Table of Contents

1. Misc Information
2. Parallel Performance
   3. Parallel Performance Metrics
      - Speedup and Efficiency of Parallel Code
      - Effect of Overhead
      - Amdahl’s Law
      - Thomas timing examples - Parallel Model
      - Code Performance: Serial Looptest.f90
3. Resource Management
4. Next Time
Today:
- Parallel Performance
- First parallel code
- Resource Monitors (queuing systems)

Next HW will be posted over the weekend

Quiz 1: tentative date is 09/23/14 (Tuesday)
Total Parallel Program Time

- The total parallel program run time is a function of a large number of variables: number of processing elements (PEs); communication; hardware (cpu, memory, software, network), and the program being run (algorithm, problem size, # Tasks, complexity, data distribution); parallel libraries:

$$T = \mathcal{F}(PEs, N, Tasks, I/O, Communication, \ldots)$$

- The execution time required to run a problem of size N on processor $i$, is a function of the time spent in different parts of the program (computation, communication, I/O, idle):

$$T^i = T^i_{\text{comp}} + T^i_{\text{comm}} + T^i_{\text{io}} + T^i_{\text{idle}}$$

- The total time is the sum of the times over all processes averaged over the number of the processors:

$$T = \frac{1}{p} \left( \sum_{i=0}^{p-1} T_{\text{comp}} + \sum_{i=0}^{p-1} T_{\text{comm}} + \sum_{i=0}^{p-1} T_{\text{io}} + \sum_{i=0}^{p-1} T_{\text{idle}} \right)$$
Parallel Performance Metrics
Speedup and Efficiency of Parallel Code

Speedup

- Refers to how much faster the parallel algorithm runs than a corresponding sequential algorithm.
- \( T_{ser} \) == time between when first processor begins execution to when the last processor completes its tasks.
- Speedup is defined to be:
  \[
  S_p = \frac{T_{ser}}{T_{par}}
  \]
  
  Where:
  - \( p \) == number of cores (processors, PE's)
  - \( T_{ser} \) == serial execution time
  - \( T_{par} \) == parallel execution time
- Linear speedup, or ideal speedup, is obtained when \( S_p = p \):
  \[
  T_{par} = T_{ser} / p
  \]
Efficiency

- Estimation of how well the processors are used to solve the problem vs. effort is wasted in communication and synchronization.

- \( T_{\text{elap}} \) is the time between when the first processor begins execution to when the last processor completes its tasks.

\[
E = \frac{S}{p} = \left( \frac{T_{\text{serial}}}{T_{\text{parallel}}} \right) p = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}
\]

- Where:
  - \( p \) is the number of cores (processors, PE’s)
  - \( T_{\text{ser}} \) is the serial execution time
  - \( T_{\text{par}} \) is the parallel execution time

- Efficiency is typically between zero and one.
Parallel Performance Metrics
Speedup and Efficiency of Parallel Code

Test data from Pacheco IPP (Ch 2) demonstrating effect of problem size on the speedup and efficiency.

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half</td>
<td>S</td>
<td>1.00</td>
<td>1.90</td>
<td>3.10</td>
<td>4.80</td>
<td>6.20</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1.00</td>
<td>0.95</td>
<td>0.78</td>
<td>0.60</td>
<td>0.39</td>
</tr>
<tr>
<td>Original</td>
<td>S</td>
<td>1.00</td>
<td>1.90</td>
<td>3.60</td>
<td>6.50</td>
<td>10.80</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1.00</td>
<td>0.95</td>
<td>0.90</td>
<td>0.81</td>
<td>0.68</td>
</tr>
<tr>
<td>Double</td>
<td>S</td>
<td>1.00</td>
<td>1.90</td>
<td>3.90</td>
<td>10.80</td>
<td>14.20</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>1.00</td>
<td>0.95</td>
<td>0.98</td>
<td>0.94</td>
<td>0.89</td>
</tr>
</tbody>
</table>
Parallel Performance Metrics

Speedup and Efficiency of Parallel Code

![Speedup vs. Number of Processes Graph](image-url)
Parallel Performance Metrics
Speedup and Efficiency of Parallel Code

![Efficiency vs. Number of Processes](chart.png)
Run Times vs. Number of Processes

- **Half**
- **Original**
- **Double**
Effect of overhead

\[ T_{\text{parallel}} = \frac{T_{\text{serial}}}{p} + T_{\text{overhead}} \]
Amdahl’s Law

- Unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited — regardless of the number of cores available.
Example

- We can parallelize 90% of a serial program.
- Parallelization is “perfect” regardless of the number of cores \( p \) we use.
- \( T_{\text{serial}} = 20 \) seconds
- Runtime of parallelizable part is

\[ 0.9 \times T_{\text{serial}} / p = 18 / p \]
Example (cont.)

