Deep Computing

Introduction to MPI Workshop

February 23-26 Part II – Review from November

Kirk E Jordan
Emerging Solutions Executive

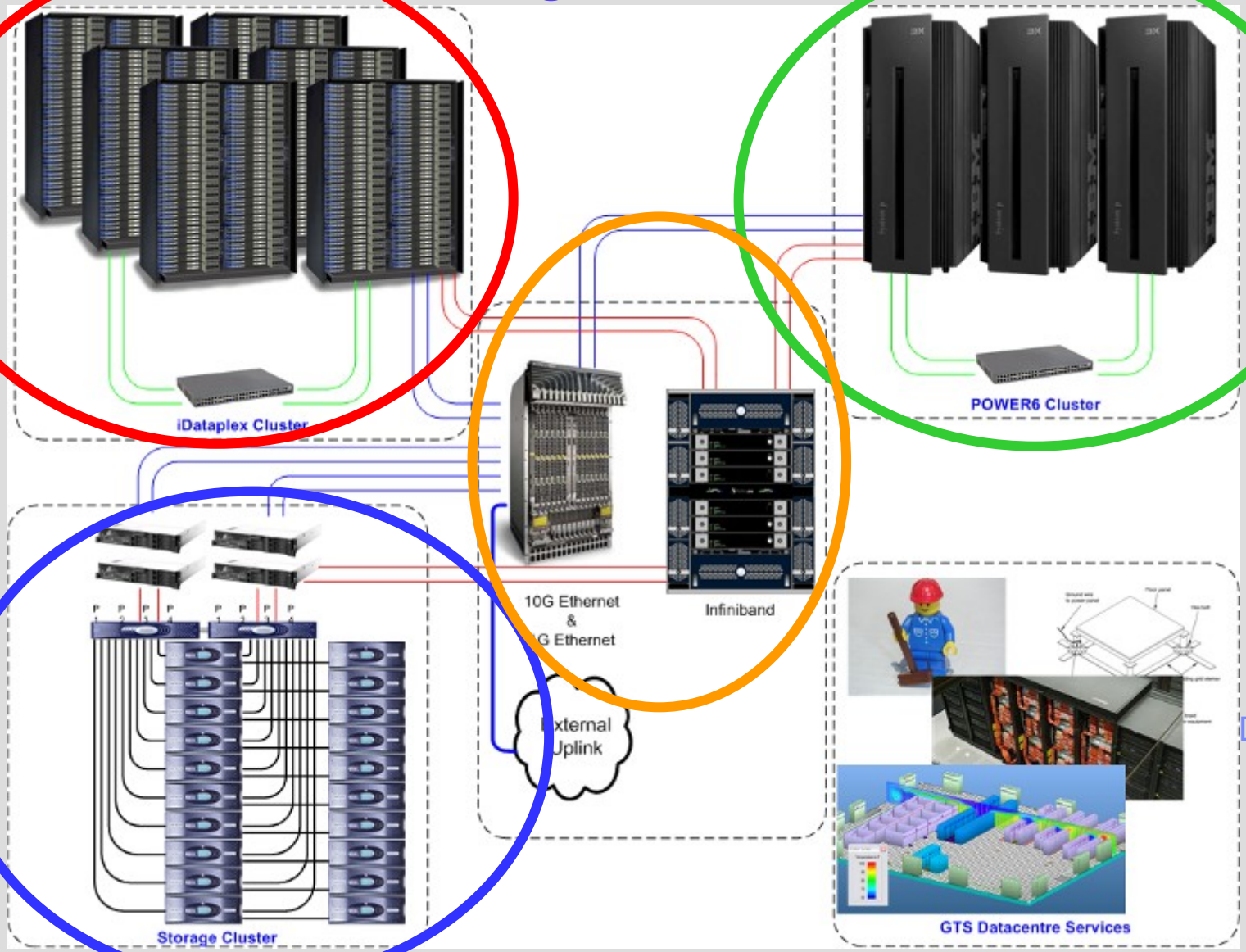
Computational Science Center
T.J. Watson Research Center

kjordan@us.ibm.com

Outline for Part 1- review (Goal – get basic background)

- **Quick review of characteristics of the hardware**
- **Overview Discussion of Parallel Programming**
- **Quick review of compilers – mpCC and mpxlf**
- **User Environment – setup/site dependent – SciNet staff provide**
- **Compile and Run/Execute a code**
- **Summary**

