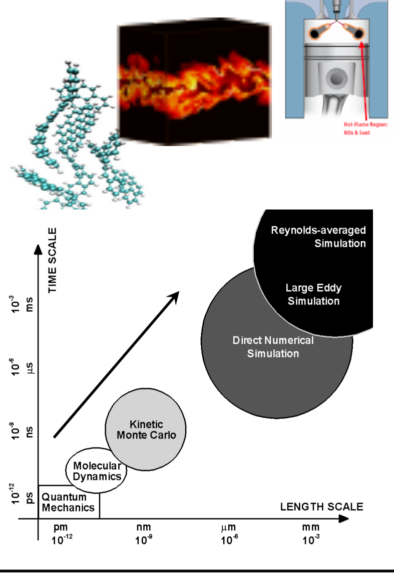
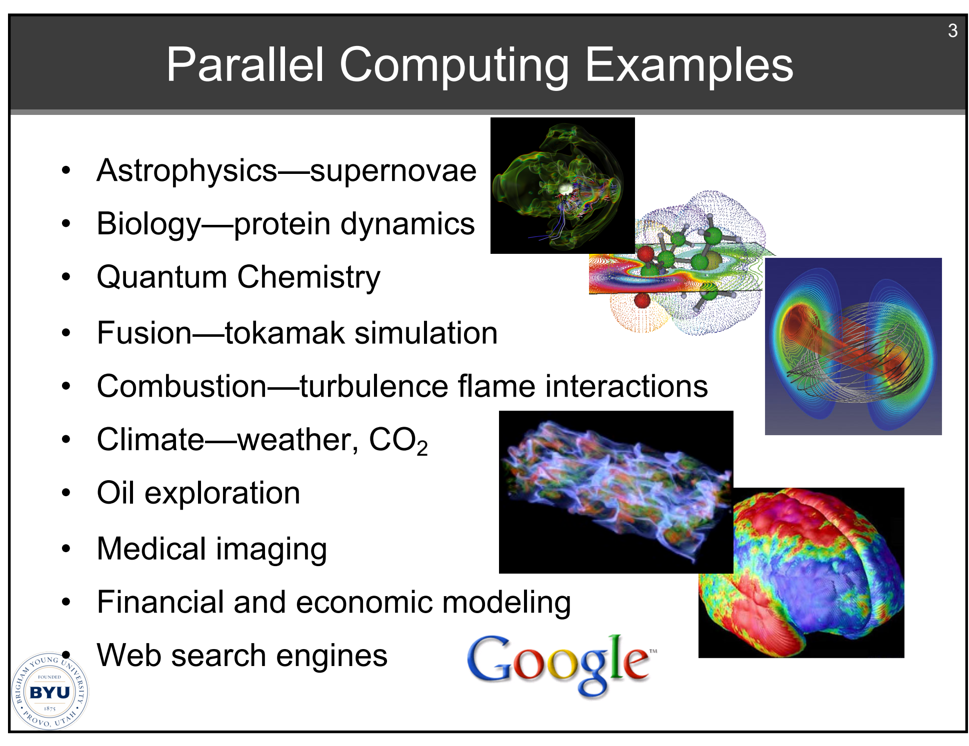
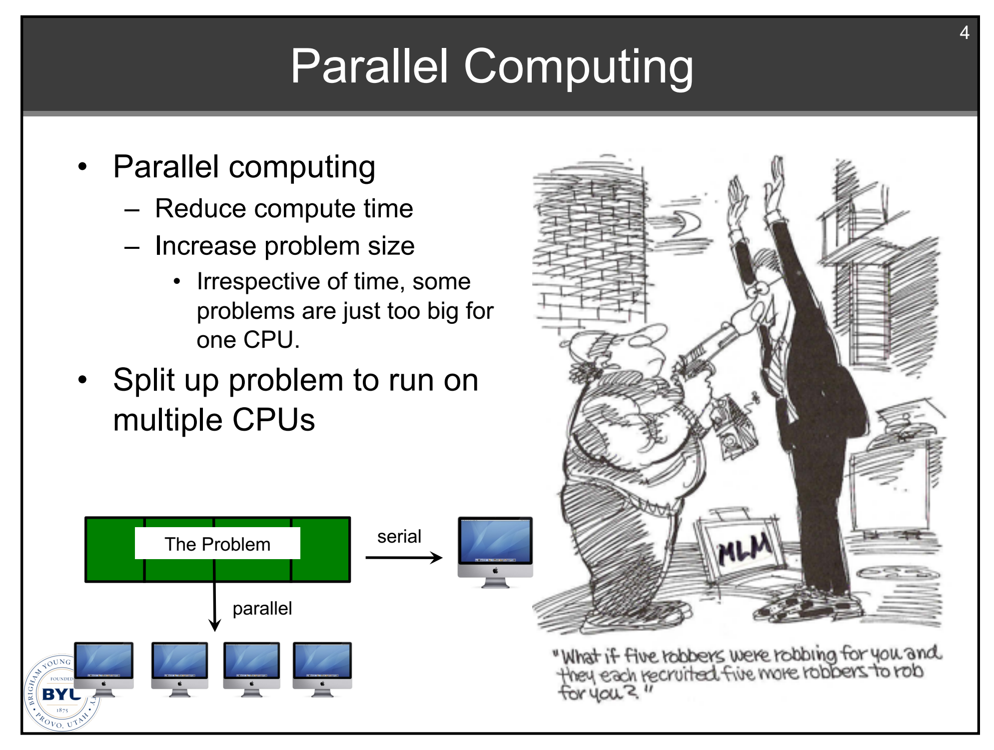
Introduction to parallel computing

