Parallel Computing with Mathematica
UVACSE Short Course

E Hall[1]

[1]University of Virginia Alliance for Computational Science and Engineering
uvacse@virginia.edu

October 8, 2014

Outline

Installing and Configuring NX Client

The NX client provides a Gnome Linux desktop interface to the login node of the fir.itc Linux cluster.

http://uvacse.virginia.edu/resources/itc-linux-cluster-overview/the-nx-client/

Starting and Configuring NX Client

Once logged into `fir.itc.virginia.edu` through NX

- Open a terminal from Applications/System Tools/Terminal menu
  - Select and right-click on Terminal to add to launcher
- Create a `mathematica` directory with `mkdir` command
- Start web browser from icon at top of desktop

Download Short Course Examples

Download the short-course materials from
http://www.uvacse.virginia.edu/software/mathematica-at-uva/

Follow the links,
→Parallel Mathematica
→Parallel Mathematica Short Course
and download 3 files to `mathematica` directory you create with `mkdir` command

- ClassExamples_Fa14.zip
- mathematica-parallel_Fa14.pdf

Solving Big Technical Problems

Computationally intensive, long-running codes

- Run tasks in parallel on many processors
- Task parallel

Large Data sets

- Load data across multiple machines that work in parallel
- Data parallel

Parallel Computing with Mathematica

Parallel computing in Mathematica is based on launching and controlling multiple Mathematica kernel (worker) processes from within a single master Mathematica, providing a distributed-memory environment for parallel programming.

Tested on Unix, Linux, Windows, and Macintosh platforms and are well suited to working locally on a multi-core machine or in a cluster of machines,

Parallel computing is now provided as a standard part of Mathematica.

Parallel Computing with Mathematica

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 process is an expression to be evaluated.

Connection Methods

Mathematica can run parallel workers locally on the same machine or remotely on a compute cluster controlled by a resource management application, e.g. PBSPro.

Local Kernels

The Local Kernels connection method is used to run parallel workers on the same computer as the master Mathematica. It is suitable for a multi-core environment, and is the easiest way to get up and running with parallel computation.

```
In[25]:= $ProcessorCount

Out[25]= 2
```

Cluster Integration

The Cluster Integration connection method is used to run parallel workers on a compute cluster from the master Mathematica process. It integrates with the PBSPro cluster management software.

Parallel Computing Functions

mathematica/guide/ParallelComputing.html

Automatic Parallelization

Parallelize — evaluate an expression using automatic parallelization

ParallelTry — try different computations in parallel, giving the first result obtained

Computation Setup & Broadcasting »

ParallelEvaluate — evaluate an expression on all parallel subkernels

DistributeDefinitions — distribute definitions to all parallel subkernels

ParallelNeeds — load the same package into all parallel subkernels

Parallel Computing Functions

Data Parallelism »

ParallelMap ▪ **ParallelTable** ▪ **ParallelSum** ▪ **...**

ParallelCombine — evaluate expressions in parallel and combine their results

---

Concurrency Control »

ParallelSubmit — submit expressions to be evaluated concurrently

WaitAll — wait for all concurrent evaluations to finish

WaitNext — wait for the next of a list of concurrent evaluations to finish

---

Shared Memory & Synchronization »

SetSharedVariable — specify symbols with values to synchronize across subkernels

SetSharedFunction — specify functions whose evaluations are to be synchronized

$SharedVariables ▪ **$SharedFunctions** ▪ **UnsetShared** ▪ **CriticalSection**

Parallel Computing Functions

Setup and Configuration »

LaunchKernels — launch a specified number of subkernels

$KernelCount — number of running subkernels

$KernelID ▪ **Kernels** ▪ **AbortKernels** ▪ **CloseKernels** ▪ **...**

$ProcessorCount — number of processor cores on the current computer

---

Multi-Processor and Multicore Computation

Compile — create compiled functions that run in parallel

Parallelization — execute compiled functions in parallel

CompilationTarget — create machine-level parallel compiled functions

---

GPU Computing »

CUDAFunctionLoad — load a function to run on a GPU using CUDA

OpenCLFunctionLoad — load a function to run on a GPU using OpenCL

Parallel Computing Features

Mathematica supports both task and data parallelism.

The main features of parallel computing in Mathematica 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
- virtual process scheduling or explicit process distribution to available processors
- virtual shared memory, synchronization, locking
- parallel functional programming and automatic parallelization support