Solution Overview & Segments

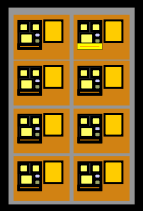


Hardware Overview

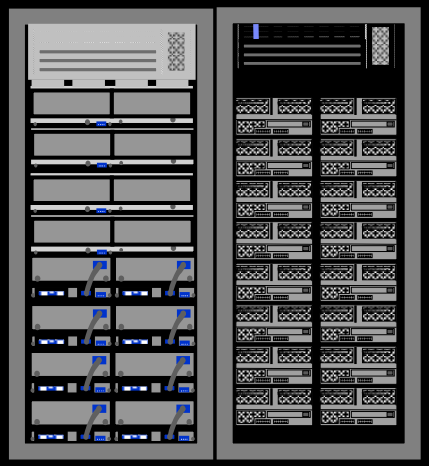
- Core:

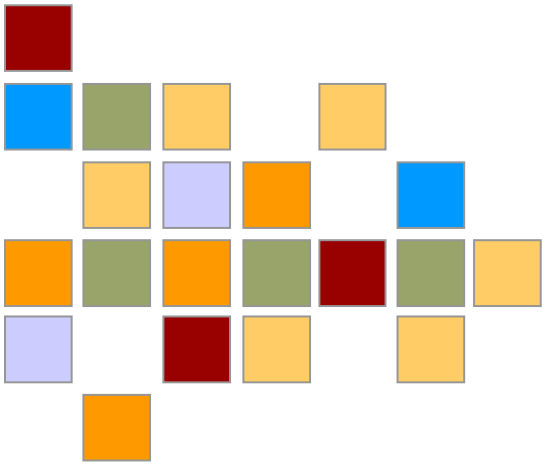
- Processors:

- Nodes:



- Clusters:

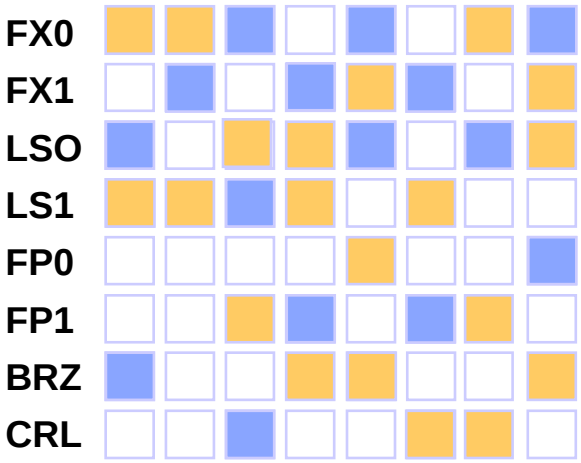




p575
POWER6

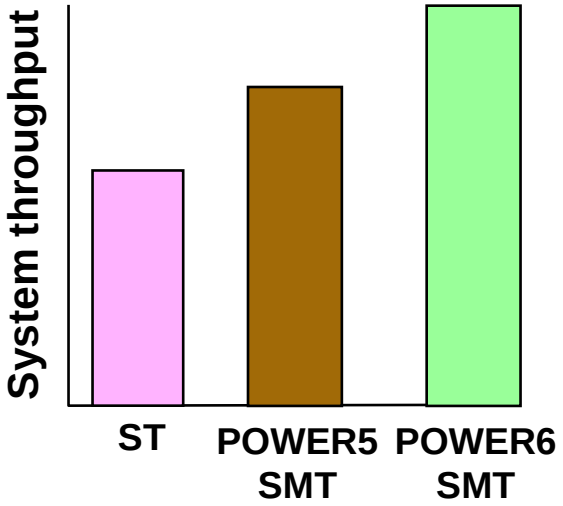
POWER6: Simultaneous Multithreading

POWER5 Simultaneous Multithreading



Thread0 active
No thread active
Thread1 active

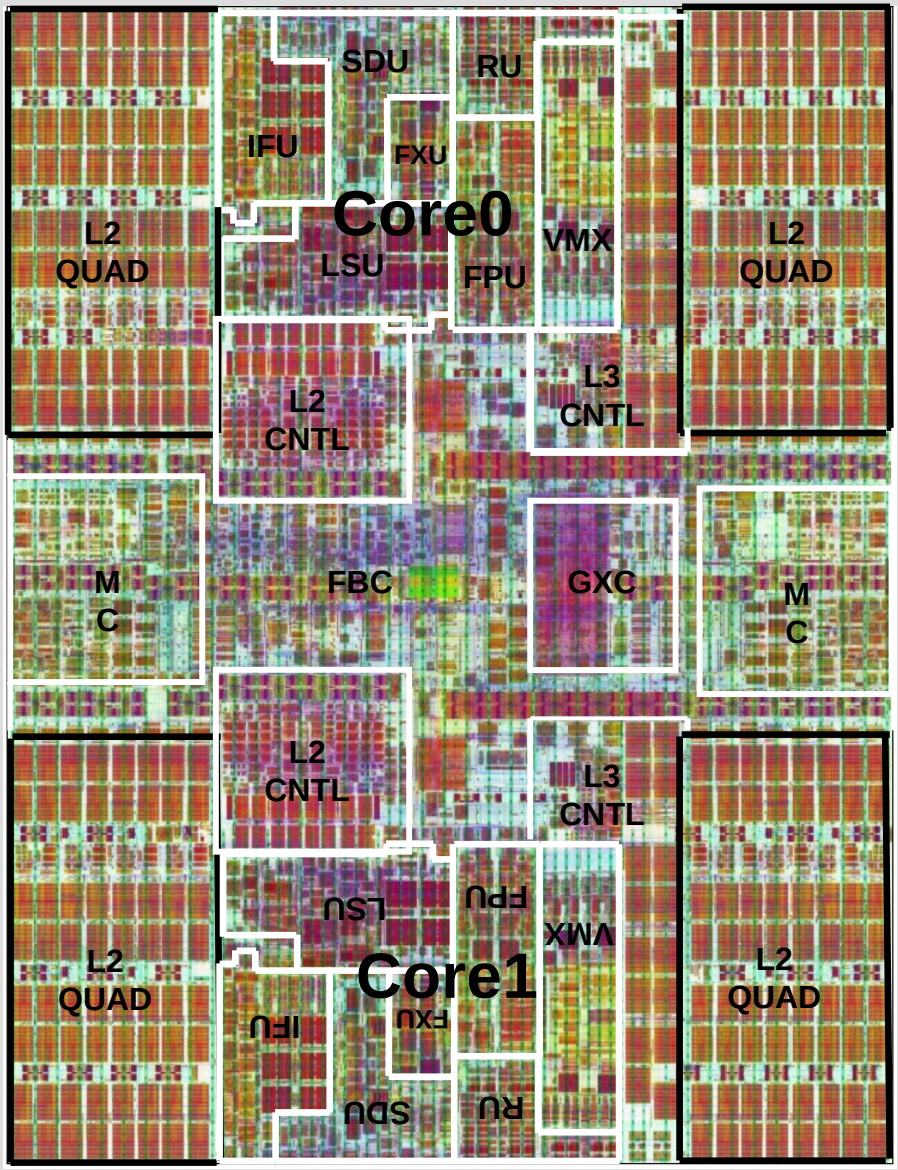
Appears as four CPUs per chip to the operating system (AIX V5.3 and Linux)



- Utilizes **unused execution** unit cycles
- **Reuse of existing transistors vs. performance from additional transistors**
- Presents symmetric multiprocessing (SMP) programming model to software
- Dispatch two threads per processor: *"It's like **doubling** the number of processors."*
- Net result:
  - ***Better performance***
  - ***Better processor utilization***

POWER6 Chip Overview

- **Ultra-high frequency dual-core chip**
  - 7-way superscalar, 2-way SMT core
    - up to 5 instr. for one thread, up to 2 for other
  - 8 execution units
    - 2LS, 2FP, 2FX, 1BR, 1VMX
  - 790M transistors, 341 mm^2 die
  - Up to 64-core SMP systems
  - 2x4MB on-chip L2 – point of coherency
  - On-chip L3 directory and controller
  - Two memory controllers on-chip
- **Technology**
  - CMOS 65nm lithography, SOI Cu
- **High-speed elastic bus interface at 2:1 freq**
  - I/Os: 1953 signal, 5399 Power/Gnd
- **Full error checking and recovery**



POWER6 Objectives

- Processor Core
- **High single-thread performance with ultra high frequency (13FO4) and optimized pipelines**
- **Higher instruction throughput: improved SMT**