Introduction

- As computers get faster, problems get more complex and computationally expensive.
- Most of engineering involves solving mathematical models of physical systems—this means computing, either on paper, or by machine!
- Increase number of grid points
- Increase number of dimensions
  - x, y, z, time, state space, etc.
- Increase problem complexity
  - Add more chemical species
  - Resolve more timescales
  - Run in larger, more realistic domains
  - Do more realizations.
  - Relax simplifying assumptions
- Require faster turnaround time
  - How long are you willing to wait: 1 day to 2 weeks.
  - Often, real-time data processing is desired.




3
Parallel Computing Examples
• Astrophysics—supernovae
• Biology—protein dynamics
• Quantum Chemistry
• Fusion—tokamak simulation
• Combustion—turbulence flame interactions
• Climate—weather, CO2
• Oil exploration
• Medical imaging
• Financial and economic modeling
• Web search engines
Google
BYU


4
Parallel Computing
• Parallel computing
– Reduce compute time
– Increase problem size
• Irrespective of time, some problems are just too big for one CPU.
• Split up problem to run on multiple CPUs
The Problem
serial
parallel
MLM
"What if five robbers were robbing for you and they each recruited five more robbers to rob for you?"
BYU

Classification—Flynn's Taxononmy

- **SISD**
  - The usual workstation,
  - *Nonparallel*

- **SIMD**
  - All processors execute same instruction at the same time, but on different data streams.
  - In many workstations: GPUs

- **MISD**
  - Several processors work on same data.
  - *Uncommon*
  - Cryptography, data filtering

- **MIMD**
  - Processors run independent code on independent data.
  - Most common parallel category
  - Often write one code, but each processor runs own version, with code exectution depending on processor ID.

See top500.org for current supercomputer info.