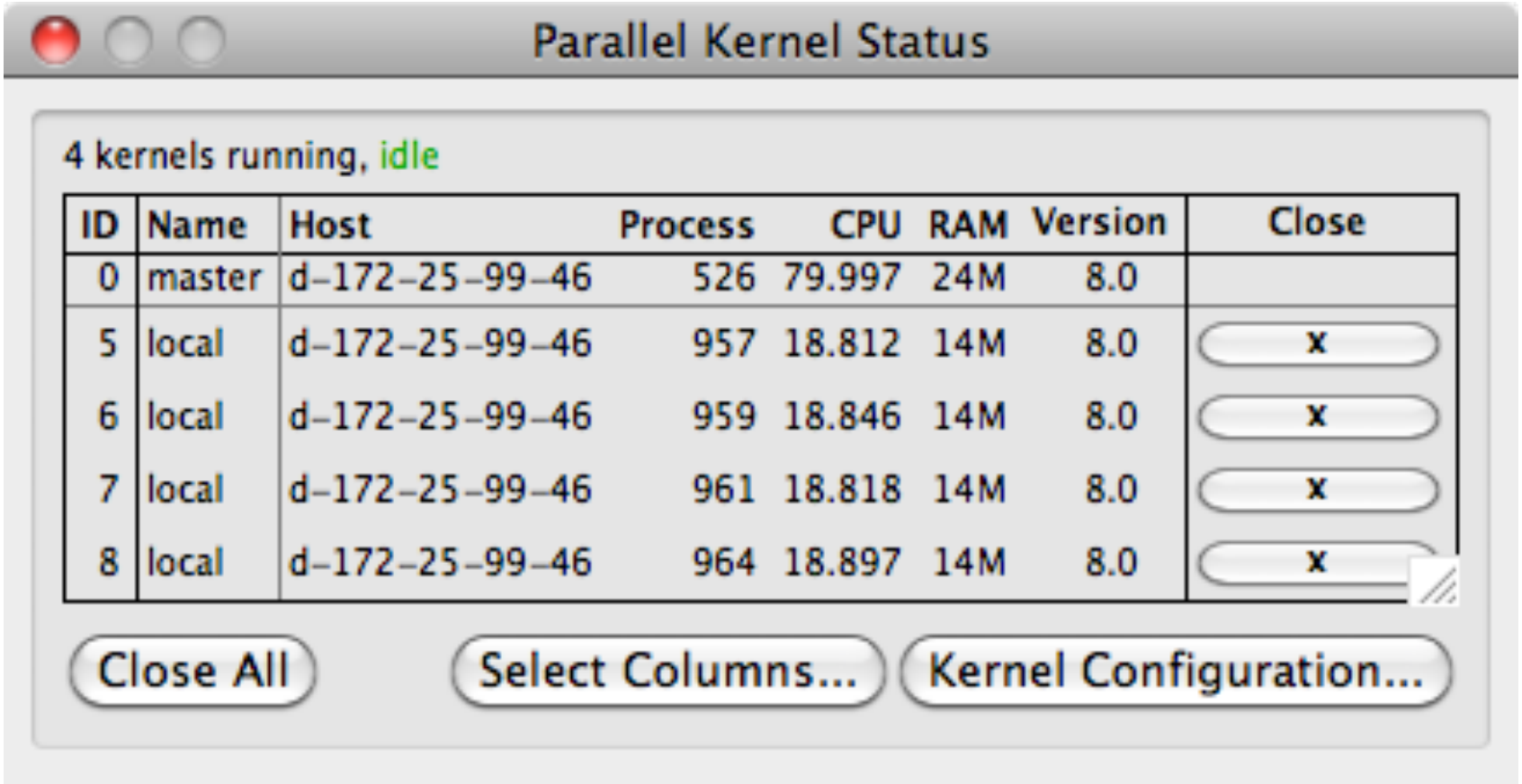
Programming Parallel Applications in Mathematica

To demonstrate demonstrate parallel processing in Mathematica, we use `ParallelEvaluate`. If this is the first parallel computation, it will launch the configured parallel kernels.

```
In[1]:= $ProcessorCount

Out[1]= 4

In[2]:= ParallelEvaluate[$ProcessID]

Out[2]= {957, 959, 961, 964}

In[3]:= ParallelEvaluate[$MachineName]

Out[3]= {d-172-25-99-46, d-172-25-99-46,
          d-172-25-99-46, d-172-25-99-46}
```

Parallel Kernels Status Monitor

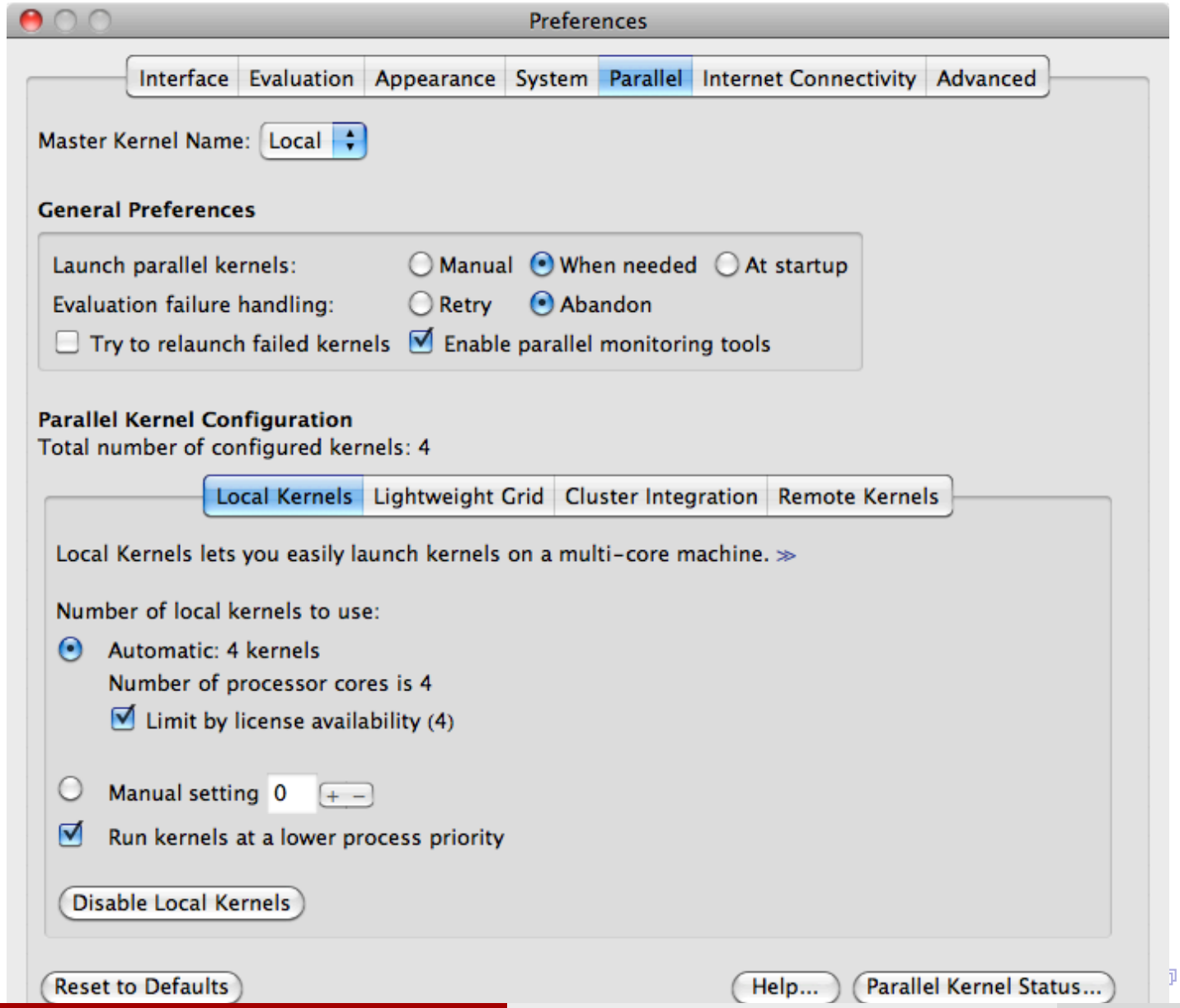
You can open the Parallel Kernels Status monitor through the Evaluation > Parallel Kernel Status menu selection.



