Introduction to Parallel Programming

Benjamin S. Skrainka

University of Chicago
The Harris School of Public Policy
skrainka@uchicago.edu

May 29, 2012

Parallel Programming

Researchers routinely encounter problems which are too big to fit in a single computer because they require:

- Too much memory (Big Data)
- Too much computing power
- Complex simulation

To solve bigger problems, must split problem up:

- Perform different parts of the computation in different threads/cores
- Periodically, combine results as necessary, which requires fast communication
- Trick is to determine the best way to parallelize the problem

Example: Dynamic Programming

Dynamic programming is often well-suited to parallelization:

- Optimization step: compute the value function for each state (e.g., Chebyshev nodes) in a different thread
- Fitting step: compute current approximation to the value function based on the optimal values at the set of state values
- Repeat as necessary for value function iteration or backwards induction if problem has finite horizon

Types of Parallelization

There are two primary types of parallelization based on memory organization

- Shared Memory
  - Multiple threads/processes run on the same node/chip but on different processor cores within the node
  - Threads communicate via shared memory
  - Good for smaller jobs or mixed with MPI
- Message Passing
  - Multiple threads reside on different nodes
  - Threads communicate via some kind of fast (network) interconnect
  - Typically call MPI to manage communication
  - Good for any scale, but especially large scale computation

Parallel Programming Tools

Many parallel programming tools are available to solve big problems:

- Simple batch processing
- MPI (Message Passing Interface) : good for large problems
- OpenMP : good for small or medium-scale problems
- CUDA :
  - Good for problems which fit into the single instruction multiple data framework (SIMD) such as dynamic programming
  - Runs on commodity (cheap) graphics cards
- Languages: C/C++, FORTRAN, Python, etc.

Choose the tool which is right for your problem!

Parallel Algorithms

Think about how to decompose your problem into different tasks:

- What tasks are assigned to which threads?
- What kind of synchronization is needed?
- What granularity?
  - I.e., how much work is assigned to each task
  - Balance computation time vs. communication & scheduler overhead
- How does the algorithm scale as you increase processing power? (Hint: linear is the desired answer.)
- Choose an algorithm based on the hardware architecture of the cluster your code will run on

References

There are many books on parallel programming. Some helpful books include:

- *Parallel Programming for Multicore and Cluster Systems* by Rauber & Rünger provides a nice introduction to many methods and is available for free on SpringerLink.com
- "Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors," Aldrich, et al (2011) JEDC
- *Computer Systems: A Programmer's Perspective* by Bryant & O'Hallaron is helpful for understanding how a computer works
- *The Linux Programming Interface: A Linux and UNIX System Programming Handbook* by Kerrisk is a helpful reference for systems programming issues
- Plus specific books on OpenMP and MPI

Objectives

This talk's objectives:

- Describe hardware and software environment of modern high performance clusters
- State features and benefits of most common parallel processing tools
- Write Unix/Linux scripts to run jobs on clusters
- Design, implement, and execute embarrassingly parallel applications
- Provide overview of OpenMP and MPI

Plan for this Talk

Overview of Parallel Programming

High Performance Unix

Embarrassingly Parallel

MPI

OpenMP

Overview of Parallel Programming

Overview of High Performance Computing

Performance gains come from executing programs/algorithms more quickly. Parallelism takes several forms:

- With-in processor parallelism
- Multicore parallelism
- Multicomputers

Single Processor Parallelism

The first cut at gaining performance is to parallelize operations within a single processor core:

- Pipeline:
  - Each instruction consists of several phases (fetch, decode, execute, write-back)
  - Parallelize by:
    - Stagger execution of these phases for sequential instructions
    - Add more pipelines
- Superscalar:
  - Add more ALUs, FPUs, etc.
  - Process operations in parallel
- Benefit: continue to write sequential code with a standard compiler
- Limited by how many transistors you can pack on a chip
- Cache and DMA are also affect performance...

Multicore Parallelism

The next gains come from adding more processor cores per chip:

- Each chip contains multiple cores
- Each core is a microprocessor
- Cores communicate via:
  - Fast shared memories
  - Shared caches (in some cases)
- Hardware must synchronize memory access
- Ubiquitous in post 1990s processors

Multicomputer Parallelism

For large scale computations, use multiple, multicore processors:

- High performance clusters consist of multiple, interconnected computers
- Known as a distributed memory machine (DMM) or cluster
- Communication depends on message passing because local memory is private
- Application = local sequential programs + communication so they can work together
- Can used shared memory within a multicore chip
- Typically have a hierarchical organization:
  - Multicore processor chips organized into blades
  - Blades organized into racks
  - Racks organized into the HPC
- Many topologies exist which are tailored to different problem domains $\Rightarrow$ choose the right machine for your problem!