Hardware

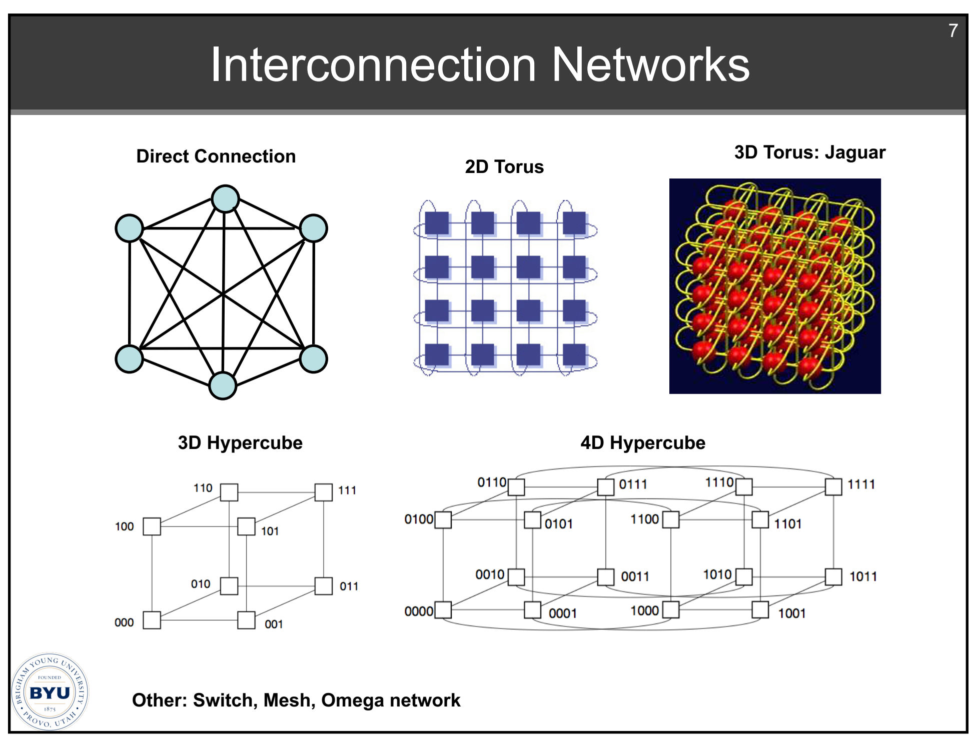
Distributed Memory

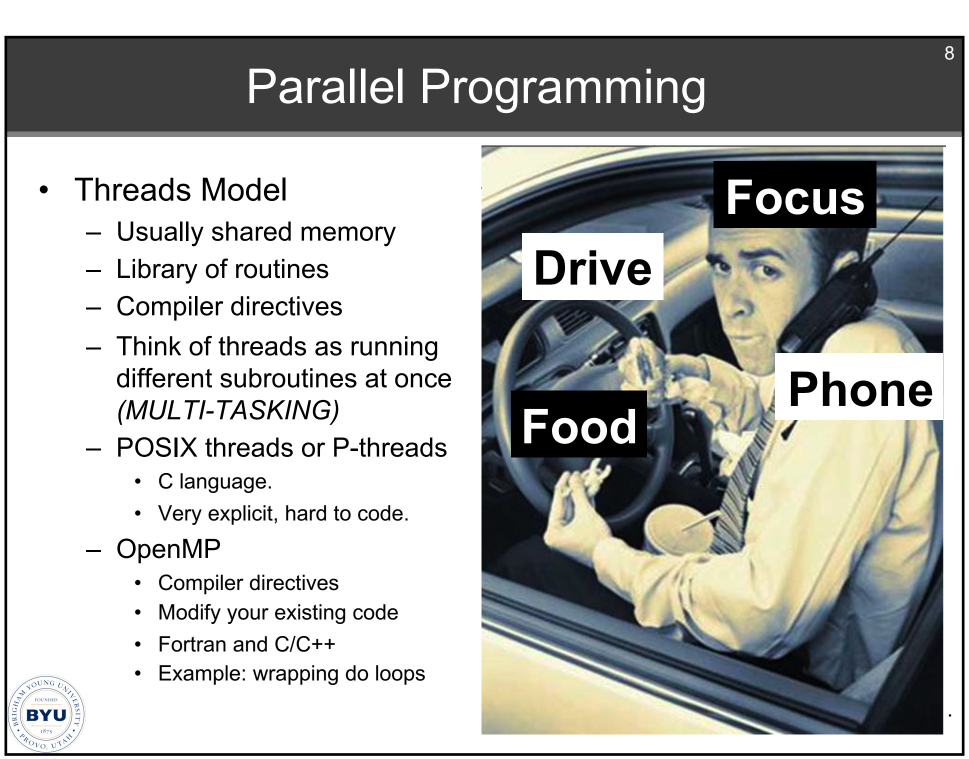
Shared Memory

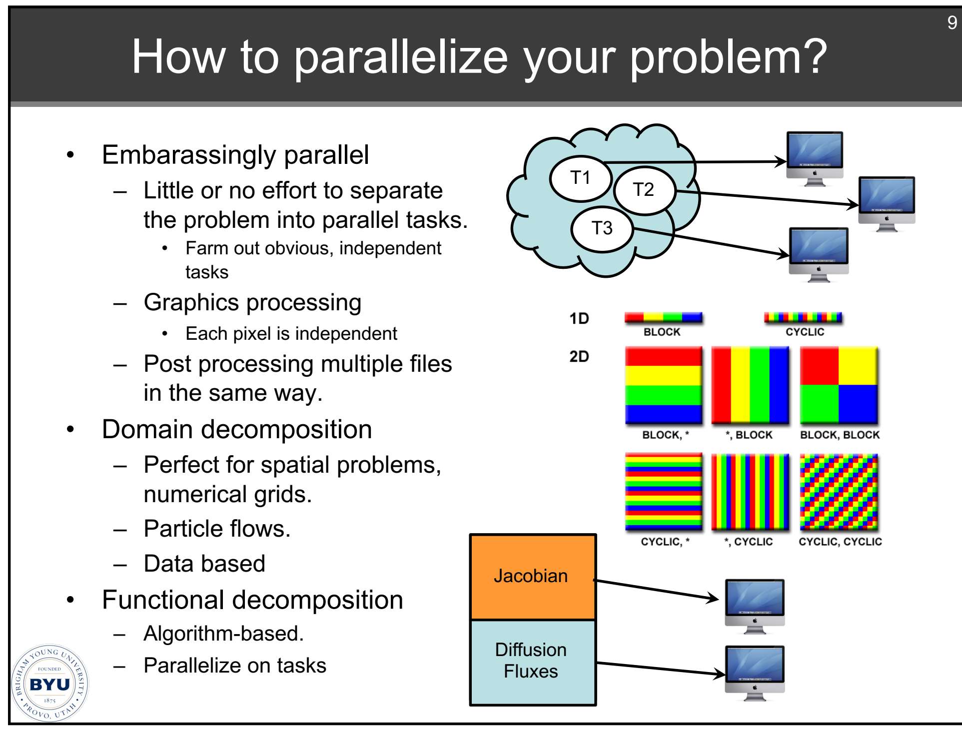
intel intel intel intel

intel intel intel intel

- Two main machine types: shared memory and distributed memory
- Shared: each processor has direct access to all memory.
  - Fast data transfers, simpler programming
  - Limited number of processors (low scalability)
  - Program with ***POSIX threads***, ***openMP***
- Distributed: each processor has own memory.
  - Highly scalable
  - More difficult to program.
  - Program with message passing: ***MPI***


7
Interconnection Networks
Direct Connection
2D Torus
3D Torus: Jaguar
3D Hypercube
110
111
100
101
010
011
000
001
4D Hypercube
0110
0111
1110
1111
0100
0101
1100
1101
0010
0011
1010
1011
0000
0001
1000
1001
Other: Switch, Mesh, Omega network
BYU