- Runtime of “unparallelizable” part is
  \[ 0.1 \times T_{\text{serial}} = 2 \]

- Overall parallel run-time is
  \[ T_{\text{parallel}} = 0.9 \times T_{\text{serial}} / p + 0.1 \times T_{\text{serial}} = 18 / p + 2 \]
Example (cont.)

- Speed up

\[ S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}} / p + 0.1 \times T_{\text{serial}}} = \frac{20}{18 / p + 2} \]
Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*. 
### Customized Timings: Parallel Framework

#### TABLE II.
**Time spent in main sections of the serial and parallel models (16 and 32 processor elements)**

<table>
<thead>
<tr>
<th>Section</th>
<th>Serial</th>
<th>16 Processors</th>
<th>32 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tinit</td>
<td>48571</td>
<td>24285</td>
<td>16190</td>
</tr>
<tr>
<td>Tloop</td>
<td>59451</td>
<td>29725</td>
<td>19817</td>
</tr>
<tr>
<td>Twall</td>
<td>108083</td>
<td>54041</td>
<td>36027</td>
</tr>
</tbody>
</table>

#### TABLE III.
**Time spent in different submodules executed during the main iteration loop**

<table>
<thead>
<tr>
<th>Section</th>
<th>Serial</th>
<th>16 Processors</th>
<th>32 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tpres</td>
<td>31619</td>
<td>15810</td>
<td>10540</td>
</tr>
<tr>
<td>Tfio</td>
<td>17961</td>
<td>8981</td>
<td>5987</td>
</tr>
<tr>
<td>Tsigs</td>
<td>3026</td>
<td>1513</td>
<td>1009</td>
</tr>
<tr>
<td>TVelw</td>
<td>1736</td>
<td>868</td>
<td>579</td>
</tr>
<tr>
<td>TVelu</td>
<td>1726</td>
<td>863</td>
<td>575</td>
</tr>
<tr>
<td>TVelv</td>
<td>1716</td>
<td>858</td>
<td>572</td>
</tr>
<tr>
<td>TbcondP</td>
<td>448</td>
<td>224</td>
<td>150</td>
</tr>
<tr>
<td>TvelcorV</td>
<td>120</td>
<td>61</td>
<td>40</td>
</tr>
<tr>
<td>TvelcorW</td>
<td>110</td>
<td>55</td>
<td>36</td>
</tr>
<tr>
<td>TvelcorU</td>
<td>109</td>
<td>54</td>
<td>367</td>
</tr>
<tr>
<td>TbcondW</td>
<td>22</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>TbcondU</td>
<td>22</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>TvelcorV</td>
<td>20</td>
<td>11</td>
<td>67</td>
</tr>
<tr>
<td>Tloop (meas)</td>
<td>58635</td>
<td>29317</td>
<td>19545</td>
</tr>
</tbody>
</table>
Runtime vs Number of Iterations as a Function of the Number of Processors (linear-linear)

- **SerRef**
- **Par4**
- **Par8**
- **Par16**
- **Par32**

**Axes:**
- Y-axis: Elapsed (seconds)
- X-axis: Number of Iterations
Runtime vs Number of Iterations as a Function of the Number of Processors (log-log)
Runtime vs Number of Iterations as a Function of the Number of Processors (log-linear)
Parallel Performance Metrics

Thomas timing examples - Parallel Model

![Graph showing Speedup (Ts/Tp) vs Processor Elements]

- **Speedup**
- **Ideal**
Efficiency = (Ts/Tp)/P = Ts/(P*Tp)
/* hello.c by James Otto, 1/31/11
   --- for running serial processes
   on a cluster... see batch.hello */
#include <stdio.h>
#include <unistd.h>
int main(void)
{
  char cptr[100];
  gethostname(cptr,100);
  printf("hello, world from %s\n", cptr);
  return 0;
}