- Cache and Memory Subsystem
- **Increase cache sizes and associativity**
- **Low memory latency and increased bandwidth**

- System Architecture
- **Fully integrated SMP fabric switch**
  - Predictive subspace snooping for significant reduction of snoop traffic
  - Higher coherence bandwidth
  - Excellent scalability

- **Ultra-high frequency buses**
  - High bandwidth per pin
  - Enables lower cost packaging

- Power
- **Minimize latch count**
- **Dynamic Power management**

Power6 Highlights for performance

- **Single cycle FX to FX pipeline (two per core)**
- **Six-cycle FP pipeline (two per core)**
- **4MB L2 per core with 32MB L3 per chip extension**
- **Comprehensive and flexible data prefetching system with**
  - High bandwidth capability from DIMMS and caches into the registers
- **VMX for 32bit calculations (fixed/single-precision)**

POWER6 p575 Node

AIX 6 AIX V5.3 Linux

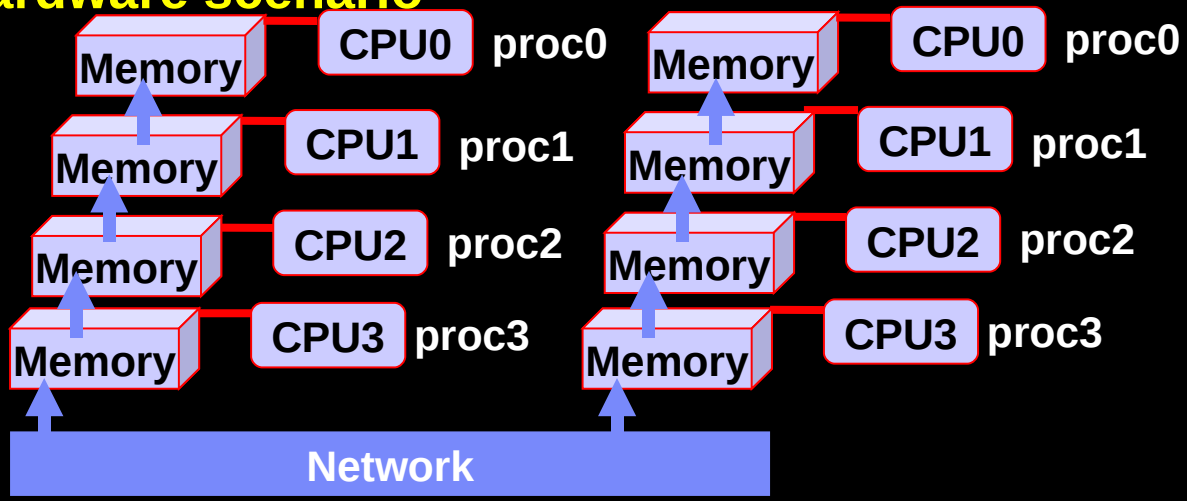
Compute Node	
Architecture	32-core node 1 – 14 nodes / rack (448 Cores) 4.7 GHz
Cache	L3: 32MB / chip
DDR2 Memory	4 to 256 GB (Buffered)
DASD / Bays	2 SAS DASD (2.5”)
Expansion	PCIe / PCI-X support
IVE	Yes
Integrated SAS	Yes
Expansion Slots	Dual GX Bus Adapters
Integrated Ethernet	Two Dual 10/100/1000 Ethernet Optional Dual 10Gb
POWER	N+1 Support 1 - 4 Nodes 2 Line Cords 5+ Nodes 4 Line Cords
Cooling	Water / Air
Remote IO Drawers	Yes Quantity: 1 PCI-X (20 Slots)

Parallel Programming Basics

Comments

Distributed Memory Program Architecture Characteristics in early 1990s

- Clusters of single CPU systems were used to run MPI jobs
- Each system had its own OS
- Single compute process ran on each system
- Each process had its own address space
- Message passing between processes had to go through network
- MPI standard was initially developed to support this hardware scenario