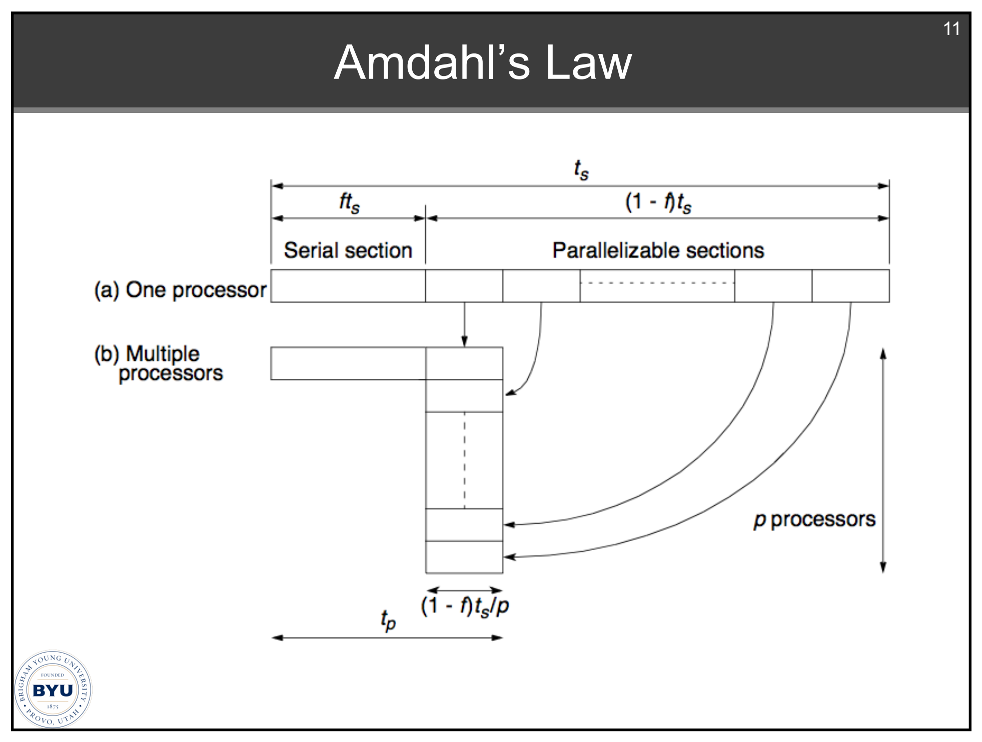

8
Parallel Programming
• Threads Model
– Usually shared memory
– Library of routines
– Compiler directives
– Think of threads as running different subroutines at once (MULTI-TASKING)
– POSIX threads or P-threads
• C language.
• Very explicit, hard to code.
– OpenMP
• Compiler directives
• Modify your existing code
• Fortran and C/C++
• Example: wrapping do loops
Focus
Drive
Phone
Food
BYU

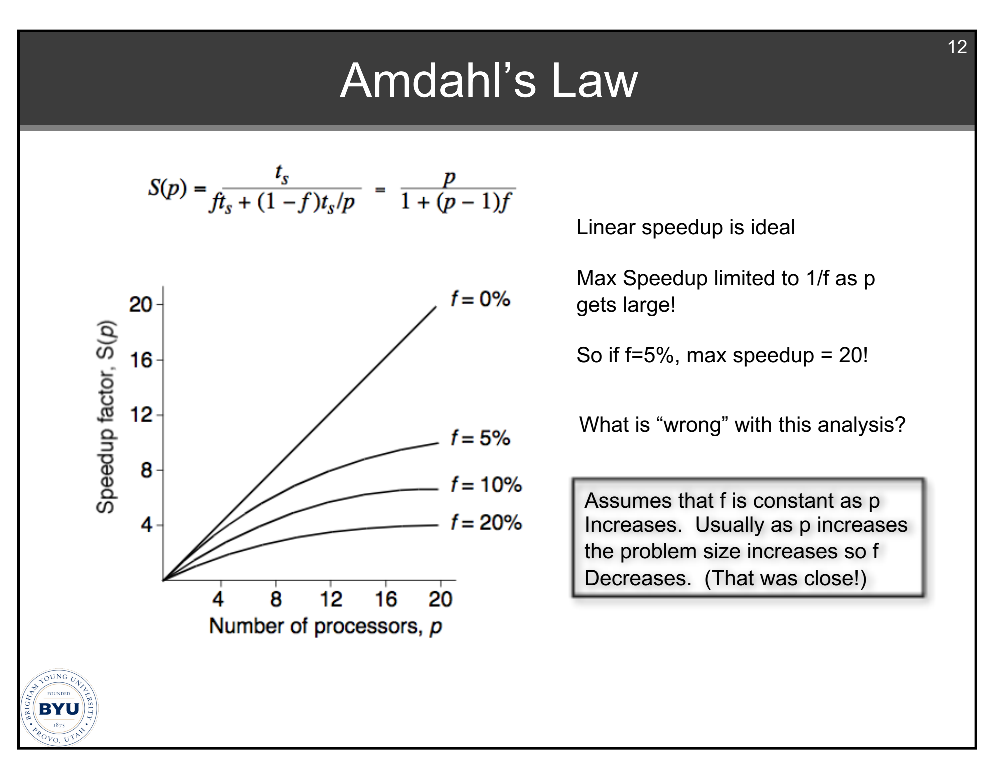


10

Some concepts and definitions

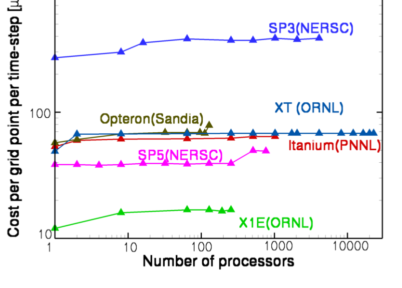
- Speedup Factor
  - $S(p) = t_{serial} / t_{parallel}$
  - S(p) = (best serial time) / (parallel time on p procs)
  - Serial and parallel algorithms are usually different.
  - Maximum speedup is p
    - Unless machines are different, or there is a nondeterministic algorithm.
    - This is rare, but happens.
- Efficiency
  - $E = S/p = t_{serial}/p*t_{parallel}$
- Tradeoff between communication and computation.
  - Communication takes time, reduces efficiency
  - Minimize communication between processors
    - Algorithm choice
    - Parallelization scheme
    - Communication network
  - As p increases, time decreases, but E decreases.