Parallel Preferences

The default settings of Mathematica automatically configure a number of parallel kernels to use for parallel computation, as seen through the Evaluation > Parallel Kernel Configuration menu selection.

Parallel Preferences



Launching and Connecting

Mathematica launches parallel kernels automatically as they are needed, but you can also launch kernels manually with the command `LaunchKernels`. This is be useful if you were running in a batch mode.

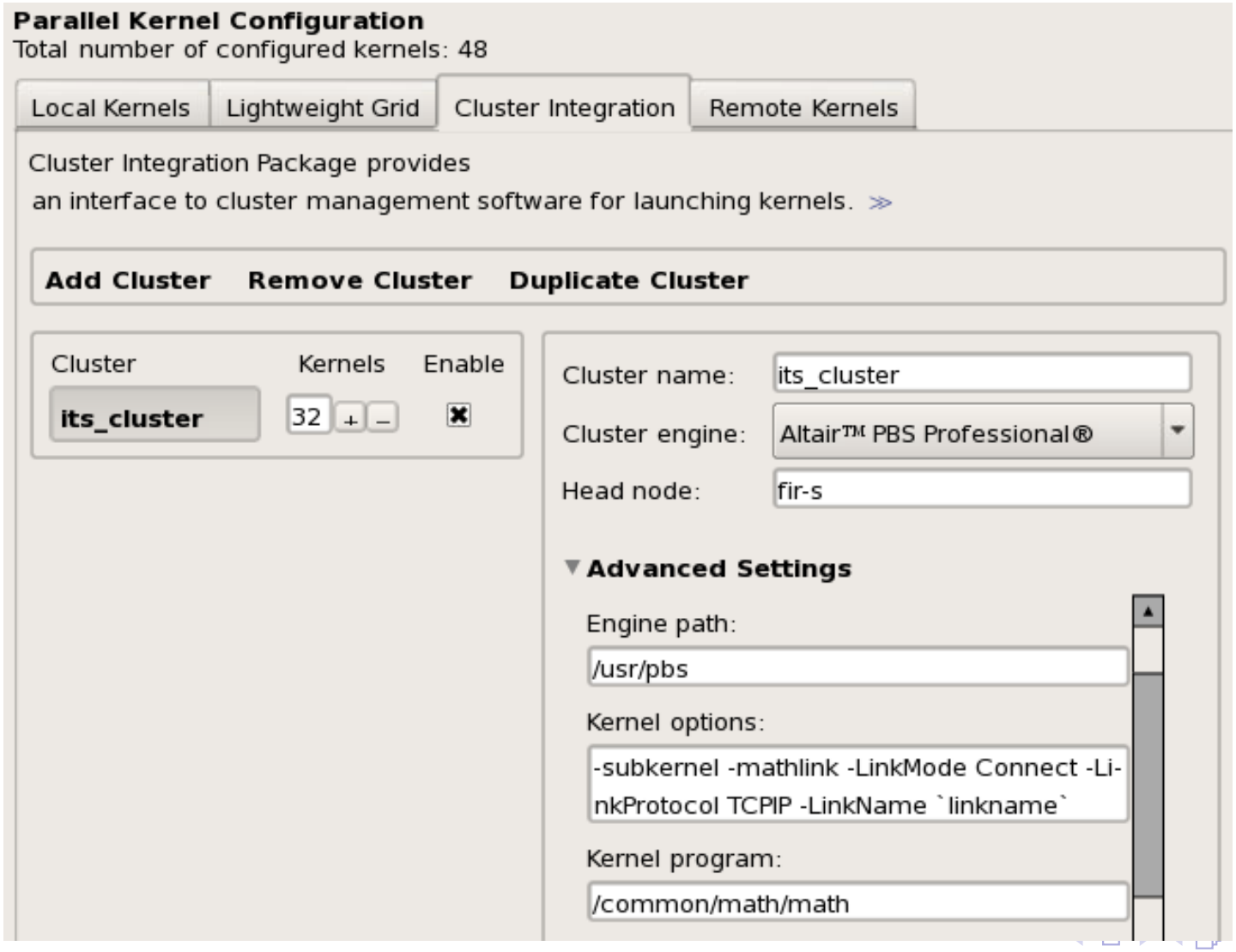
Local Kernels

The Local Kernels connection method supports launching local kernels directly from `LaunchKernels` by passing it an integer (setting the number of kernels)

```
In[1]:= LaunchKernels[4]

Out[1]= {KernelObject[1, local], KernelObject[2, local],
         KernelObject[3, local], KernelObject[4, local]}
```