Parallel Programming Concepts

Some terms to know:

- *High Throughput Computing* (HTC): a system designed for long-running, sequential jobs (e.g., Condor system for scavenging cycles)
- *High Performance Computing* (HPC): a system designed for long-running, tightly-coupled parallel jobs with low latency (e.g., BlueGene/Q)
- *Synchronous*: one job cannot proceed until an earlier job has completed (E.g., finite horizon dynamic programming – must compute V^t before V^{t-1}.)
- *Asynchronous*: one job can proceed even if another has not finished
- *Barrier*: a point in the algorithm where all threads pause until all work in current iteration is finished
- A *node* is a multicore processor chip + memory which is connected to other processors, i.e., a node in the cluster's communication graph

Threads

A *thread* is an independent flow of control (program counter) with its own stack but shared memory:

- A process contains one or more threads
- Assign work to threads
- OS kernel schedules threads to run
- Threads exist in several states:
  - Executable: the thread is eligible to run
  - Running: the processor is allocated to the thread and it is executing
  - Suspended: the thread is marked ineligible to run, often because of an error
  - Waiting: the thread is unable to run until some event occurs, such as data is available or some amount of time elapses
- If a process has multiple threads, you must write reentrant or thread-safe code!

Thread States

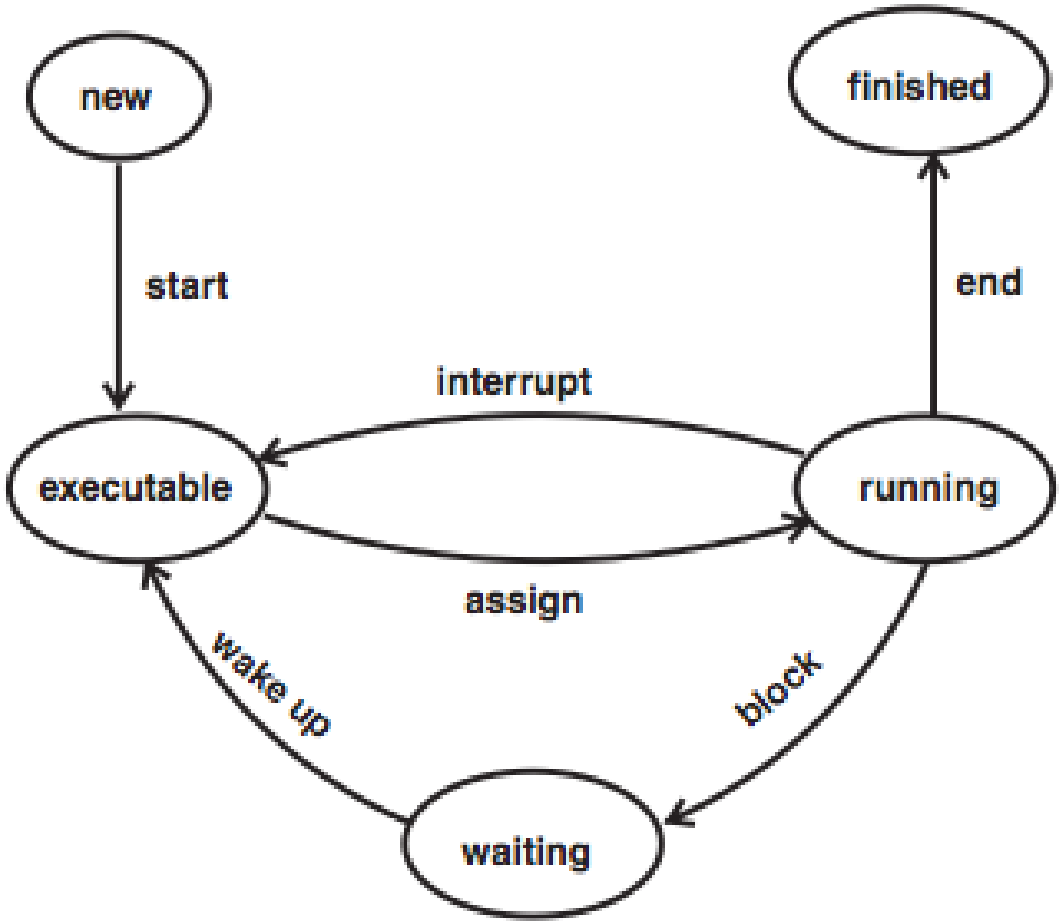


Figure from Rauber und Rünger

Communication

Parallelization depends on fast communication between the different threads:

- Different threads (usually) run on different cores
- Threads must pass information back and forth, such as current approximation to the value function
- Two main types of communication
  - Shared Memory
  - Message Passing (Network)
- Fast interconnects are one of the key expenses and bottlenecks of high performance computing systems . . . and the faster they are, the less reliable they usually are!