11
Amdahl's Law
ts
fts
(1 - f)ts
Serial section
Parallelizable sections
(a) One processor
(b) Multiple processors
p processors
(1 - f)ts/p
tp
BYU


12
Amdahl's Law
S(p) = ts / (fts + (1 − f)ts/p) = p / (1 + (p − 1)f)
Linear speedup is ideal
Max Speedup limited to 1/f as p gets large!
So if f=5%, max speedup = 20!
What is "wrong" with this analysis?
Assumes that f is constant as p Increases. Usually as p increases the problem size increases so f Decreases. (That was close!)
f = 0%
f = 5%
f = 10%
f = 20%
Speedup factor, S(p)
20
16
12
8
4
Number of processors, p
BYU

Example, S3D DNS Solver

- Cartesian domain decomposition
  - Only nearest neighbor communication.
  - Gives ~linear weak scaling since communication to computation ratio remains fixed
  - Communication scales with cube surface area, but computation goes with cube volume
- Explicit finite difference solver
  - 8th order central differences
  - 4th order, low storage RK integrator
- Solve reacting, compressible, Navier-Stokes equations.



***Strong scaling**: fix problem size, vary processors*

***Weak scaling**: vary problem size with processors to keep a fixed load.*

Which is more relevant is problem dependent (weak scaling will give "nicer" results.