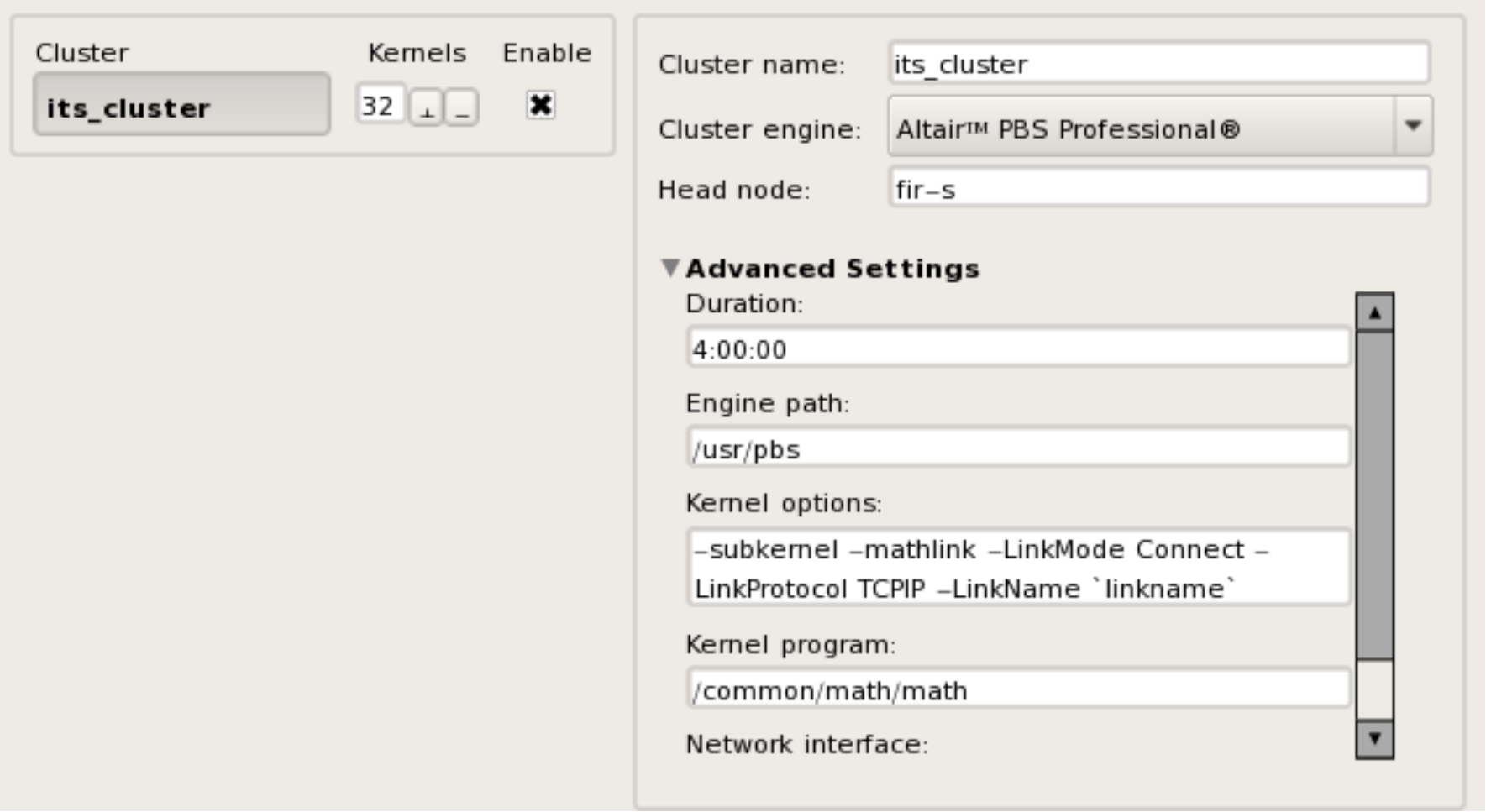
Launching and Connecting

Cluster Integration configuration done through Parallel Preferences



Launching and Connecting

Cluster Integration configuration done through Parallel Preferences



Launching and Connecting

Cluster Integration

The Cluster Integration connection method is used to run parallel workers on the compute nodes of a cluster from the master Mathematica.

The Cluster Integration connection method supports launching kernels directly from `LaunchKernels`. To do so you must first load the ClusterIntegration‘ package, this is shown below.

To launch on a particular cluster you have to pass the name for that cluster into LaunchKernels.