Shared Memory

Shared memory is the easiest and fastest communication strategy:

- Threads communicate via shared memory which is accessible to all threads
- Limited by the number of cores on the node which can access the shared memory $\Rightarrow$ shared memory is only good for smaller problems
- Easiest to access via OpenMP – can also use pthreads and Unix system calls
- Can also use OpenMP in conjunction with MPI, e.g. for Physics problems with local and global communication needs
- Because memory is shared, you must write code which is reentrant and/or thread safe.
  - Otherwise, your code will exhibit intermittent behavior (a.k.a. a race condition)
  - A *race condition* occurs when the order in which threads execute affects the outcome

Reentrancy and Thread Safety

In a high performance environment, your code may execute on a machine with shared memory:

- Multiple threads (and interrupt service routines (ISRs)) could execute the same code at (roughly) the same time
- If the code accesses a shared resource like a global or static variable, the order of execution will affect results
- Your code is *thread safe* if multiple threads can execute at the same time without clobbering the shared resource
- Your code is *reentrant* if multiple threads can execute the same code (e.g., a function call) without corrupting the shared resource
- Thus, reentrancy implies thread safety

Try to write code which is thread safe/reentrant ab initio so that you don't have to refactor it later: i.e., avoid global variables.

Message Passing

Processes running on each node are private, but they coordinate their work by passing messages to communicate data and synchronize work:

- Manager:
  - Assigns tasks to workers
  - Processes results (e.g., a fitting step)
- Workers:
  - Receive tasks to perform
  - Read inputs
  - Write outputs
- Data is passed as a message
- MPI message passing API can use several different communication channels, but typically uses fast network interconnects

Overview of Network Applications

Network communication is based on the OSI 7 Layer Model:

- Fortunately, MPI hides most of the details of network programming
- Accessed via sockets or AT&T streams APIs
- To solve communication problems, identify which layer is causing the problem

Application	←	Application
Presentation	←	Translation between host and network representation; encryption
Session	←	Manages connection between applications (processes)
Transport	←	Transportation between applications (e.g., TCP or UDP)
Internet	←	Routing between nodes
Data Link	←	Packet encoding/decoding
Physical	←	Physical transmission of data (fiber, Ethernet, wireless, etc.)

Common Parallelization Methods

Three of the most common parallelization methods are:

- Batch processing with communication via files
- MPI
- OpenMP

Batch Processing

Batch processing is the simplest approach:

- Use job manager to run jobs which can run independently for each stage
- Jobs communicate via files
- Cluster must have a shared file system or some way for different jobs to access shared files
- Swift and Condor are very good for handling this scenario
- Example: computing the finite sample properties of an estimator

MPI

The message passing interface (MPI) provides parallelization for large problems:

- MPI provides a high-level API for communication between nodes
- Very flexible: can be configured for a several different methods of communication
- MPI is tightly coupled to your problem:
  - Must call MPI functions to initialize MPI resources
  - Must marshall data
  - Must call MPI functions to send and receive data
- MPI provides C and FORTRAN APIs
- Example: dynamic programming where you compute the value function for different state values on different nodes

OpenMP

Provides parallelism within only a single processor:

- Good for easy, small to medium scale (e.g., ~4, 8, 16, or more jobs) parallelization
- Allows you to exploit full processing power of a node
  - Different threads communicate via shared memory in processor
  - Different threads usually run on different cores
- Easy to program:
  - Replace loops with compiler directives. E.g., `for` $\rightarrow$ `#for`
  - Write thread safe code
  - Compile with OpenMP's compiler flags
- Some simple configuration to control number of threads and related issues
- May be combined with MPI.

Common Application Designs

Three common application designs are:

- Embarrassingly Parallel
  - Queue up asynchronous (serial) jobs for processing
  - Jobs are independent
- Manager-Worker:
  - Manager assigns tasks to workers and processes their results/output
  - Originally called, master-slave
  - Doesn't scale well
- Asynchronous Dynamic Load Balancing (ADLB):
  - ADLB servers manage work requests
  - Application processes get work from servers

First split application into tasks, then map to processes and processors

Unix and Supercomputing

Unix is operating system of choice for supercomputing:

- Flexible and powerful scripting to automate jobs:
  - Describe jobs to run
  - Diagnose problems
  - Analyze results
  - Process data
- Used by job mangers & schedulers

Getting Access to a Bigger Computer

Obviously, you need access to a bigger machine in order to exploit parallel programming:

- Athens restricts you to three jobs and is designed for just MATLAB and Stata
- U of C's Research Computing Center (RCC) is bringing a new cluster for research computing online.
- XSEDE:
  - Nationwide facility for researchers to access large-scale computing resources
  - See the Getting Started guide to sign up
  - Most startup allocations will be enough for whatever project you want to complete (100,000 node hours!)
  - Can access via ssh or a web interface
- To transfer large data files between machines, use Globus

High Performance Unix

Introduction to High Performance Unix

Unix/Linux is the language of scientific computing. To perform large scale computing, you will need:

- Shell scripts:
  - To describe jobs
  - To write utilities to analyze data and diagnose problems
- Modules: to load environment for desired software packages
- Job manager:
  - To queue jobs for processing
  - To check jobs' status

Documentation

Unfortunately, there is no magic book to read:

- Check man pages for specific commands such as module, qsub, and qstat
- Here are the support pages for PADS:
  - Job Management
  - Job Scheduling
- In general, check the documentation for the machine you are using so you can conform to local mores

Modules

On regular Unix, configuration involves:

- Modifying startup files (`.bashrc`, `.profile`, or `.bash_profile`)
- Set `PATH` & `LD_LIBRARY_PATH`
- Set other environment variables, such as licensing information

Configuration for clusters and supercomputers use modules:

- Each module configures environment for the corresponding tool
- To use a tool (e.g., compilers, libraries, Stata, MATLAB, R, etc.):
  - Load the appropriate module for the version of the tool you want to use.
  - IT maintains the necessary modules
- Specify modules to load in a startup file:
  - On PADS, put modules in `~/.modules`
  - Other machines use `~/.bashrc`, etc.