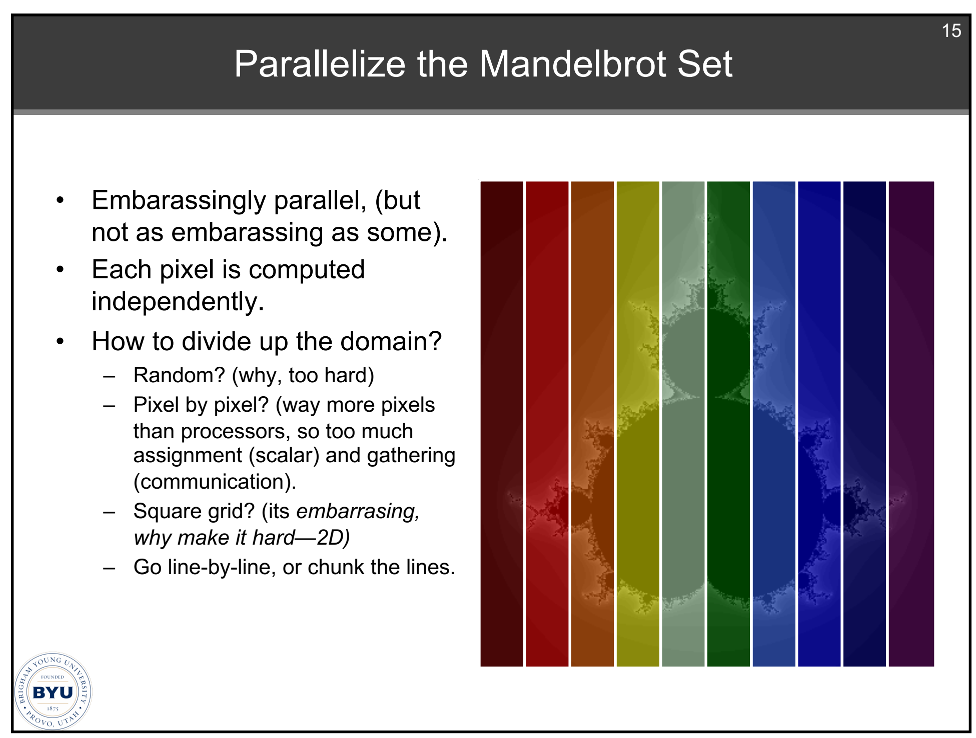
My first MPI Program

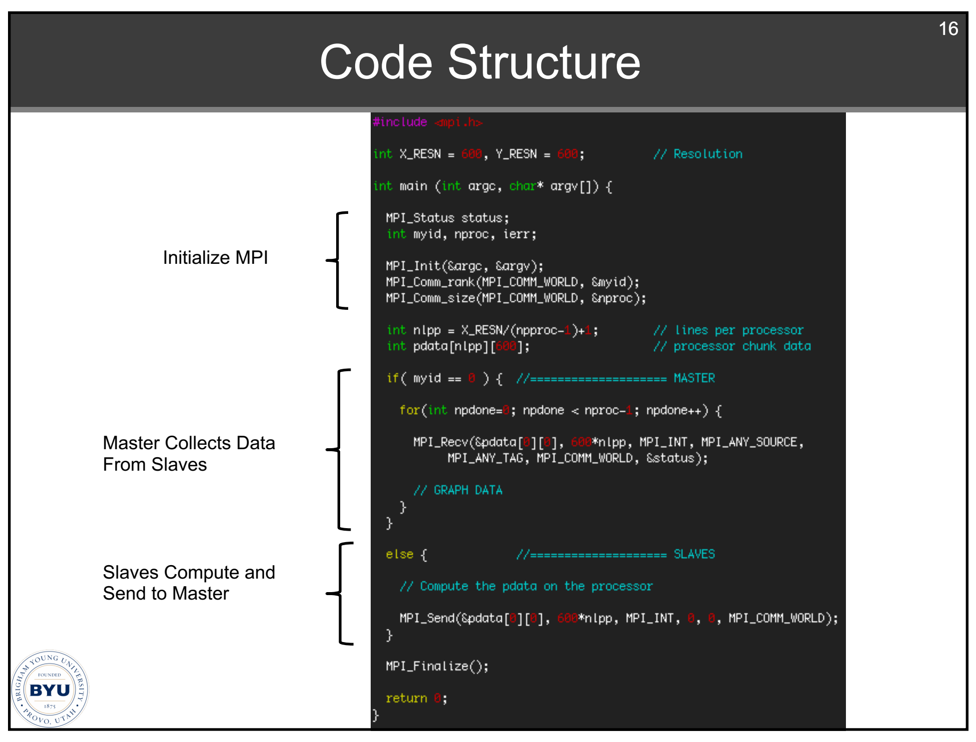
- Open MPI
- Compile:
  mpicxx myfirstMPI.cc
- Run:
  mpirun –np 128 a.out
- Include the mpi library
- MPI_Init: startup MPI, the first function called
- MPI_Comm_rank: get rank of processor making the call.
  - Relative to communicator MPI_COMM_WORLD (which just means all the processors). A communicator is a group of processors.
  - Processors ordered 0 to N-1.
- MPI_Comm_size: get # of procs, N
- MPI_Finalize: shutdown MPI (last MPI call).
- Note syntax.

```
#include <iostream>

#include <mpi.h>

using namespace std;

int main(int argc, char* argv[]) {

  int myid, nprocs, ierr;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  if(myid==0)
      cout << endl << "Hello from process " << myid << endl;
  if(myid==1)
      cout << endl << "Hello from process " << myid << endl;
  if(myid==2)
      cout << endl << "Hello from process " << myid << endl;
  if(myid==3)
      cout << endl << "Hello from process " << myid << endl;

  MPI_Finalize();

  return 0;
}
```

This works, but its silly, why?

Just replace ALL if statements with 1 cout statement


15
Parallelize the Mandelbrot Set
• Embarassingly parallel, (but not as embarassing as some).
• Each pixel is computed independently.
• How to divide up the domain?
– Random? (why, too hard)
– Pixel by pixel? (way more pixels than processors, so too much assignment (scalar) and gathering (communication).
– Square grid? (its embarrasing, why make it hard—2D)
– Go line-by-line, or chunk the lines.
BYU


16
Code Structure
Initialize MPI
Master Collects Data From Slaves
Slaves Compute and Send to Master
#include <mpi.h>
int X_RESN = 600, Y_RESN = 600; // Resolution
int main (int argc, char* argv[]) {
MPI_Status status;
int myid, nproc, ierr;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
int nlpp = X_RESN/(npproc-1)+1; // lines per processor
int pdata[nlpp][600]; // processor chunk data
if(myid == 0) { //==================== MASTER
for(int npdone=0; npdone < nproc-1; npdone++) {
MPI_Recv(&pdata[0][0], 600*nlpp, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
// GRAPH DATA
}
}
else { //==================== SLAVES
// Compute the pdata on the processor
MPI_Send(&pdata[0][0], 600*nlpp, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
MPI_Finalize();
return 0;
}
BYU

Send and Receive Messages

```
MPI_Send(void*          message,          MPI_Send(&pdata[0][0],
         int            count,                     600*nlpp,
         MPI_Datatype   datatype,                  MPI_INT,
         int            dest,                      0
         int            tag,                       0
         MPI_COMM       comm)                      MPI_COMM_WORLD)
```

```
MPI_Recv(void*          message,          MPI_Recv(&pdata[0][0],
         int            count,                     600*nlpp,
         MPI_Datatype   datatype,                  MPI_INT,
         int            source,                    MPI_ANY_SOURCE,
         int            tag,                       MPI_ANY_TAG,
         MPI_COMM       comm,                      MPI_COMM_WORLD,
         MPI_Status     status)                    &status)
```