Launching and Connecting

Cluster Integration

```
<< "ClusterIntegration`"
ClusterIntegration`PBS`Private`DebugPrint := Print
NumKernels=8;

LaunchKernels[
 PBS["localhost", {"EnginePath" -> "/usr/pbs/",
   "KernelProgram" -> "/common/math/math", "ToQueue" -> True,
   "QueueName" -> "standard",
   "BatchCommand" ->
    "#!/bin/bash \n #PBS -N test \n #PBS -q standard \n #PBS -l \
walltime=00:10:00 \n #PBS -o \
/home/teh1m/math/gridmathematica/cse-030/test_out1 \n #PBS -j oe \
 \n #PBS -l select=1  \n `mathkernel`"}], NumKernels ]
```

Sending Commands to Remote Kernels

Values of variables defined on the local master kernel are usually not available to remote kernels.

```
In[15]:= mykernel = First[Kernels[]]

Out[15]= KernelObject[1, local]

In[16]:= a = 2;
         ParallelEvaluate[a === 2, mykernel]

Out[17]= False
```

A convenient way to insert variable values into unevaluated commands is to use `With`, as demonstrated in the following command. The symbol `a` is replaced by 2, then the expression `2 === 2` is sent to the remote kernel where it evaluates to `True`.

```
In[18]:= With[{a = 2}, ParallelEvaluate[a === 2, mykernel] ]

Out[18]= True
```

If you need variable values and definitions carried over to the remote kernels, use `DistributeDefinitions` or shared variables.

Sending Commands to Remote Kernels

Recall that connections to remote kernels, as opened by `LaunchKernels`, are represented as kernel objects. The commands below take parallel kernels as arguments and use them to carry out computations.

Low-Level Parallel Evaluation

`ParallelEvaluate[`*cmd*`,`*kernel*`]`	sends *cmd* for evaluation to the parallel kernel *kernel*, then waits for the result and returns
`ParallelEvaluate[`*cmd*`, {`*kernels*`}]`	sends *cmd* for evaluation to the parallel kernels given, then waits for the results and returns them
`ParallelEvaluate[`*cmd*`]`	sends *cmd* for evaluation to all parallel kernels and returns the list of results; equivalent to `ParallelEvaluate[`*cmd*`, Kernels[]]`

Remote Definitions

Mathematica contains a command `DistributeDefinitions` that makes it easy to transport local variables and function definitions to all parallel kernels.

`DistributeDefinitions[`s_1`,`s_2`, ... ]`	distribute all definitions for symbols s_i to all remote kernels
`DistributeDefinitions["`*Context*`"]`	distributes definitions for all symbols in the specified context

Higher-level parallel commands, such as `Parallelize`, `ParallelTable`, `ParallelSum`, ... will automatically distribute definitions of symbols occurring in their arguments.

Sending Commands to Remote Kernels

Parallel Mapping and Iterators

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

`ParallelMap[f, h[e1, e2, ...]]`	evaluates `h[f[e1], [f[e2], ...]` in parallel
`ParallelTable[ expr, {i, i0, i1, di}, {j, j0, j1, dj}, ...]`	builds `Table[expr, {i, i0, i1, di}, {j, j0, j1, dj}, ...]` in parallel; parallelization occurs along the first (outermost) iterator `{i, i0, i1, di}`
`ParallelSum[...], ParallelProduct[...]`	computes sums and products in parallel

Parallel evaluation, mapping, and tables.

`ParallelMap[f, h[e1, e2, ...]]` is a parallel version of `f /@ h[e1, e2, ...]` evaluating the individual `f[ei]` in parallel rather than sequentially.

Automatic Distribution of Definitions

Parallel commands such as `ParallelTable` will automatically distribute the values and functions needed, using effectively `DistributeDefinitions`.

For this parallel table, the function `f` and the iterator bound `n` will evaluate on the subkernels, so their definitions need to be distributed to make it work.

```
In[1]:= f[x_, y_] := x^y
        {m, n} = {4, 2};

In[3]:= ParallelTable[f[i, j], {i, m}, {j, n}]

Out[3]= {{1, 1}, {2, 4}, {3, 9}, {4, 16}}
```

This automatic distribution happens for any functions and variables you define interactively, within the same notebook (technically, for all symbols in the default context). Definitions from other contexts, such as functions from packages, are not distributed automatically.

Lower-level functions, such as `ParallelEvaluate`, do not perform any automatic distribution of values.

Loading Packages on Remote Kernels

`ParallelNeeds["Context`"]`	evaluate `Needs["Context`"]` on all available parallel kernels

Loading packages.

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

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 with `ParallelNeeds[]`. 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 *Mathematica* package `FiniteFields`` on the master kernel and all remote kernels.

```
In[12]:= Needs["FiniteFields`"]

In[13]:= ParallelNeeds["FiniteFields`"];
```

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_] := RandomReal[{-1, 1}, {n, n}, WorkingPrecision → prec]
```

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

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

It is enough to distribute the definition of the main function `tf`. Any values it depends on will be distributed automatically.

```
In[4]:= DistributeDefinitions[tf]
Out[4]= {tf, mat, prec}
```

A Sample Run

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

```
In[5]:= ParallelMap[tf, {50, 60, 70, 80}]
Out[5]= {0.894864, 1.63675, 2.69759, 3.70744}
```

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

```
In[6]:= ParallelEvaluate[tf[70]]
Out[6]= {2.99454, 2.39164, 2.71859, 2.46563}
```

Automatic Parallelization

`Parallelize[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.

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

In[6]:= Parallelize[Map[f, {a, b, c, d, e, f}]]
Out[6]= {f[a], f[b], f[c], f[d], f[e], f[f]}
```

$$\text{In[7]:= } \texttt{Parallelize}\left[\{a, b, c, d\}.\begin{pmatrix} w1 & w2 \\ x1 & x2 \\ y1 & y2 \\ z1 & z2 \end{pmatrix}\right]$$

$$\text{Out[7]= } \{a\,w1 + b\,x1 + c\,y1 + d\,z1,\ a\,w2 + b\,x2 + c\,y2 + d\,z2\}$$

