigger", "Event"}}]
```

(orion) Out[78]//TableForm=

	Trigger	Event
1	SendReceive	Sending to $slave_1$[n1]: –internal value– (q=1)
2	SendReceive	Sending to $slave_2$[n2]: –internal value– (q=1)
3	SendReceive	Sending to $slave_3$[n3]: –internal value– (q=1)
4	SendReceive	Sending to $slave_4$[n4]: –internal value– (q=1)
5	SendReceive	Sending to $slave_5$[n5]: –internal value– (q=1)
6	SendReceive	Sending to $slave_6$[n6]: –internal value– (q=1)
7	SendReceive	Sending to $slave_7$[n7]: –internal value– (q=1)
8	SendReceive	Sending to $slave_8$[n8]: –internal value– (q=1)
9	SendReceive	Sending to $slave_9$[n9]: –internal value– (q=1)
10	SendReceive	Sending to $slave_{10}$[n10]: –internal value– (q=1)
11	SendReceive	Sending to $slave_{11}$[n11]: –internal value– (q=1)
12	SendReceive	Receiving from $slave_1$[n1]: –internal value– (q=0)
13	SendReceive	Receiving from $slave_2$[n2]: –internal value– (q=0)
14	SendReceive	Receiving from $slave_3$[n3]: –internal value– (q=0)
15	SendReceive	Receiving from $slave_4$[n4]: –internal value– (q=0)
16	SendReceive	Receiving from $slave_5$[n5]: –internal value– (q=0)
17	SendReceive	Receiving from $slave_6$[n6]: –internal value– (q=0)
18	SendReceive	Receiving from $slave_7$[n7]: –internal value– (q=0)
19	SendReceive	Receiving from $slave_8$[n8]: –internal value– (q=0)
20	SendReceive	Receiving from $slave_9$[n9]: –internal value– (q=0)
21	SendReceive	Receiving from $slave_{10}$[n10]: –internal value– (q=0
22	SendReceive	Receiving from $slave_{11}$[n11]: –internal value– (q=0
23	Queueing	pid_1 queued (0)
24	Queueing	pid_2 queued (1)
25	SendReceive	Sending to $slave_1$[n1]: pid_1[x = x + 1] (q=1)
26	Queueing	pid_1 on $slave_1$[n1]
27	SendReceive	Sending to $slave_2$[n2]: pid_2[x = x + 1] (q=1)
28	Queueing	pid_2 on $slave_2$[n2]
29	SharedMemory	$slave_1$[n1]: x ⟶ 5
30	SharedMemory	$slave_2$[n2]: x ⟶ 5
31	SharedMemory	$slave_1$[n1]: x = 6 ⟶ 6
32	SharedMemory	$slave_2$[n2]: x = 6 ⟶ 6
33	SendReceive	Receiving from $slave_1$[n1]: pid_1[6] (q=0)
34	Queueing	pid_1 done
35	SendReceive	Receiving from $slave_2$[n2]: pid_2[6] (q=0)
36	Queueing	pid_2 done
37	Queueing	pid_1 dequeued
38	Queueing	pid_2 dequeued

```
(orion) In[79]:= newTraceList[]

(orion) In[80]:= SetOptions[$DebugObject, TraceHandler → Print, Trace → {}]

(orion) Out[80]= {Trace → {}, TraceHandler → Print}

(orion) In[81]:= ClearSlaves[];
```

The New Examples Directory

From Application Examples, an annotated list of all new examples in the `Examples` subdirectory:

- Performance Measurement and Calibration presents ideas for measuring performance of a parallel computation and shows how to calibrate remote processor speeds for better scheduling.
- Interval Minimum presents a distributed global minimization using interval arithmetic.
- Diameter of a Point Set gives a parallel solution to a classic programming problem. This example is of a tutorial nature.
- Single−Image Stereograms shows how an existing sequential program can be parallelized with a few small modifications.
- Advanced Use of Shared Variables presents various advanced topics on the use of virtual shared memory.
- Parallel Try shows how to set up nondeterministic computations.
- Large Calculations gives examples of large, infinite, or batch computations.
- Graphics and Animations provides examples for using the `Parallel‘Commands‘` package.
- The Cascade explains how to evaluate associative expressions using a binary tree.

Summary

- Parallel computation with *Mathematica*
- Distribution of loop−like *Mathematica* commands
- Automatic Parallelization
- Nondeterministic Parallelism