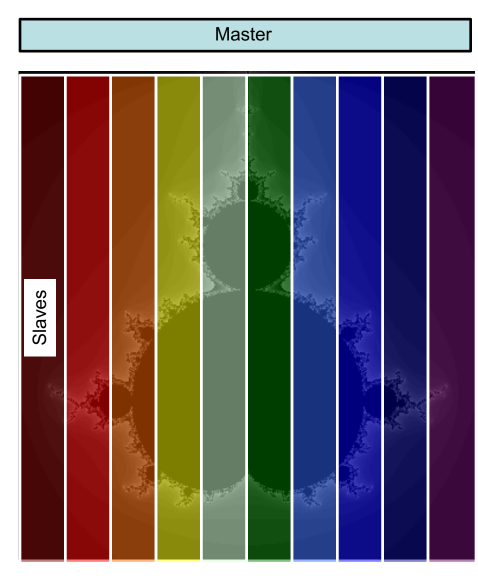
MPI_ANY_SOURCE is a built-in variable, but could be just an integer corresponding to a given processor.

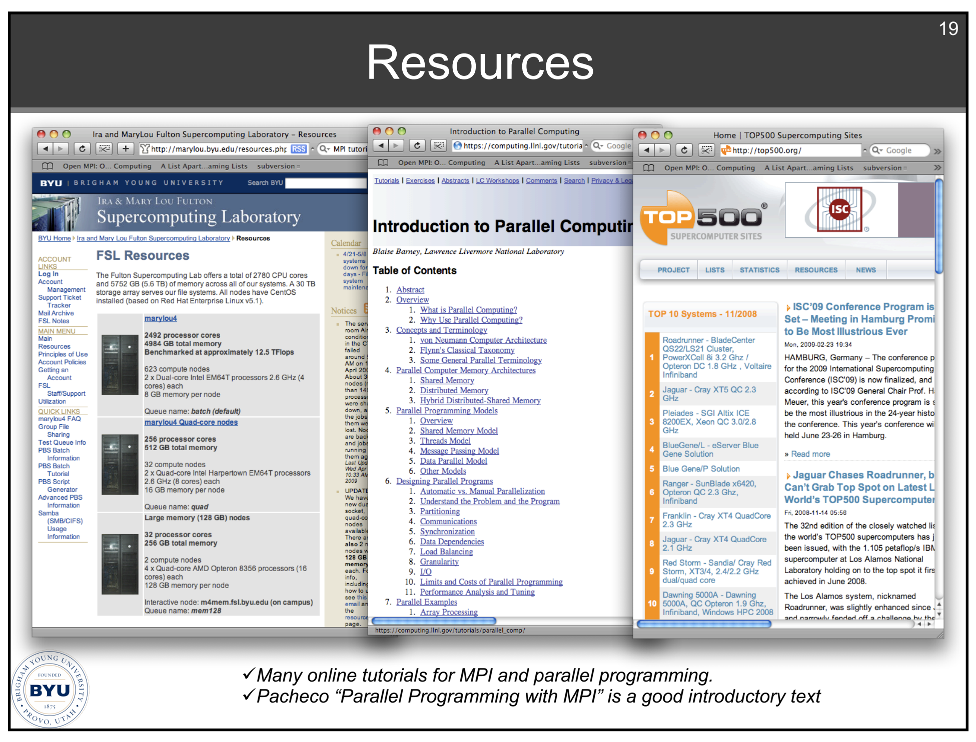
Same with MPI_ANY_TAG.

The status variable is declared as MPI_Status status.

Load Balancing

- Master-slave arrangement.
- One processor coordinates, gathers, directs, the others compute.
- Cost is not uniform over domain
  - Some processors will finish before others, and waste time.
- How to fix?
  - Divide into smaller chunks and farm them out as processors become available.
  - One could even dynamically determine workload and adjust optimal chunk size on the fly.




19
Resources
Supercomputing Laboratory
FSL Resources
Introduction to Parallel Computing
Table of Contents
TOP500
SUPERCOMPUTER SITES
✓Many online tutorials for MPI and parallel programming.
✓Pacheco “Parallel Programming with MPI” is a good introductory text
BYU