Automatically Parallelizing Existing Serial Expressions

Use Parallelize to have Mathematica decide how to distribute work across multiple kernels.

```
semiprimes =
 Times @@@ Map[Prime, RandomInteger[{10 000, 1 000 000}, {1000, 2}], {2}];
Prime[10 000]
```

```
104 729
```

```
{timing1, result} =
 AbsoluteTiming[Parallelize[Map[FactorInteger, semiprimes]]];
timing1
```

```
0.366724
```

```
{timing2, result} = AbsoluteTiming[Map[FactorInteger, semiprimes]];
timing2
```

```
0.695663
```

Automatically Parallelizing Existing Serial Expressions

There is a natural trade-off in parallelization between controlling the overhead of splitting a problem or keeping all the cores busy.

```
{timing1, result} =
 AbsoluteTiming[Parallelize[Map[FactorInteger, semiprimes],
   Method → "CoarsestGrained"]];
timing1
```

```
0.347445
```

```
{timing2, result} =
 AbsoluteTiming[Parallelize[Map[FactorInteger, semiprimes],
   Method → "FinestGrained"]];
timing2
```

```
0.855140
```

Sending a Command to Multiple Kernels

Use `ParallelEvaluate` to send commands to multiple kernels and wait for completion. Use `With` to bind locally defined variables before distribution.

```
Take[
 Flatten[ParallelEvaluate[
   RandomInteger[{-100, 100}, Ceiling[100 / $KernelCount]]]], 100]
```

```
{55, 30, -55, 5, 30, -19, -76, 97, 69, 27, -70, 3, 68, -19, -92, 75, -55, -71,
 18, 60, 17, -10, 26, -24, 70, 59, -66, -87, -64, 67, 15, -19, 52, 45, 93,
 -92, -74, -87, 62, 75, -29, -39, -90, 58, 2, 23, 86, 52, 29, -20, -69,
 -91, 66, -24, 25, 30, 21, 79, -95, -20, -76, 67, 43, 34, -62, -40, 81,
 64, -33, 83, -26, 30, 78, -38, -81, -27, 61, -39, -100, 70, 40, 68, 46,
 -54, 25, -59, -80, 33, -31, -81, 38, -25, -40, -70, 91, 8, 47, 18, 73, 93}
```

```
vars = With[{num = 1000},
  Take[
   Flatten[ParallelEvaluate[RandomInteger[{-100, 100},
      Ceiling[num / $KernelCount]]]], num]];
Length[vars]
```

```
1000
```

Implementing Task-Parallel algorithms

ParallelDo

`ParallelDo[expr, {i_max}]`
evaluates *expr* in parallel i_{max} times.

`ParallelDo[expr, {i, i_max}]`
evaluates *expr* in parallel with the variable i successively taking on the values 1 through i_{max} (in steps of 1).

`ParallelDo[expr, {i, i_min, i_max}]`
starts with $i = i_{min}$.

`ParallelDo[expr, {i, i_min, i_max, di}]`
uses steps di.

`ParallelDo[expr, {i, {i_1, i_2, ...}}]`
uses the successive values $i_1, i_2, \ldots$.

`ParallelDo[expr, {i, i_min, i_max}, {j, j_min, j_max}, ...]`
evaluates *expr* looping in parallel over different values of j, etc. for each i.

Implementing Data-Parallel algorithms

`ParallelMap` is a natural way to introduce parallelism using a functional programming style. When you have a computationally expensive function to execute of a large data set, Mathematica can execute the operations in parallel by splitting the mapping among multiple kernels.

```
Module[{data = RandomInteger[{10^40, 10^50}, 32]},
 SeedRandom[8];
 Column[{
   AbsoluteTiming[ParallelMap[PrimeOmega, data]],
   AbsoluteTiming[Map[PrimeOmega, data]] }]
]
```

```
{15.373621, {3, 7, 7, 7, 8, 5, 9, 5, 3, 2, 6, 8,
  4, 3, 3, 7, 7, 7, 3, 7, 2, 4, 12, 6, 4, 4, 3, 9, 7, 5, 7, 6}}
{23.803586, {3, 7, 7, 7, 8, 5, 9, 5, 3, 2, 6, 8,
  4, 3, 3, 7, 7, 7, 3, 7, 2, 4, 12, 6, 4, 4, 3, 9, 7, 5, 7, 6}}
```

Decomposing a Problem in Parallel Data Sets

Estimating Pi Using Monte Carlo Simulation: Serial and Parallel.

```
r = 1 / 2;
pts = RandomReal[{-r, r}, {50 000, 2}];
insidepts = Select[pts, Norm[#1] ≤ r &];
Length[insidepts] / (Length[pts] * r^2) // N
```

```
3.13656
```