Module Commands

You only need to know three commands to use modules:

- `module avail`: display modules which are supported
- `module list`: display which modules you have loaded
- `module load`: load a specific module:
  - `module load R/2.14.0`
  - `module load gcc #Use default version`
  - `module load gdb`
  - `module load python/2.7.1 # Choose a version`
  - `module load matlab`
  - Which modules are available will depend on the specific cluster

Example: Modules

```
$ module list
No Modulefiles Currently Loaded.
$ which R
/usr/bin/R
$ module load R/2.14.0
GotoBLAS2 version 1.13 (gnu-4.1 compiler) loaded
Netlib LAPACK library version 3.3.1 (gnu-4.1 compiler) l
R version 2.14.0 (gnu-4.1 compiler) loaded
$ module list
Currently Loaded Modulefiles:
  1) goto2/1.13     2) lapack/3.3.1   3) R/2.14.0
$ which R
/soft/R/gnu-4.1/2.14.0/bin/R
```

Job Managers & Job Schedulers

Job scheduler determines when jobs are processed whereas the job manager allows you to queue your jobs for processing. Common options include:

- PBS/Sun Grid Engine
  - Job management system for batch processing
  - Write jobs using bash commands
  - Probably the easiest option to use
- Condor
  - Popular and free
  - Has own scripting language
  - Covered at ICE
- Swift
  - Designed to run complex jobs on complex machines
  - Much redundancy built in for error recovery
  - Uses its own language to describe the commands to run and how to marshall data

Condor

Condor is a popular tool for job management and scheduling:

- Condor is an alternative to PBS
- Documentation and code are available on the Condor website.
- Condor consists of:
  - Tools to submit and manage jobs
  - A scripting language to describe how to run your job
- Can communicate between jobs via shared files or MPI
- Supports checkpoints to increase fault tolerance
- Can use a directed acyclic graph to specify complex job structures where one job depends on another
- Condor will be covered in detail at ICE

Basic PBS Commands

Only a few commands are needed to submit and manage jobs via PBS:

- `qstat`: determine the status of jobs which have been submitted
  - Use `qstat -u` *`username`* to examine only your jobs
  - Use to determine job ID and whether or not your job is running
- `qsub`: submit a job for processing
  - Syntax: `qsub` *`JobName.pbs`*
  - By default, whatever your job writes to standard output and error are captured in .o and .e files with the name of your job:

    `One.Char.T50.J100.pbs.o1402632-369`

    `Slow.BLP.char.pbs.e1398267-0`
- `qdel` *`JobID`*: remove a job from the queue
- `qdel all`: delete all of your jobs

Example: qstat

```
$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
2752386.svc               M96_1.sh         obuse           30:41:42 R extended
2752387.svc               M96_2.sh         obuse           30:41:24 R extended
2752388.svc               M96_3.sh         obuse           30:44:02 R extended
2752389.svc               M96_4.sh         obuse           30:45:23 R extended
2752390.svc               M96_5.sh         obuse           30:39:30 R extended
2752391.svc               M96_6.sh         obuse           30:37:23 R extended
2752392.svc               M96_7.sh         obuse                  0 Q fast
2752393.svc               M96_8.sh         obuse                  0 Q fast
```

Queues

Clusters have different queues for different types of jobs. The queue type will determine what resources you have access to (number of CPUs, maximum time, etc.):

- debugging
- training
- fast
- extended

What queues are supported and how jobs are scheduled depend on the cluster, so always glance at the documentation to learn local customs when working on a cluster. Otherwise, you may incur the wrath of the system administrator...

Advanced PBS Commands

A few other commands are helpful:

- `qhold` - place a hold on a job so it does not run but remains in the queue
- `qrls` - remove a hold on a job
- `showq` - show all jobs in priority order

Filthy Lucre: Billing

Most clusters are setup to bill a project for the time you use:

- On Pads, for example, use the `projects` to manage billing for usage
  - `projects`: list project you are currently billing
  - `projects --available`: determine which projects you belong to
  - `projects --set`: choose a project to bill
- Billing is cluster specific, so you will have to check the documentation for the machine you are using

PBS Job Files

Embed PBS scheduling commands in your job file:

- Makes it clear how the job should be run
- Makes your work easier to replicate

I specify the job options in the first couple lines of the file:

```
#!/bin/bash
#PBS -l nodes=1:ppn=1,walltime=24:00:00
#PBS -q long
#PBS -t 0-99
#PBS -m n
```

- `#PBS` introduces an option you can pass to `qsub`
- Subsequent lines are just bash commands
- Abstract as much of the configuration quirks into environment variables so it is easy to change your job file if your configuration changes