COMPILE & RUN SERIAL PGM

[tuckoo]$ mpicc -o hello hello.c
[mthomas@tuckoo ex.2014]$ mpirun -np 5 ./hello
hello, world from tuckoo
hello, world from tuckoo
hello, world from tuckoo
hello, world from tuckoo
hello, world from tuckoo

COMPILE & RUN PARALLEL PGM

[tuckoo]$ mpicc -o hello_mpi hello_mpi.c
[tuckoo]$ mpirun -np 5 ./hello_mpi
Hello Processor: rank: 0, nprocs: 5
Hello Processor: rank: 1, nprocs: 5
Hello Processor: rank: 3, nprocs: 5
Hello Processor: rank: 4, nprocs: 5
Hello Processor: rank: 2, nprocs: 5

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mpi.h"

int main (int argc, char* argv[])
{
  int rank, nprocs, ierr, i, error=0;
  MPI_Status status;

  ierr = MPI_Init(&argc, &argv);
  if (ierr != MPI_SUCCESS) {
    printf("MPI initialization error\n");
  }

  // processing element ID
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // ID of communicator connecting PE's
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  printf("Hello Processor: rank: %d, nprocs: %d\n", rank, nprocs);

  MPI_Finalize();
  return 0;
}
program loop_test
implicit none
integer, parameter :: max=10000
integer :: i, j
double precision :: tws, twe, ts, te
double precision :: a(max,max), x(max), y(max)
call cpu_time(tws)
!
initialize arrays
a=0.0; x=0.0; y=0.0
do i=1,max
  x(i) = i
  y(i) = max-i
  do j=1,max
    a(i,j) = 10*j + i
  enddo
endo
!
compute loop 1
call cpu_time(ts)
call loop1(y,max)
call cpu_time(te)
print *,"Telap: loop 1 = ", (te - ts)
!
compute loop 2
ts=0.0; te=0.0;
call cpu_time(ts)
call loop2(y,max)
call cpu_time(te)
print *,"Telap: loop 2 = ", (te - ts)
!
compute loop 3
ts=0.0; te=0.0;
call cpu_time(ts)
call loop3(y,max)
call cpu_time(te)
print *,"Telap: loop 3 = ", (te - ts)
call cpu_time(twe)
print *,"Wallclock Time: ", (twe - tws)

contains
subroutine loop1(yloc,maxloc)
integer :: maxloc
double precision :: yloc(maxloc)
do i=1,maxloc
  do j=1,maxloc
    yloc(i) = a(i,j) * x(j)
  enddo
endo
end subroutine loop1

subroutine loop2(yloc,maxloc)
integer :: maxloc
double precision :: yloc(maxloc)
do j=1,maxloc
  do i=1,maxloc
    yloc(i) = a(i,j) * x(j)
  enddo
endo
end subroutine loop2

subroutine loop3(yloc,maxloc)
integer :: maxloc
double precision :: yloc(maxloc)
do i=1,maxloc
  do j=1,maxloc
    yloc(i) = a(i,j) * sqrt(x(j))
  enddo
endo
end subroutine loop3
end program loop_test
Parallel Performance Metrics
Code Performance: Serial Looptst.f90

Code Example: run job from command line

```bash
[mthomas@tuckoo]$ cat makefile

MAKE FILE

MPIF90 = mpif90
MPICC = mpicc
CC = gcc
all: looptst looptstp
looptst: looptst.f90
$(MPIF90) -o looptst looptst.f90

looptstp: looptst.f90
$(MPIF90) -p -o looptstp looptst.f90

clean:
rm -rf *.o looptst looptst-mpi
```

```bash
[mthomas@tuckoo]$ ./looptstp

Testing FORTRAN loops (column major):
Telap: loop 1 = 960.8539 msec
Telap: loop 2 = 580.9109 msec
Telap: loop 3 = 1744.7349 msec
Wallclock Time: = 5861.1099 msec
```

```bash
[mthomas@tuckoo]$ gprof looptstp gmon.out

Flattened profile:

<table>
<thead>
<tr>
<th>Name</th>
<th>cumulative self</th>
<th>self time</th>
<th>s/call</th>
<th>total time</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN___</td>
<td>37.58</td>
<td>37.58</td>
<td>1</td>
<td>3.67</td>
<td></td>
<td>MAIN___</td>
</tr>
<tr>
<td>frame_dummy</td>
<td>27.04</td>
<td>2.40</td>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>frame_dummy</td>
</tr>
<tr>
<td>loop1.1529</td>
<td>23.25</td>
<td>3.26</td>
<td>1</td>
<td>0.86</td>
<td>0.86</td>
<td>loop1.1529</td>
</tr>
<tr>
<td>loop2.1523</td>
<td>10.95</td>
<td>3.67</td>
<td>1</td>
<td>0.41</td>
<td>0.41</td>
<td>loop2.1523</td>
</tr>
</tbody>
</table>
```
Parallel Performance Metrics