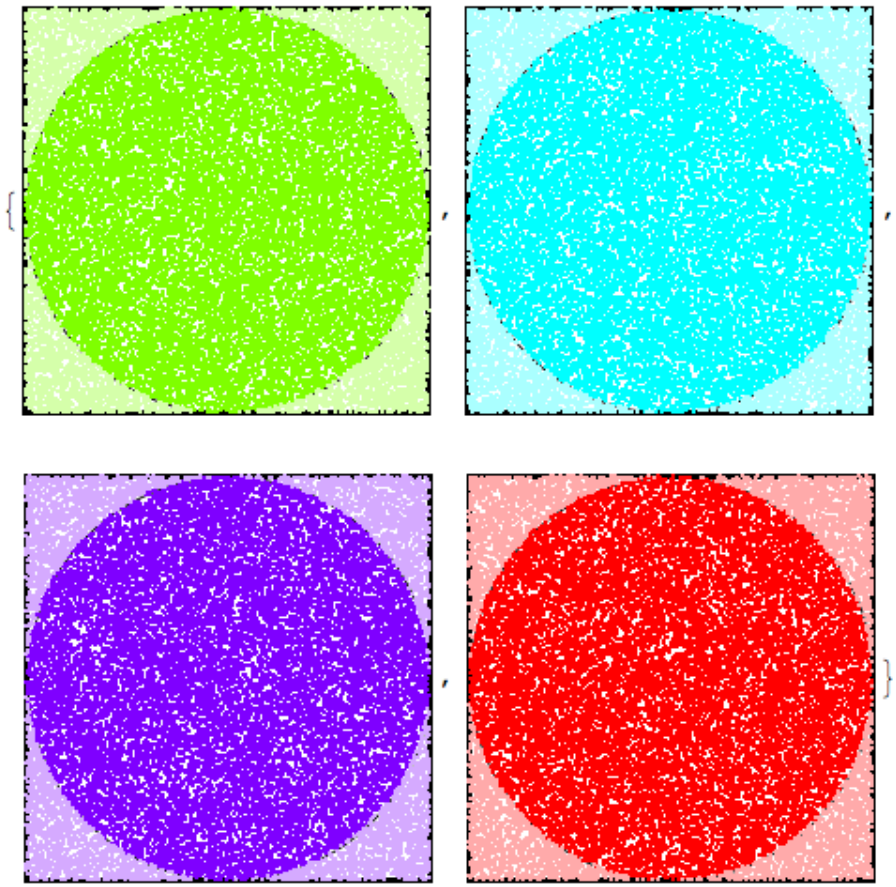
```
DistributeDefinitions[r];
pts = ParallelTable[RandomReal[{-r, r}, {50 000, 2}],
    {Length[Kernels[]]}];
insidepts = ParallelMap[Select[#, (Norm[#] < r &)] &, pts];
DistributeDefinitions[pts, insidepts];
N[
 Mean[ParallelTable[Length[insidepts[[i]]] /
     (Length[pts[[i]]] * r^2), {i, Length[Kernels[]]}]]]
```

```
3.13924
```

Decomposing a Problem in Parallel Data Sets

Graphic of work distribution among the 4 kernels.

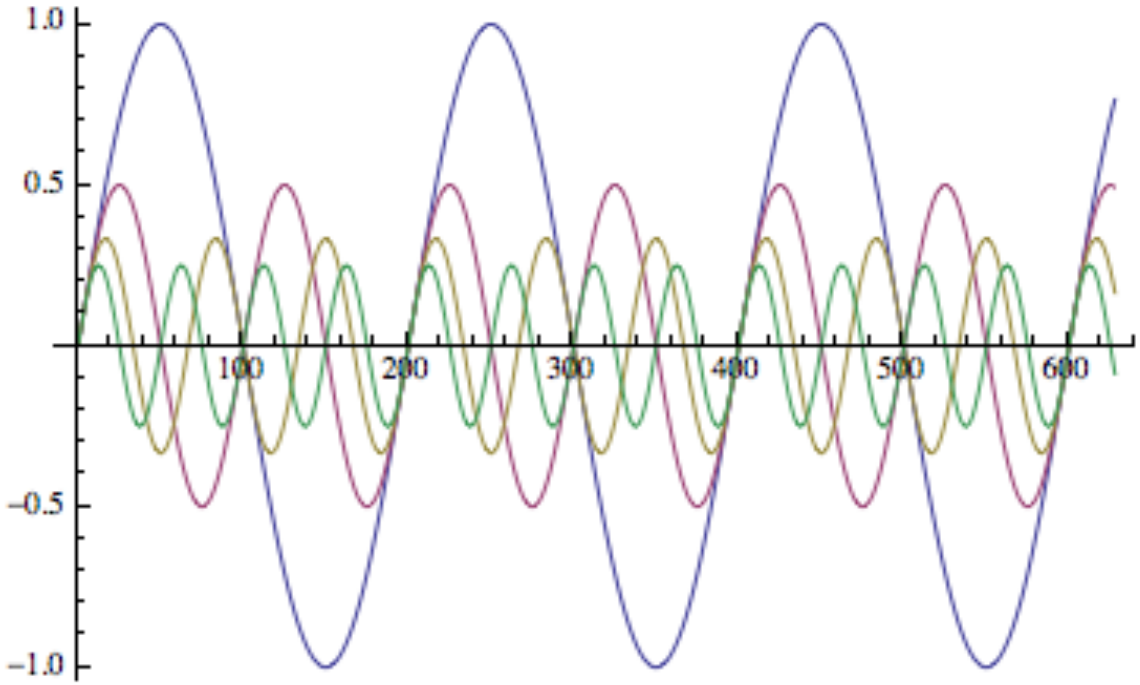
```
With[{len = Length[pts]},
 MapIndexed[
  Graphics[{EdgeForm[Thick], White, Rectangle[{-r, -r}], Black, Circle[{0, 0}, r],
    Hue[#2[[1]] / len], PointSize[Small], Point[#1[[1]]], Lighter[Hue[#2[[1]] / len, .5]],
    Point[Complement[#1[[1]], #1[[2]]]]}, ImageSize → 225] &,
  Transpose[{pts, insidepts}]]]
```



Decomposing a Problem in Parallel Data Sets

You can generate multiple data sets in parallel, then plot or process them further.

```
ListLinePlot[ParallelTable[Sin[n * Pi * x] / n, {n, 1, 4}, {x, 0, 2 * Pi, 0.01}]]
```



Choosing an Appropriate Distribution Method

The parallel primitives `Parallelize`, `ParallelMap`, `ParallelTable`, `ParallelDo`, `ParallelSum`, and `ParallelCombine` support an option called `Method`

It allows you to specify the granularity of the subdivisions used to distribute the computation across kernels.