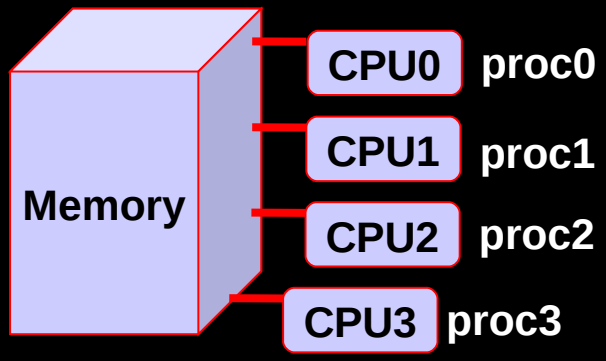
Distributed Memory Program Architecture
New characteristics in late 1990s

- Large SMP systems started to be used to run MPI jobs
- It had multiple CPU systems, Each system had its own OS
- Multiple compute processes ran within each system
- Each process had its own address space
- Message passing between the processes can go through memory instead of network

- Hardware vendors developed algorithm using shared memory to conduct message passing between the processes
- There's no need to change MPI standard for this scenario

Just add the following into MPI job run script

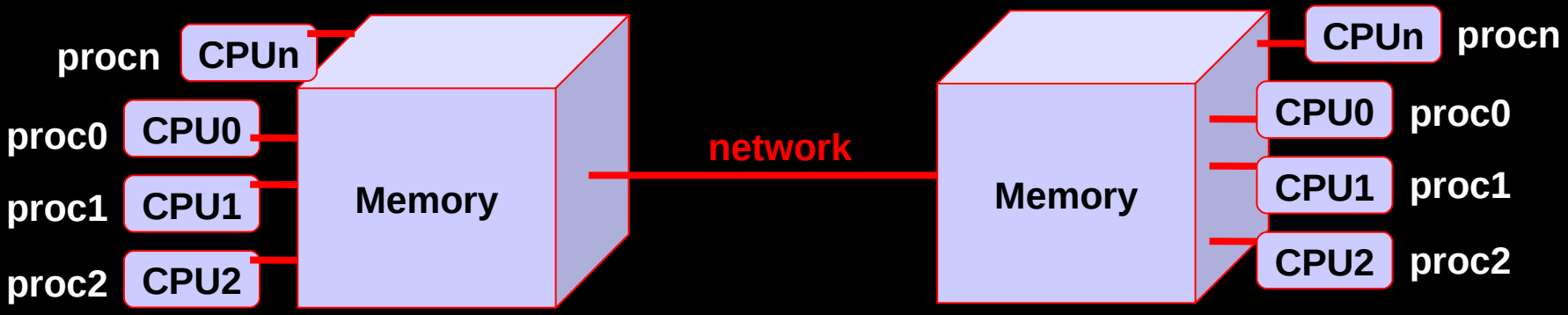
export MP_SHARED_MEMORY=yes



Distributed Memory System Architecture
New characteristics in 2000s

- Cluster of SMP systems started to be used to run MPI jobs
- Each system has multiple CPUs, each system had its own OS
- Multiple compute processes ran on each system
- Each process had its own address space
- Message passing between the processes may go through both memory and network

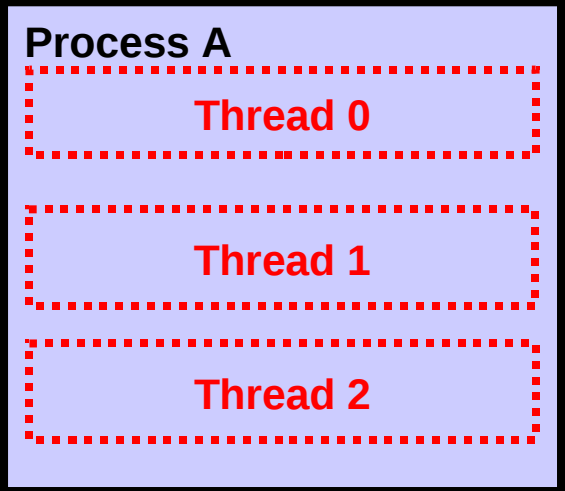
- We can still use shared memory for message passing between processes within each SMP system
- There's no need to change MPI standard for this scenario