Code Performance: Serial Looptest.f90

Run Serial Job In Queue

SUBMIT SERIAL JOB TO QUEUE

[mthomas@tuckoo looptst]$ cat batch.looptstp
#!/bin/sh
#PBS −V
#PBS −l nodes=2:ppn=4:core4
#PBS −N looptstp
#PBS −j oe
#PBS −q batch
cd $PBS_O.WORKDIR
NCORES='wc −w < $PBS_NODEFILE'
echo "looptstp−test using $NCORES cores..."
mpirun −np 4 −hostfile $PBS_NODEFILE
    −nooversubscribe ./looptstp

[mthomas@tuckoo looptst]$ !qsub
qsub batch.mpi−looptstp
16478.tuckoo.sdsu.edu

OUTPUT (asynchronous)

Telap: loop 1 = 0.8428719999
Telap: loop 1 = 0.8308729
Telap: loop 1 = 0.8308739
Telap: loop 1 = 0.8328739
Telap: loop 1 = 0.8428719

Telap: loop 2 = 0.4549309
Telap: loop 2 = 0.4559310
Telap: loop 2 = 0.4559310
Telap: loop 2 = 0.4559310

Telap: loop 3 = 0.9918490
Telap: loop 3 = 0.9918490
Telap: loop 3 = 0.9918490
Telap: loop 3 = 1.0078469

Wallclock Time: = 5.02523599
Wallclock Time: = 5.0262349
Wallclock Time: = 5.02823599
Wallclock Time: = 5.049231

Note: no gain by using multiple PE’s −− > no MPI calls in code
Add MPI Calls

program looptest
!
implicit none
include "mpif.h"
integer, parameter :: max=10000
double precision, allocatable :: a(:,:), x(:,), y(:)
double precision :: tws,twe, ts, te
integer :: i,j, rank, nprocs, ierr, token
integer :: status(MPI_STATUS_SIZE)

! initialize arrays
do i=1,max
   x(i) = i; y(i) = max-i
   do j=1,max
      a(i,j) = 10*j + i
   enddo
enddo
Parallel Performance Metrics

Code Performance: Serial Looptest.f90

Run MPI Job In Queue

```bash
[mthomas@tuckoolooptstp]$ qsub
qsub batch.mpi-looptstp
16478.tuckoo.sdsu.edu

[OUTPUT (asynchronous)]

[mthomas@tuckoolooptstp]$ cat mpi-looptstp-o16485
mpi-looptstp-test using 8 cores...
LocaMAX: 2500
LocaMAX: 2500
LocaMAX: 2500
LocaMAX: 2500
PE[ 0]: Telap, loop 1= 0.07698800
PE[ 1]: Telap, loop 1= 0.07698800
PE[ 2]: Telap, loop 1= 0.07598900
PE[ 3]: Telap, loop 1= 0.07698800
PE[ 0]: Telap, loop 2= 0.03799400
PE[ 1]: Telap, loop 2= 0.03799500
PE[ 2]: Telap, loop 2= 0.03799400
PE[ 3]: Telap, loop 2= 0.03699500
PE[ 0]: Telap, loop 3= 0.07898800
PE[ 1]: Telap, loop 3= 0.07998700
PE[ 2]: Telap, loop 3= 0.07998800
PE[ 3]: Telap, loop 3= 0.08098700
PE[ 0]: Telap, Twall= 0.33594800
PE[ 1]: Telap, Twall= 0.33794800
PE[ 2]: Telap, Twall= 0.33294900
PE[ 3]: Telap, Twall= 0.34294600
```

Note: $T_{\text{wall}}$ reduced from 5+ seconds to 0.3
mpi-looptst RunTime (Twall)

Note: Ideal runtime computed using $T_{\text{ideal}} = \frac{T_{\text{ser}}}{p}$
mpi-looptst: Speedup

![Graph showing speedup vs. number of processes (Nprocs)]
Resource Management on HPC Systems

- **Distributed Resource Management Systems (D-RMS)**
  - assigns computational resources to users
  - Platform Load Sharing Facility (LSF), Portable Batch Systems (PBS), Sun Grid Engine (SGE), IBM Load Lever, Condor.