PBS Job File Options

Here are the options I recommend:

```
#PBS -l nodes=1:ppn=1,walltime=24:00:00
```

- `-l` specifies the resource list that the job requires.
  - Change for your setup
  - `ppn` is processors per node
  - Make sure `walltime` is long enough for your job. It is in `hh:mm:ss` format

```
#PBS -q long
```

- `-q long` specifies that the job should run in the long queue. Check which queues are available with `qstat -q`

```
#PBS -t 0-99
```

- `-t` sets the range of parameter IDs to generate for a parameter sweep

```
#PBS -m n
```

- Use `-m n` to prevent the job from sending you thousands of emails on job start, failure, and completion

Parallel PBS Job Files

Can run MPI and OpenMP jobs using PBS (or Condor or Swift):

- Requires extra options which depend on how the HPC is configured
- See local documentation
- Examples:
  - `#PBS -l mppwidth=` *`NMPI`* - specify number of MPI tasks to execute
  - `#PBS -l mppdepth=` *`mm`* - specify number of OpenMP threads per MPI task
  - `#PBS -l mppnppn=` *`NPEs`* - specify number of processing elements per node (should be $\leq$ number of MPI tasks per node)
- Then run the job using the appropriate local command such as `mpirun` or `aprun`

```
#!/bin/bash
#PBS -l nodes=1:ppn=1,walltime=12:00:00
#PBS -q long
#PBS -t 0-499
#PBS -m n

#-------------------------------------------------------------------------
# Purpose: run one BLP start at a time by
# converting the parameter sweep index to a
# start and replication.
#-------------------------------------------------------------------------
# Global definitions to describe this run
export TOPDIR=/gpfs/pads/projects/CI-SES000069
export N_STARTS_PER_JOB=10
export N_REP=100
export N_STARTS=50
export IVType=Char
export T=25
export J=100
```

```
# Ensure that GFORTRAN output occurs as written
# instead of all C/C++ followed by FORTRAN
export GFORTRAN_UNBUFFERED_ALL='y'

#-------------------------------------------------------------------------
# Regular logic : run ten estimations per job
#-------------------------------------------------------------------------
ix=${PBS_ARRAYID}
ixBegin=$(( ix * N_STARTS_PER_JOB ))

for(( ix = ixBegin ; \
  ix < ixBegin + N_STARTS_PER_JOB ; ix++ ))
do

  ixRep=$(( ix / N_STARTS ))
  ixStart=$(( ix % N_STARTS ))

  echo
```

```
  echo '>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>'
  echo '====>>>> Commencing ONE BLP Estimation '
  echo '          Replication : ' ${ixRep}
  echo '          Start       : ' ${ixStart}
  echo "          Using ${IVType} instruments"
  echo '<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<'
  echo

  inFile=${TOPDIR}/data/blp\
/arm.T${T}.J${J}/rep${ixRep}\
/config/DFS.${IVType}.config.${ixStart}.prop

  ${TOPDIR}/sbox/blpcpp/src/blpDriver -f ${inFile}

done
```

Debugging Tips

- Get a small case working
- Run PBS script as shell script – may require simple modifications
- Write Python, bash, sed, awk scripts to diagnose failures
- Reasons for failure:
  - Bug in code
  - Bug in script
  - Node failure
  - Exceeding resource limits

Why Isn't My @Q#$% Job Running?

Your job may not run even if it is queued if:

- You have run a lot of jobs recently
- Other jobs are in the queue before yours
- Someone has a reservation

In general, the scheduler's policy tries to provide some amount of fairness

Unix Scripting

PBS job scripts use standard bash commands, including:

- Environment variables to manage information and state
- Command line arguments
- Flow control (looping, if/then)
- Regular commands (ls, diff, sed, awk, grep, etc.)

Creating a Script

To create a script, put the desired commands in a text file:

- The first line of your script specifies what shell to use:

  ```
  #!/bin/bash
  ```

- Start comments with '#'
- Make standalone scripts executable using `chmod +x MyScript.sh`
  - Only needed for standalone scripts – not needed for PBS job scripts
  - Can also run a script with `source MyScript.sh`

Variables

Variables are handy for passing information

- Syntax: `VAR=VALUE` (no spaces around the '=')
- Use `export` to propagate a variable to any child process:
  - `export VAR=VALUE`
- Can perform arithmetic on numeric values using standard C++ operators (+, -, *, /, %) by placing the expression inside `$(( expression ))`:

```
$ echo $((1 + 2 ))
3
$ a=1
$ b=2
$ c=$((a+b))  # No '$' to access a & b here!
$ echo $c
3
```

- More complex operations are possible: see the bash man page

Command Line Arguments

Can pass command line arguments to a script:

- Invoke like a program with command line arguments:

```
./MyScript.sh a b c
```

- To access the arguments, use the special variables

Variable	Meaning
$#	Number of command line arguments
$0	Name of script
$*n*	n-th argument
$*	All arguments other than script name

Branching

Bash supports branching, although you must combine it with the test operator ('[' and ']'):

```
if [ $# -ne 1 ]
then
  echo "Syntax: gwhich <CMD>"
  exit -1
fi
```