Use `Method` $\rightarrow$ `"FinestGrained"` when the completion time of each atomic unit of computation is expected to vary widely.

Use `Method` $\rightarrow$ `"CoarseGrained"` when the completion time of each atomic unit of computation is expected to be uniform.

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.

Mathematica provides functions that implement virtual shared memory for these remote kernels.

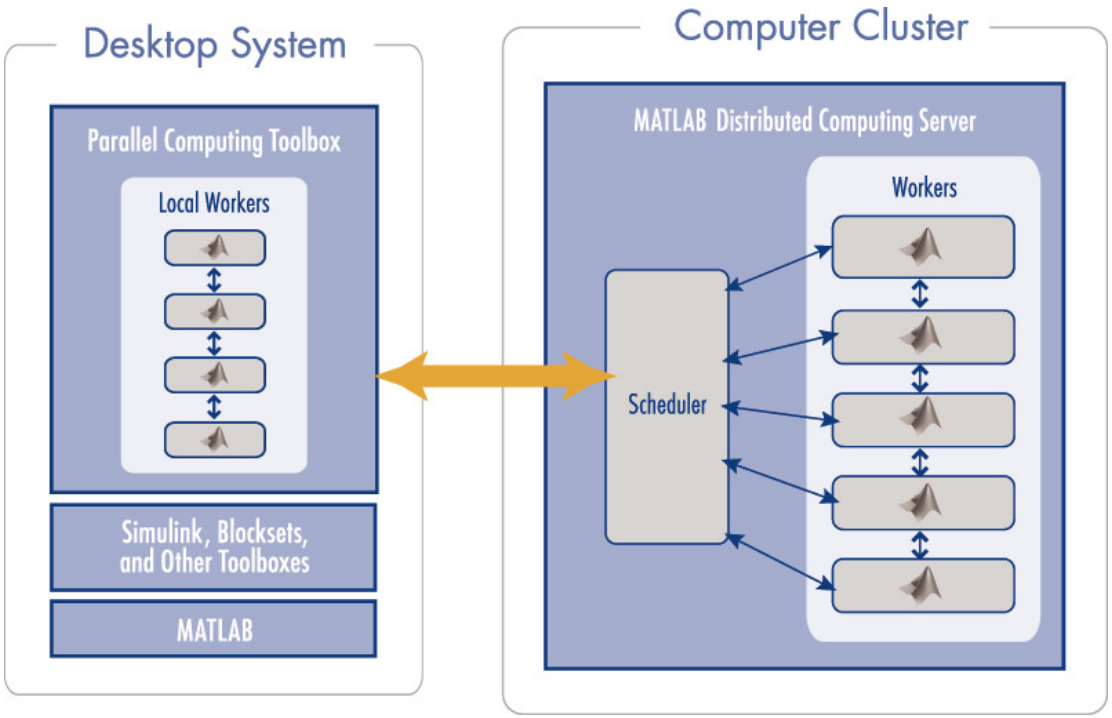
The drawback of a shared variable is that every access for read or write requires communication over the network, so it is slower than access to a local unshared variable.

Declaring shared variables and functions.

`SetSharedVariable[`s_1`,` s_2`, ...]`	declares the symbols s_i as shared variables
`SetSharedFunction[`f_1`,` f_2`, ...]`	declares the symbols f_i as shared functions or data types

Mathematica Parallel Workflow

The toolbox enables application prototyping on the desktop with up to 16 local workers (left), and with the Mathematica Cluster Integration package(right), applications can be scaled to multiple computers on a cluster (subsitute Mathematica for Matlab in figure below).



Scaling Up from the Desktop

Mathematica's parallel computing provides the ability to use up to 16 local kernels on a multicore or multiprocessor computer.

When used together with Cluster Integration package, you can scale up your application to use any number of kernels running on any number of computers.

ITS Linux cluster allows for 128 kernels.

Alternatively, you can run up to 16 kernels on a single multi-core compute node of the cluster.

Running Mathematica on Cluster Front-end Node

Mathematica Parallel Computing jobs can be submitted to the ITC Linux cluster by first logging onto the cluster front-end node fir.itc.virginia.edu using the NX client.

Start up Mathematica from a Linux desktop terminal window.

Parallel Mathematica jobs can be submitted from with the Mathematica notebook interface as well as using PBS command files and the example scripts show how to setup and submit the jobs

Documentation: Submitting Mathematica Parallel Jobs

Example Mathematica Scripts

The files in this folder are organized into two groups:

Mathematica Script M-files that can be run interactively from the Mathematica notebook to illustrate various Parallel Computing constructs.

`math_script1.m` Example script M-file using that estimates Pi with a Monte Carlo method using local kernels.

`math_script2.m` Example script M-file using that estimates Pi with a Monte Carlo method using remote kernels on the cluster compute nodes.

Job Submission script files that submit a job through PBS Pro to the cluster using the Mathematica M-files from above.

`math_submit1.sh` PBS script to submit a Mathematica parallel job to run on multiple cores of a single cluster compute node.

`math_submit2.m` PBS script to submit a Mathematica parallel job to run on multiple cores across multiple cluster compute nodes.

Notebook files that contain code examples.

`parallel_mathematica_ex.nb` Code examples from lecture slides.

References

1. Parallel Computing Tools User Guide
   reference.wolfram.com/mathematica/ParallelTools/tutorial/Overview.html
2. Parallel Computing: Mathematica Documentation
   reference.wolfram.com/mathematica/guide/ParallelComputing.html
3. *Mathematica Cookbook*, by Sal Mangano, O'Reilly Press.

Need further help? Email uvacse@virginia.edu.