- **Job Management subsystem (JMS)**
  - Interface between users and RMS
  - Job submission, queuing, running, accounting, ...

- **Physical Resource Manager**
  - control the use of hardware (CPU cycles, memory, swap, disk, network)
  - report and account status and usage of resources.

- **Scheduler and Queue Systems**
  - applies scheduling algorithms to organize and optimized jobs
  - enforces policies for usage and charging, etc.

Source: Yan & Chapman, Comparative Study of Distributed Resource Management Systems
Resource Management

Source: Yan & Chapman, Comparative Study of Distributed Resource Management Systems
- **Job Server**
  - commands/daemons communicate with Server
  - batch job services: receiving/creating, running, modifying, protecting against system crashes
- **Job Executor (MOM)**
  - places copy of job into execution
  - user login session
- **Job Scheduler**
  - control when/where jobs run
  - communicate with MOM
- **Command Interface**
  - command line or GUI, user, operator, manager
  - submit, monitor, modify, delete
TORQUE/PBS

- TORQUE RM: controls batch jobs and distributed computing resources
- based on the original PBS project
## PBS: Environment Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBS_O_QUEUE</td>
<td>Queue job was submitted to.</td>
</tr>
<tr>
<td>PBS_QUEUE</td>
<td>Queue job is running in (same as PBS_O_QUEUE)</td>
</tr>
<tr>
<td>PBS_JOBID</td>
<td>Job ID – used by qstat, showstart, and dque.</td>
</tr>
<tr>
<td>PBS_JOBNAME</td>
<td>Name of the job. This can be set using -N in script.</td>
</tr>
<tr>
<td>PBS_NODEFILE</td>
<td>Name of file that contains list of HOSTS for job</td>
</tr>
<tr>
<td>PBS_ARRAYID</td>
<td>Array ID numbers for jobs submitted with the -t flag.</td>
</tr>
<tr>
<td>PBS_VNODENUM</td>
<td>Determine the task number of each processor. See (2).</td>
</tr>
<tr>
<td>PBS_O_PATH</td>
<td>Original PBS path. Used with pbsdsh.</td>
</tr>
</tbody>
</table>

(1) PBS User Guide Example:  
https://wiki.hpcc.msu.edu/display/hpccdocs/Advanced+Scripting+Using+PBS+Environment+Variables  
(2) http://www.ep.ph.bham.ac.uk/general/support/torquepbsdsh.html
PBS: Common Commands

- **Job control**
  - qsub submit a job
  - qdel delete a batch job
  - qsig send a signal to a batch job
  - qhold hold a batch job
  - qrerun rerun a batch job
  - qmove move a batch job to another queue

- **Job monitoring**
  - qstat show the status of batch jobs
  - qselect select a specific subset of jobs

- **Node status**
  - pbsnodes list the status and attributes of all nodes in the cluster.
  - [http://linuxinfo.physik.hu-berlin.de/pbs.html](http://linuxinfo.physik.hu-berlin.de/pbs.html)

- **Others**
  - qalter alter a batch job
  - qmsg send a message to a batch job

Example of User Guide: [http://rcc.its.psu.edu/user_guides/system_utilities/pbs/](http://rcc.its.psu.edu/user_guides/system_utilities/pbs/)
PBS: Batch Script Example