- `man test` for full options
- Common tests include:

Test	Meaning
-f	true if the file exists
-eq	==
-ne	!=
-gt	>
-ge	>=
-lt	<
-le	<=

Looping

Bash also provides looping:

```
# Count completed starts
for d in output/*
do
  echo $d `ls -1 $d/optimal.X.* | wc -l `
done
```

Or, fancier loops:

```
for(( ix = ixBegin ; \
  ix < ixBegin + N_STARTS_PER_JOB ; ix++ ))
do
  ...
done
```

Embarrassingly Parallel

Embarrassingly Parallel

Embarrassingly parallel[1] problems are the easiest to compute:

- A problem is embarrassingly parallel if it can be broken down into many independent jobs
- Simple queue up the necessary jobs to run at the same time via job manager
  - Multiple identical jobs are typically run using a *parameter sweep* (see below)
  - Jobs are independent because no job depends on the output of another
- Analyze output from jobs using a separate program, if necessary

[1] The PC police tell me, we now should be more sensitive and refer to this as '*pleasingly parallel*'.

Case Study: Finite Sample Analysis

Computing the finite sample properties of an estimator is embarrassingly parallel (E.g., Skrainka (2012)):

1. Generating data for one configuration is independent of other configurations $\Rightarrow$ compute in parallel
   - Make sure you generate data so smaller datasets are subsets of larger datasets to ensure data has correct statistical properties
   - Use a seed when generating data to ensure reproducibility
   - Generate multiple replications for each configuration
2. Estimating model for one data set is independent of other estimations $\Rightarrow$ compute in parallel
3. Analyze output from individual estimations (not parallel)

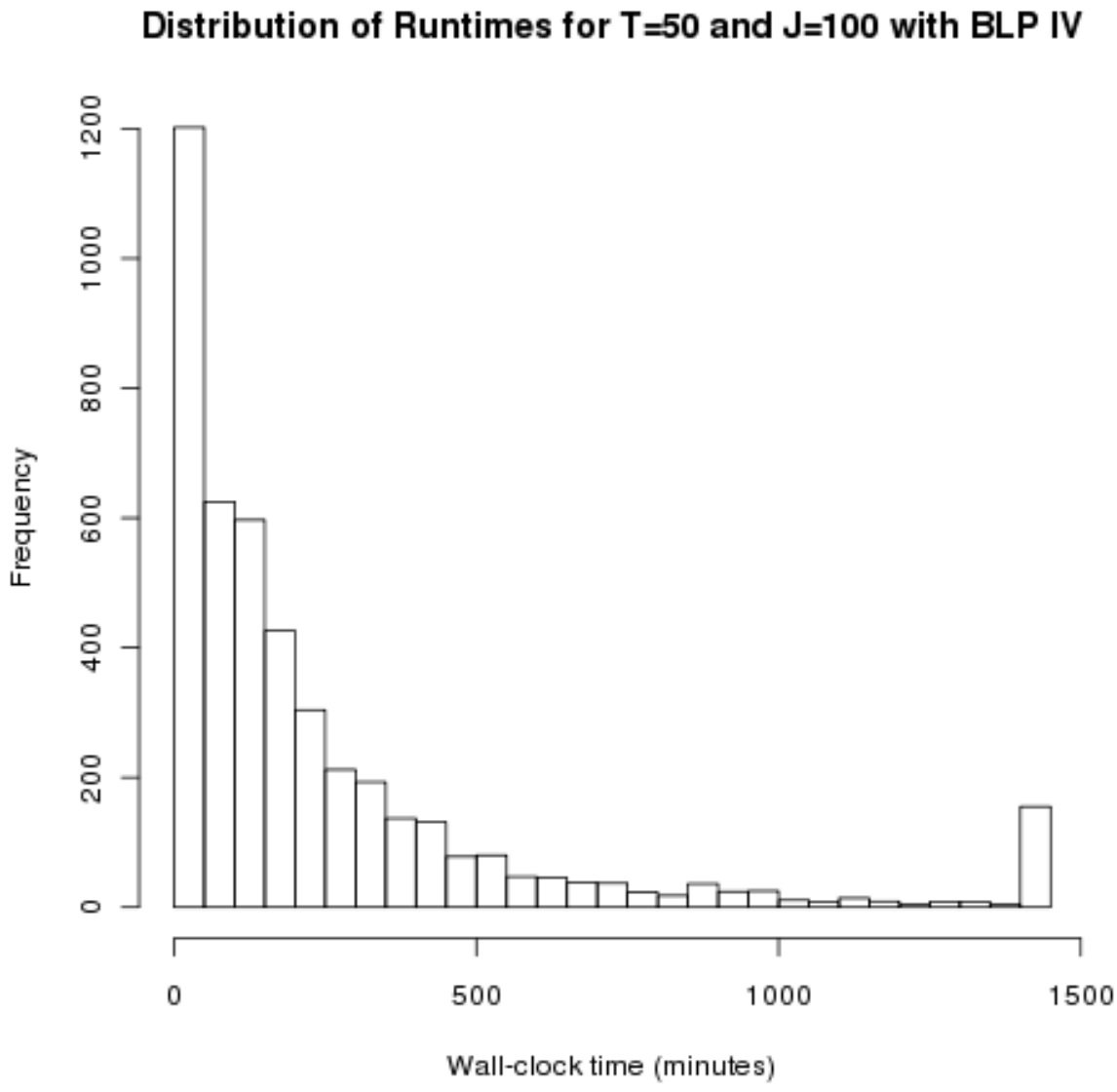
- Unix makes this easy
- Both Python and R are very good at walking directory trees
- Choose sensible naming conventions to facilitate computation and analysis

