Parallel Paradigms

Dirk Colbry, Research Specialist
Institute for Cyber-Enabled Research

Scientific Programming

Inputs (Data)
Computation
Outputs (Results)
Single Thread Jobs

One CPU can only run one thing at a time. (sort of)

Goal: Use more than one CPU at a time to get more work done

Pleasantly Parallel
Loosely Coupled

Tightly Coupled
Agenda

- Introduction to High Performance Computing
- Job Schedulers
- Example Computational Research Problems
- Running Code in Parallel

1957 MISTIC Mainframe

- MSU’s first mainframe
- Hand built by grad students
  - Dick Reid
  - Glen Keeney
After MISTIC

- 1957 MISTIC
- 1963-1973 CDC 3600
- 1967 Computer Science Department
- 1968 CDC 6500
- 1971 MERIT
- 1978 Cyber 750
- 2004 HPCC
- 2009 ICER

2004 MSU HPCC

- Provide a level of performance beyond what you could get and reasonably maintain as a small group
- Provide a variety of technology, hardware and software, that would allow for innovation not easily found
Hardware:
- High speed network interconnect
  - MPI jobs
  - High Speed Parallel Scratch Space
- General Purpose Graphics Cluster
  - Common OS Image
    - RHEL 6.3
    - Compile once
    - Run anywhere
- Large Capacity “FAT” Nodes
  - Up to 2TB of RAM
  - Up to 64 cores

HPC Systems
- Large Memory Nodes (up to 2TB!)
- GPU Accelerated cluster (K20, M1060)
- PHI Accelerated cluster (5110p)
- Over 540 nodes, 10000 computing cores
- Access to high throughput condor cluster
- 363TB high speed parallel scratch file space
- 50GB replicated file spaces
- Access to large open-source software stack and specialized bioinformatics VMs
Commodity Cluster

- Most computers at HPCC fall into this category:
  - Racks of commodity Nodes
  - Connected with network
  - Uses a scheduler to run jobs.
Commodity Cluster

- Cores
- Processors / Sockets
- Nodes
- Chassis
- Rack
- High Speed Network

Blade System (Cluster 14)

- General purpose base nodes
  - 20 cores, 64 gb
- 256gb nodes
- 256gb and 2xK20 GPU nodes
- 256gb and 2xPhi Card nodes
Agenda

- Introduction to High Performance Computing
- Job Schedulers
- Example Computational Research Problems
- Running Code in Parallel

Resource Manager and scheduler

Moab → Torque

Not First In First Out!!
Schedulers vs Resource Managers

- Scheduler (Moab)
  - Tracks and assigns
    - Memory
    - CPUs
    - Disk space
    - Software Licenses
    - Power / environment
    - Network

- Resource Manager (PBS/Torque)
  - Hold jobs for execution
  - Put the jobs on the nodes
  - Monitor the jobs and nodes

Submission Script

1. List of required resources
2. All command line instructions needed to run the computation
Typical Submission Script

- Define Shell
- Resource Requests
- Shell Commands
- Special Environment Variables

Submitting a job

- qsub –arguments <Submission Script>
  - Returns the job ID. Typically looks like the following:
    - 5945571.cmgr01

- Time to job completion

Queue Run Time
Scheduling Priorities

- Jobs that use more resources get higher priority (because these are hard to schedule)
- Smaller jobs are backfilled to fit in the holes created by the bigger jobs
- Eligible jobs acquire more priority as they sit in the queue
- Jobs can be in three basic states: Blocked, eligible or running

Agenda

- Introduction to High Performance Computing
- Job Schedulers
- Example Computational Research Problems
- Running Code in Parallel
What problems are we solving?

- Boundary Simulations
- Data Analysis
- Search

Boundary Simulations

- Typically System of PDE (Partial Differential equations)
  - Fluid dynamics
  - Finite element analysis
  - Molecular dynamics
  - Weather
  - Etc.
- Mathematically equivalent to inverse of a matrix
Data Analysis

- Computer vision tasks
- Some Bioinformatics
- Astrophysics
- Etc.

Video Provided by Dr. Fred Dyer

Search

- Genome sequencing
- Analytics
- Optimization
- Etc.