#!/bin/sh
# this example batch script requests many processors...
# for more info on requesting specific nodes see "man pbs\_resources"
#PBS -V
#PBS -l nodes=2:ppn=2:core4+3:ppn=2:core8
#PBS -N many
#PBS -j oe
#PBS -q batch
cd $PBS\_O\_WORKDIR
echo ------------------------------------------------------
echo -n 'Job is running on node '; cat $PBS\_NODEFILE
echo ------------------------------------------------------
echo PBS: qsub is running on $PBS\_O\_HOST
echo PBS: originating queue is $PBS\_O\_QUEUE
echo PBS: executing queue is $PBS\_QUEUE
echo PBS: working directory is $PBS\_O\_WORKDIR
echo PBS: execution mode is $PBS\_ENVIRONMENT
echo PBS: job identifier is $PBS\_JOBID
echo PBS: job name is $PBS\_JOBNAME
echo PBS: node file is $PBS\_NODEFILE
echo PBS: current home directory is $PBS\_O\_HOME
echo PBS: PATH = $PBS\_O\_PATH
echo ------------------------------------------------------
NCORES='wc -w < $PBS\_NODEFILE'
echo "many-test using $NCORES cores..."
mpirun -np 10 -hostfile $PBS\_NODEFILE ./many
### use nooversubscribe to run 1 job/pe or core
###mpirun -np 10 -hostfile $PBS\_NODEFILE --nooversubscribe ./many

Example of advanced batch script: http://beowulf.pppl.gov/batchscript.txt
int main (int argc, char* argv[])  
{
    int rank, num_nodes, ierr, other;
    int tkn;
    char cptr[100];
    MPI_Status status;
    gethostname(cptr,100);
    ierr = MPI_Init(&argc, &argv);
    if (ierr != MPI_SUCCESS) {
        printf("MPI initialization error\n");
        return 0;
    }
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_nodes);
    if (num_nodes != 2) {
        if (rank == ROOT) {
            printf("program error: runs on 2 processes!\n");
        }
        MPI_Finalize();
        return 0;
    }
    if (rank == 0) { /* get things started */
        token = 0;
        other = OTHER;
        token = 0; other = OTHER;
        MPI_Send(&tkn,1,MPI_INT,other,0,MPI_COMM_WORLD);
        MPI_Recv(&tkn,1,MPI_INT,other,0,MPI_COMM_WORLD,
                 &status);
        printf("process %d: (%s) received the token...
", rank, cptr);
        if (token == 1) {
            printf("process %d: token= %d, matches (things
                      look ok).\n", rank, tkn);
        } else {
            printf("process %d: ERR -- val of token (%d)\n                        does not match expected (%d)!\n", rank, token, 1);
        }
    }
    else {
        other = ROOT;
        MPI_Recv(&token, 1, MPI_INT, other, 0,
                  MPI_COMM_WORLD, &status);
        token++;
        MPI_Send(&token, 1, MPI_INT, other, 0,
                 MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
- Copy examples

[mthomas@tuckoo ]$ cp -r /examples/ ex.2014
[mthomas@tuckoo pardev]$ cd ex.2014

- make

[mthomas@tuckoo ex.2014]$ make two
mpicc -o two two.c

- BATCHSCRIPT: batch.twoF

[mthomas@tuckoo hw1]$ cat batch.twoF
#!/bin/sh
# for more info on requesting specific
# nodes see "man pbs_resources"
#PBS -V
#PBS -l nodes=2:ppn=1:core4
#PBS -N twoF
#PBS -j oe
#PBS -q batch

echo NODEFILE: $PBS_NODEFILE
cd $PBS_O_WORKDIR
NCORES=`wc -w < $PBS_NODEFILE`

echo "twoF test using $NCORES cores..."
mpirun -np 2 -host node6,node7 ./two

### use nooversubscribe to run 1 job/pe or core
###mpirun -np 4 -hostfile $PBS_NODEFILE --nooversubscribe ./looptstp

- RUN JOB

[mthomas@tuckoo ex.2014]$ qsub batch.two
16442.tuckoo.sdsu.edu

- OUTPUT

[mthomas@tuckoo ex.2014]$ cat two.o16442
NODEFILE: /var/spool/torque/aux//16442.tuckoo.sdsu.edu
twoF test using 8 cores...
process 0 of 2 (on node6)
process 1 of 2 (on node7)
success (process 0)
Common PBS Commands

- **node or PE**: CPU computer, with multiple (typically 4 or 8) processor cores.
- **pbsnodes -a**: nd out which nodes are available, and how the nodes are configured.
- **qstat or qstat -n**: used to see which jobs have been submitted and show up in the queue.
- **qsub batch.name**: used to submit a PBS batch script to the queue.
- **qdel job-id**: used to delete a job from the queue based on the job-id returned by qsub.
Next class: 09/16/14
Topic: Resource Queue