Parameter Sweeps

Parameter Sweep provides easy parallelization:

- Each job:
  - Estimates one or more replication and starting value
  - Short runs are chunked to minimize scheduler overhead
  - Independent of all other jobs
  - Identified by an index it receives from *Job Manager* $\rightarrow$ use to determine which starts to run
  - Writes results to several output files
- Job manager logs whatever the job writes to standard output and standard error to a `.o` and a `.e` file
- Impose time limit to terminate slow or runaway jobs

Example: BLP Job Times



PBS Parameter Sweeps

Most job managers make it easy to run nearly identical jobs which just differ according to an index:

- In your script specify the number of jobs you want and the range of indexes to refer to them
- For example, with PBS, use

```
#PBS -t 0-99
to create 100 jobs indexed from 0 to 99
```

- The job manager will pass the parameter sweep ID to each instance so it can determine the correct input and output to use

Using the Parameter Sweep ID

PBS passes the parameter sweep ID via the environmental variable PBS_ARRAYID

- Other systems have a similar facility
- You may need to compute other indexes from the parameter sweep ID
- For example, you may need to process an experiment with multiple replications, each of which has N_STARTS:

```
ix=${PBS_ARRAYID}
ixBegin=$(( ix * N_STARTS_PER_JOB ))
for(( ix = ixBegin ; \
  ix < ixBegin + N_STARTS_PER_JOB ; ix++ ))
do
  ixRep=$(( ix / N_STARTS ))
  ixStart=$(( ix % N_STARTS ))
  inFile=rep${ixRep}/config/config.${ixStart}.prop
  ./blpcpp -f ${inFile}
done
```

Dealing with Problems

Common problems running on a cluster include:

- Jobs which do not complete on time:
  - If several jobs are chunked together, must rerun with a smaller chunk size
  - Common because most starts are quick but there is a long thin tail of slow starts
- Jobs which crash:
  - Running on a large scale exposes flaws in your code
  - Must find bug in code and fix!
- Node failures and cluster crashes $\Rightarrow$ must figure out what to rerun
- Impossible to read $1,000,000$ output and error files!

Diagnosing Problems

When you scale an experiment up, you encounter many unexpected problems:

- Must automate verification as much as possible
- Read some log files carefully to learn what to look for
- Perform some random inspections
- Write scripts (among others) to perform common tasks:
  - Check .e and .o files
  - Compare results from original start and restart run from original optimum
  - Check residuals when computing price equilibrium
  - Check solver exit codes
  - Compute statistics about job time used for an experiment
  - Generate PBS files automatically to avoid errors
- Unix makes it easy to create tools with bash, Python, grep, sed, awk, R, etc.

MPI

MPI Overview

Use MPI for large-scale problems:

- Problem is too big to solve using a single processor chip
- Application decomposed into separate processes which run on different cores:
  - Each node has its own private memory
  - Nodes joined by fast interconnects
  - Processes pass messages to send and receive data as well as to synchronize work
- Can use OpenMP for within node communication if problem can be solved can be decomposed so some parts communicate more frequently, such as the PDEs in computational fluid dynamics (CFD)

To use MPI:

- Compile with MPI version of the compiler
- Communicate by calling MPI library functions

Standards

MPI is an open, free standard:

- MPI-1:
  - Provides basic communication
  - Synchronization is coupled with communication
- MPI-2: adds more advanced features
- Can get free versions of libraries from OpenMPI and MPICH (an Argonne product)
- Bottom line: use whatever is installed on your cluster...

An introductory tutorial is available at ANL

The Big Six

MPI provides a simple, higher-level interface for communication. Most applications only need six basic functions:

- `MPI_Init()`: initialize application to use MPI
- `MPI_Finalize()`: cleanup after using MPI
- `MPI_Comm_size()`: get number of processes in application
- `MPI_Comm_rank()`:
  - Get process's ID, known as rank (0 is initial process, often the master)
  - Use rank to assign work
- `MPI_Send()`: write data, may block
- `MPI_Recv()`: read data, may block

MPI Hello World

```
#include <stdio.h>
#include “mpi.h”

int main( int argc, const char *argv[] )
{
  int rank;
  int size;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  printf( “I am %d of %d processes.\n”, rank, size );
  MPI_Finalize();
  return 0;
}
```

Example from ANL tutorial cited above

Compiling

Compile with the local MPI compiler:

```
$ module load mpich2
$ mpicc hello.c -o hello
```