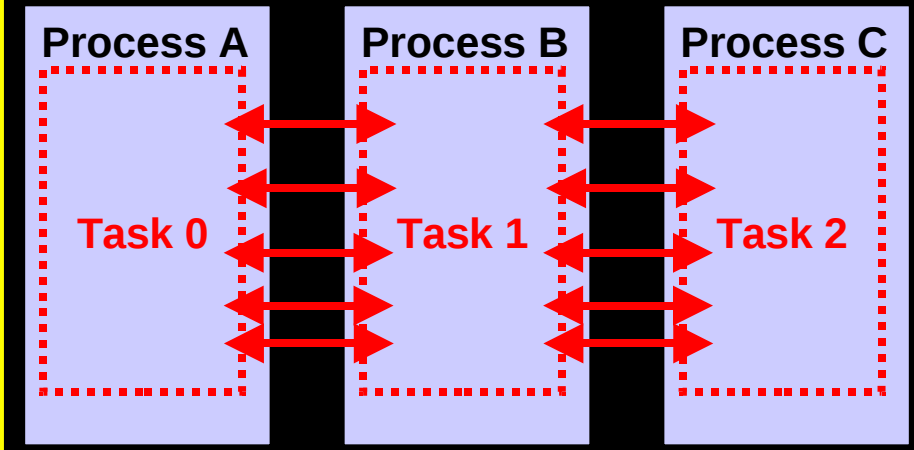
User shared-memory for message passing

Comparison: Shared Memory Programming vs. Distributed Memory Programming

- **Shared memory Single process ID for all threads**
  - List threads
    - ps –om THREAD



- **Distributed memory**
  - Each “task” has own process ID
  - List tasks:
    - ps



As we saw in SMP chapter

Parallel programming is essential to exploit modern computer architectures

- Single processor performance is reaching limits
  - Moore's Law still holds for transistor density, but…
  - Frequency is limited by heat dissipation and signal cross talk
  - Multi-core chips are everywhere...
- Advances in network technology allow for extreme parallelization

Parallel choices

- MPI
  - Good for tightly coupled computations
  - Exploits all networks and all OS
  - No limit on number of processors
  - Significant programming effort; debugging can be difficult
  - Master/Slave paradigm is supported, as well
- OpenMP
  - Easy to get parallel speed up
  - Limited to SMP (single node)
  - Typically applied at loop level ← limited scalability
- Automatic parallelization by compiler
  - Need clean programming to get advantage
- pthreads = Posix threads
  - Good for loosely coupled computations
  - User controlled instantiation and locks
- fork/execl
  - Standard Unix/Linux technique

Parallel programming recommendations (for scientific and engineering computations)

- Use MPI if possible
  - Performance on SMP node is almost always at least as good as OpenMP
  - For 1-D, 2-D domain decomposition: schedule 2 months work
  - For 3-D domain decomposition: schedule 3-4 months
- OpenMP can get good parallel speed up with minimal effort
  - 1 week to get 70% efficient on 4 cores; 3 weeks to get 90%
  - May get best performance with –qsmp=omp instead of relying on compiler to auto-parallelize for older codes
    - Can use -qsmp –qreport=smplist to get candidate loops.
- Hybrid is also possible
  - OpenMP under MPI
- pthreads are fine. Use them if it makes sense for your program.

Terminology Review: Processor vs. Node

Identical to what we said for SMP

- At the scale of microprocessors
  - CPU = processor = core
  - Chip = socket
  - IBM started delivering dual-core POWER4 technology to the user community in 2001
- At the scale of a computer system
  - Node = system = box
  - Cluster = many nodes connected together via fast network
  - A node runs a SINGLE image of operating system

Terminology Review: Thread vs. Process

In addition to what we said for SMP

- Thread:
  - An independent flow of control, may operate within a process with other threads.
  - An schedulable entity
  - Has its own stack, thread-specific data, and own registers
  - Set of pending and blocked signals
- Process
  - Can not share memory directly
  - Can not share file descriptors
  - A process can own multiple threads
- An OpenMP job is a process. It creates and owns one or more SMP threads. All the SMP threads share the same PID
- An MPI job is a set of concurrent processes (or tasks). Each process has its own PID and communicates with other processes via MPI calls