Evolution of an artificial organism that can move and forage for food, Dr. Nicolas Chaumont
Agenda

- Introduction to High Performance Computing
- Job Schedulers
- Example Computational Research Problems
- Running Code in Parallel

Pleasantly Parallel

- Lots-o-Computers
- Same thing different data
- No Communication
Example

- Folder full of input files:
  1. in  5. in  9. in  13. in  17. in
  2. in  6. in 10. in  14. in  18. in
  3. in  7. in 11. in  15. in  19. in
  4. in  8. in 12. in  16. in

- Want folder full of output files:
  1. out  5. out  9. out  13. out  17. out
  2. out  6. out 10. out  14. out  18. out
  3. out  7. out 11. out  15. out  19. out
  4. out  8. out 12. out  16. out

- Command Syntax:
  - ./myprogram inputfile > outputfile

Simple Job Array

#!/bin/bash -l
#PBS -l walltime=00:05:00,mem=2gb
#PBS -l nodes=1:ppn=1,feature=gbe
#PBS -t 1-200

cd ${PBS_O_WORKDIR}

./myprogram ${PBS_ARRAYID}.in > ${PBS_ARRAYID}.out

qstat -f ${PBS_JOBID}
Condor
High Throughput Computing

- Job submission system
- Runs like a screen saver
- Steals CPU Cycles

Communication

- Shared Memory
- Shared Network
- Distributed Network
- Dedicated Accelerators
- Hybrid Systems
Shared Memory Communication

- Cores on a processor share the same memory
- OpenMP
- Fat nodes
  - 64 cores
  - 2TB of memory

Intel10

- 8 cores
- 24 GB memory

Figure Generated using hwloc
amdo9

- 32 cores
- 256 GB memory

We have nodes with 64 cores and 2TB of memory

Figure Generated using hwloc
OpenMP

- Common Shared Memory parallelization
- C/C++/FORTRAN
- Single program runs in many cores
- Really easy to pick loops that are parallel and split them into multi threads
- Minor modifications to code that can be written not to affect single cpu version of code.

OpenMP is easy

```c
#include <omp.h>
...

#pragma omp parallel for
for (i=0;i<100;++i) {
    A(I) = A(I) + B
}
...
```
Compile OpenMP Jobs

- Use compiler option openmpi.
  - fopenmp
- Example:

  gcc -fopenmp mycode.cc -o mycode

Running OpenMP code

```
export OMP_NUM_THREADS=2
./myCode
```
OpenMP Job Script

```bash
#!/bin/bash -login
#PBS -l walltime=00:05:00,mem=7gb
#PBS -l nodes=1:ppn=8,feature=gbe

cd ${PBS_O_WORKDIR}
export OMP_NUM_THREADS=8

./myOMPprogram

qstat -f ${PBS_JOBID}
```

Limits of Shared Memory

- Maximum 64-100 cores
- Even with a lot of cores doesn’t always scale well

- How do we scale even bigger?
Network parallelization

- High Speed Network
- What does the programming look like?

MPI

- Message Passing Interface (MPI)
- C/FORTRAN library that allows programs to pass “messages” between computers over the internet.
- Works best with a high speed network such as 10gigE or Infiniband
MPI program (1 of 4)

/* Needed for printf'ing */
#include <stdio.h>
#include <stdlib.h>

/* Get the MPI header file */
#include <mpi.h>

/* Max number of nodes to test */
#define max_nodes 264

/* Largest hostname string hostnames */
#define str_length 50

MPI program (2 of 4)

int main(int argc, char **argv)
{
    /* Declare variables */
    int proc, rank, size, namelen;
    int ids[max_nodes];
    char hostname[str_length][max_nodes];
    char p_name[str_length];

    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(p_name,&namelen);
MPI program (3 of 4)

```c
if (rank==0) {
    printf("Hello From: %s I am the receiving processor \n%d of \n", p_name, rank+1, size);
    for (proc=1; proc<size; proc++) {
        MPI_Recv(hostname[0][proc], str_length, MPI_INT, proc, 1, MPI_COMM_WORLD, &status);
        MPI_Recv(&ids[proc], str_length, MPI_INT, proc, 2, MPI_COMM_WORLD, &status);
        printf("Hello From: %-20s I am processor %d of \n", hostname[0][proc], ids[proc]+1, size);
    }
}
```

MPI program (4 of 4)

```c
} else { // NOT Rank 0
    srand(rank);
    int t = rand()%10+1;
    sleep(t);
    MPI_Send(&p_name, str_length, \n        MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Send(&rank, str_length, \n        MPI_INT, 0, 2, MPI_COMM_WORLD);
}
MPI_Finalize();
return(0);
```
Compile MPI Jobs

• To compile an mpi program you need to use the mpi compiler wrappers:
  – mpicc
  – mpif90