Different MPI implementations have different names for the compiler:

- Actually a wrapper script
- Invokes GNU or other compiler with MPI flags and libraries

Running

To run on a specific cluster, read the manual. Some common configurations are:

- Run from command line (if allowed):
  1. Start an MPI demon, `mpd`, to coordinate MPI communications (if necessary)
  2. Launch your application from command line:

```
$ mpiexec -n 8 myMPIApp arg1 arg2
$ mpirun -np 8 myMPIApp arg1 arg2
```

- Otherwise, use a PBS script or equivalent:

```
#!/bin/bash
#PBS -l nodes=1:ppn=4,walltime=00:05:00
#PBS -q fast
#PBS -m n
cd ${PBS_O_WORKDIR}
mpirun -np 4 ./hello
```

Output

Note: jobs run in the order they are scheduled so they may not complete in the order you expect.

```
$ mpirun -np 8 ./hello
I am 2 of 8 processes.
I am 3 of 8 processes.
I am 4 of 8 processes.
I am 5 of 8 processes.
I am 6 of 8 processes.
I am 7 of 8 processes.
I am 0 of 8 processes.
I am 1 of 8 processes.
```

OpenMP

OpenMP Extensions

OpenMP is another free, portable, open standard for parallel programming:

- Based around thread programming
- Threads live in a common address space (shared memory):
  - Must coordinate access to shared memory
  - Ensure other variables are private
  - Necessary for thread safe code. . . otherwise your code will exhibit race conditions
- Easy to use:
  - Add pragmas to:
    - Create parallel regions
    - Specify whether variables are shared or private
  - Additional functions to get information about process
- Typically cannot create as many parallel tasks as with MPI because threads must all run on one node or blade

Compiling and Running

- Check manual for options for your compiler. For GNU,

```
$ g++ -fopenmp -Wall -Wextra -pedantic -O2 \
  neo.c -o neo
$ ./neo
```

- Configure OpenMP by setting environment variables:
  - `OMP_NUM_THREADS` : number of threads to use
  - `OMP_SCHEDULE` : which scheduler to use
  - `OMP_DYNAMIC` : boolean whether to set number of threads dynamically

```
/* neo - neo-classical growth model in OpenMP */

/*
 * This code was originally written by
 * Serhiy Kozak
 *
 * Note: this code demonstrates how to
 * use OpenMP but does not use the best
 * numerical methods.  For example, a better
 * interpolation method should be used
 * to approximate the value function.
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
```

```
const int    N     = 100 ;
const int    freq  = 12 ;
const double sigma = 2 ;
const double alpha = 0.3 ;
const double pk    = 2 ;

const double A     = 1.0 / freq ;
const double rho   = 0.075 / freq ;
const double delta = 0.075 / freq ;
const double beta  = 1. / (1. + rho) ;

double k[N] ;
double V[N]  = {0.0} ;  // Value function
int    P[N]  = {0} ;    // Policy function
```

```
// Utility function
inline double u(double c)
{
  double cp = (c > 0)*c + (c <= 0)*1e-10 ;
  double res = pow(cp, 1 - sigma) / (1 - sigma) ;
  return res ;
}

// Production function
inline double f(double k)
{
  double res = A * pow(k, alpha) ;
  return res ;
}
```

```
int main()
{

  // steady state
  double kstar =
    pow( (A*alpha/(pk*(rho + delta))),
         (1/(1 - alpha)) ) ;
  double kmin = 0.9 * kstar ;
  double kmax = 1.1 * kstar ;
  int i, j ;

  for (i = 0 ; i < N; ++i)
  {
    k[i] = kmin + i*(kmax-kmin)/(N-1) ;
    V[i] = 0 ;
    P[i] = 0 ;
  }
  //e = ones(N, 1) ;
```

```
const double conv = 1e-12 ;

// Value function iteration
double dist = 1 ;
int it = 0 ;
while( dist > conv )
{
  double vmax = 0.0 ;
```

```
#pragma omp parallel default(none) \
  shared(V,P,k,vmax) private(i,j)
{
    // Compute value function for each
    // of N values of state k
#pragma omp for nowait
    for( j = 0 ; j < N; ++j )
    {
      double omax = -1e12 ;
      int    oidx = -1 ;

      int i0 = 0 ;
```

```
// max Bellman eq. for current state k[j]
for (i = i0 ; i < N; ++i )
{
  double v =
    u(f(k[j]) + pk*(1-delta)*k[j] - pk*k[i])
    + beta*V[i] ;
  if (v >= omax)
  {
    omax = v ;
    oidx = i ;
  }
  else
  {
    //break ;
  }
}
```

```
        if( fabs((omax-V[j])/(1+fabs(V[j]))) > vmax )
          vmax = fabs((omax-V[j])/(1+fabs(V[j]))) ;

        V[j] = omax ;
        P[j] = oidx ;
      }   // End of omp for loop
}   // end of omp parallel block

    ++it ;
    dist = vmax ;
  }

  printf("Converged in %d iteration ; %f\n",
    it, dist);

  return 0 ;
}
```