Apply MPI Technology to Real World Problem

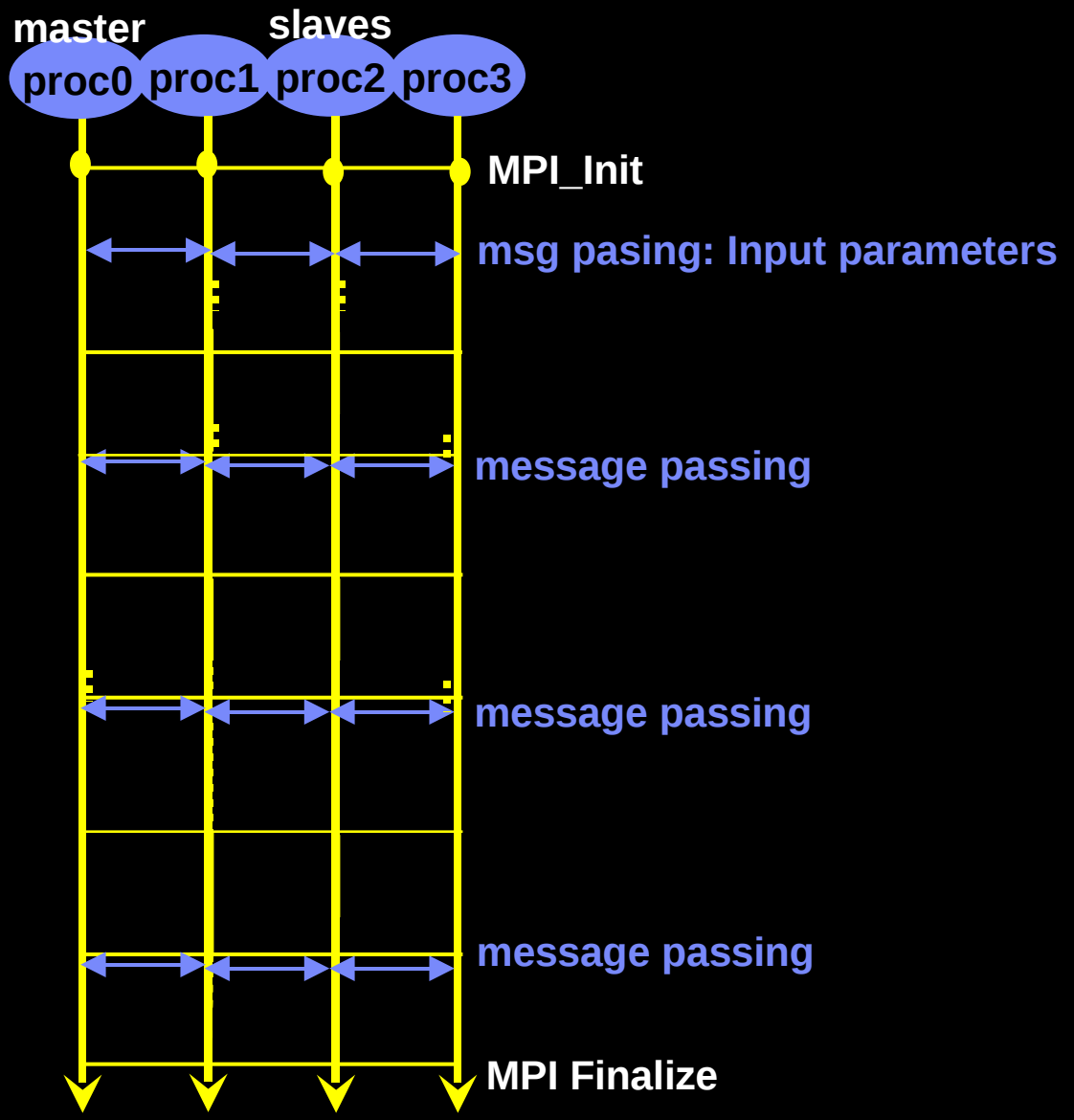
- Multiple steps in applying MPI technology to solve a Sci&Eng problem
  - 1. Divide workload to multiple processes (domain decomposition)
  - 2. Execute your MPI program
  - 3. Collect and process the output data
- Questions
  - Which filesystem should I use for my input, scratch and output files? What do I do if global filesystem is not available (i.e. grid computing scenario)?
  - How to map the MPI processes to available processors?

3 Steps in a Distributed Computing Job

- **Step 1: Domain Decomposition (workload partition)**
  - To divide workload into N chunks, one for each MPI tasks
  - Often carried out as a serial or SMP pre-processing job/ Example: FLUENT, PowerFLOW,STARCD
- **Step 2: the MPI program**
  - To performance computation
- **Step 3: final result assembly**
  - Some code merge this into stage 2
  - while others need to run a post-processing job to assemble output from each MPI tasks. Example: LSDYNA. LSDYNA also merged stage 1 and stage 2.