Using MPI

• MPI programs are run using the “mpirun” command:
  mpirun –np 10 –hostfile ./hosts ./myprogram
  – Number of processor cores
  – Hostfile:
    • Ipaddress
    • Computer names

• User needs to be able to remotely connect to each computer and run the program.
MPI Job Script

```bash
#!/bin/bash –login
#PBS –l walltime=00:05:00,mem=7gb
#PBS –l nodes=100:ppn=1

cd ${PBS_O_WORKDIR}

mpirun ./.myOMPprogram

qstat -f ${PBS_JOBID}
```

Distributed Network Parallelization

- Map-Reduce (HADOOP)
- Fault tolerant
- Does not require high speed network
- Scales very well.
- Not all problems map well to map-reduce
Dedicated Accelerators

- Small shared memory/network systems.
- HPC on a card
  - GPGPU (CUDA)
  - Phi Cards
  - FPGA

GPUs

- Cards used to render graphics on a computer
- Hundreds of cores
- Not very smart cores
- But, if you can make your research look like graphics rendering you may be able to run really fast!
GPU vs CPU

CPU and GPU working together
Running on the GPU

• Program Starts on the CPU
  – Copy data to GPU (slow-ish)
  – Run kernel threads on GPU (very fast)
  – Copy results back to CPU (slow-ish)

• There are a lot of clever ways to fully utilize both the GPU and CPU.

Pros and Cons

• Benefits
  – Lots of processing cores.
  – Works with the CPU as a co-processor
  – Very fast local memory bandwidth
  – Large online community of developers

• Drawbacks
  – Can be difficult to program.
  – Memory Transfers between GPU and CPU are costly (time).
  – Cores typically run the same code.
  – Errors are not detected (on older cards)
  – Double precision calculations are slow (On older cards)
CUDA program (1 of 5)

```c++
#include "cuda.h"
#include <iostream>

using namespace std;

void printGrid(float an_array[16][16]) {
  for (int i = 0; i < 16; i++){
    for (int j = 0; j < 16; j++) {
      cout << an_array[i][j];
    }
    cout << endl;
  }
}
```

CUDA program (2 of 5)

```c++
__global__ void theKernel(float * our_array)
{
  // This is array flattening,
  // (Array Width * Y Index + X Index)
  our_array[(gridDim.x * blockDim.x) * \ 
    (blockIdx.y * blockDim.y + threadIdx.y) + \ 
    (blockIdx.x * blockDim.x + threadIdx.x)] = \ "
  = 5;
}
```
CUDA program (3 of 5)

```c
int main()
{
    float our_array[16][16];

    for (int i = 0; i < 16; i++) {
        for (int j = 0; j < 16; j++) {
            our_array[i][j] = 0;
        }
    }
}
```

CUDA program (4 of 5)

```c
//STEP 1: ALLOCATE
float * our_array_d;
int size = sizeof(float)*256;
cudaMalloc((void **) &our_array_d, size);

//STEP 2: TRANSFER
cudaMemcpy(our_array_d, our_array, size, cudaMemcpyHostToDevice);
```
CUDA program (5 of 5)

```c
//STEP 3: SET UP
dim3 blockSize(8,8,1);
dim3 gridSize(2,2,1);

//STEP 4: RUN
theKernel<<<gridSize, blockSize>>>(our_array_d);

//STEP 5: TRANSFER
printGrid(our_array);
cudaMemcpy(our_array, our_array_d, size, cudaMemcpyDeviceToHost);
cout << "--------------------" << endl;
printGrid(our_array);
}
```

Compile CUDA Jobs

- Just like MPI, to compile a CUDA program, you need to use the CUDA compiler wrappers:
  - `nvcc simple.cu -o simple_cuda`
GPGPU Job Script

```bash
#!/bin/bash -login
#PBS -l walltime=00:05:00,mem=7gb
#PBS nodes=1:ppn=1,gres=gpu:1,feature=gpgpu

cd ${PBS_O_WORKDIR}
./myOMPprogram
qstat -f ${PBS_JOBID}
```

Intel Xeon Phi

- Cross between CPU and GPU
- About 60 Pentium I cores
  - Less cores than GPU
  - Easier to use than GPU
    - OpenMP
    - MPI
- Very new
  - January 2013
Which approach is the best?

• Depends on what you are doing?
• Depends on how much communication you need.
• Depends on what hardware you have.
• Depends on how much time you have.

QUESTIONS?