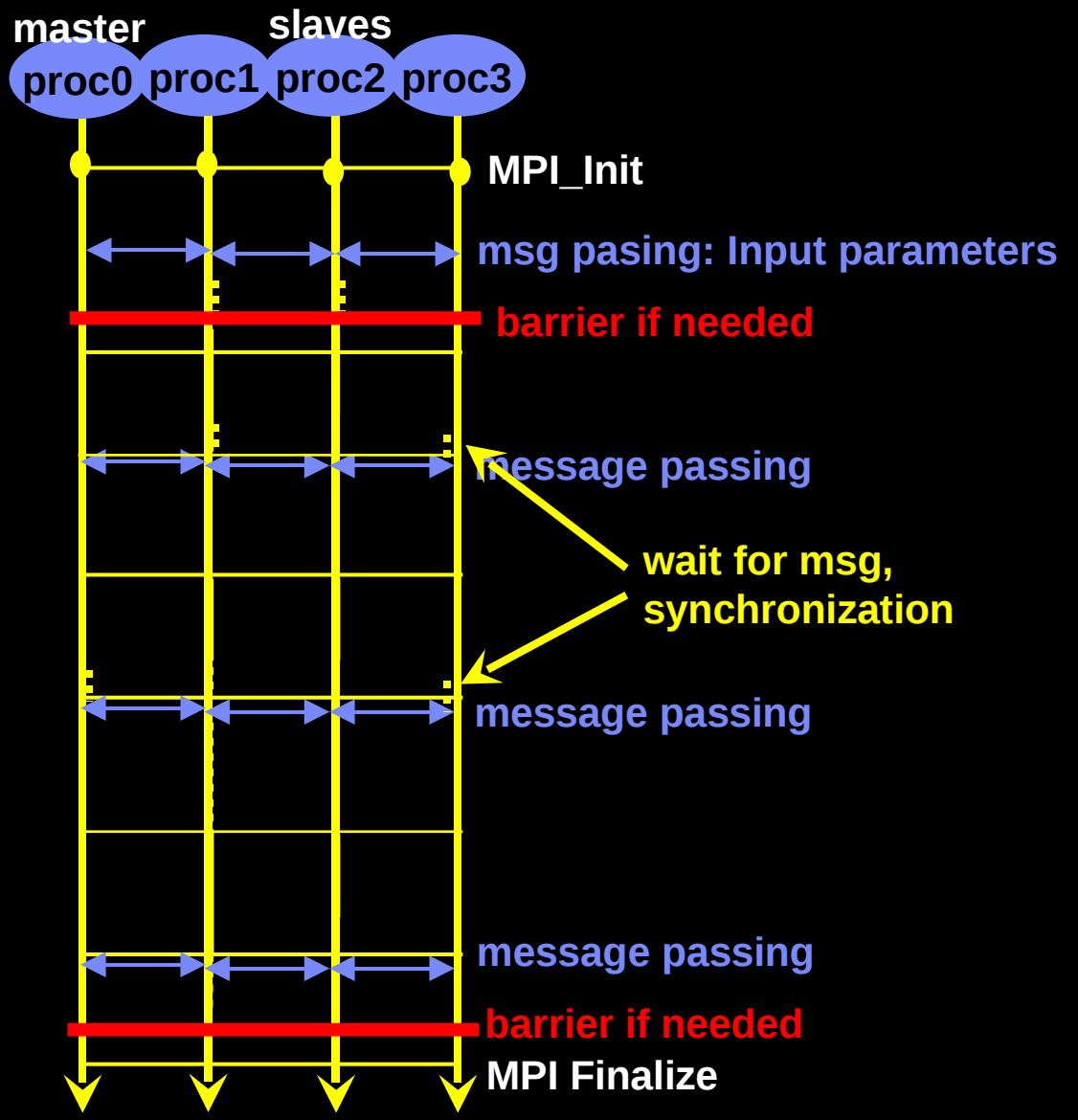
Schematic Flow of an MPI Code

```
Program ParallelWork

start the program
Call MPI_Init(ierr)
read input
set up comp. parameters
…
DO i=1,imax
…
…
End do
…
serial work
…
DO i=1,imax
…
…
End do
…
serial work
…
DO i=1,imax
…
…
End do
…
Output
Call MPI_Finalize
End program ParallelWork
```

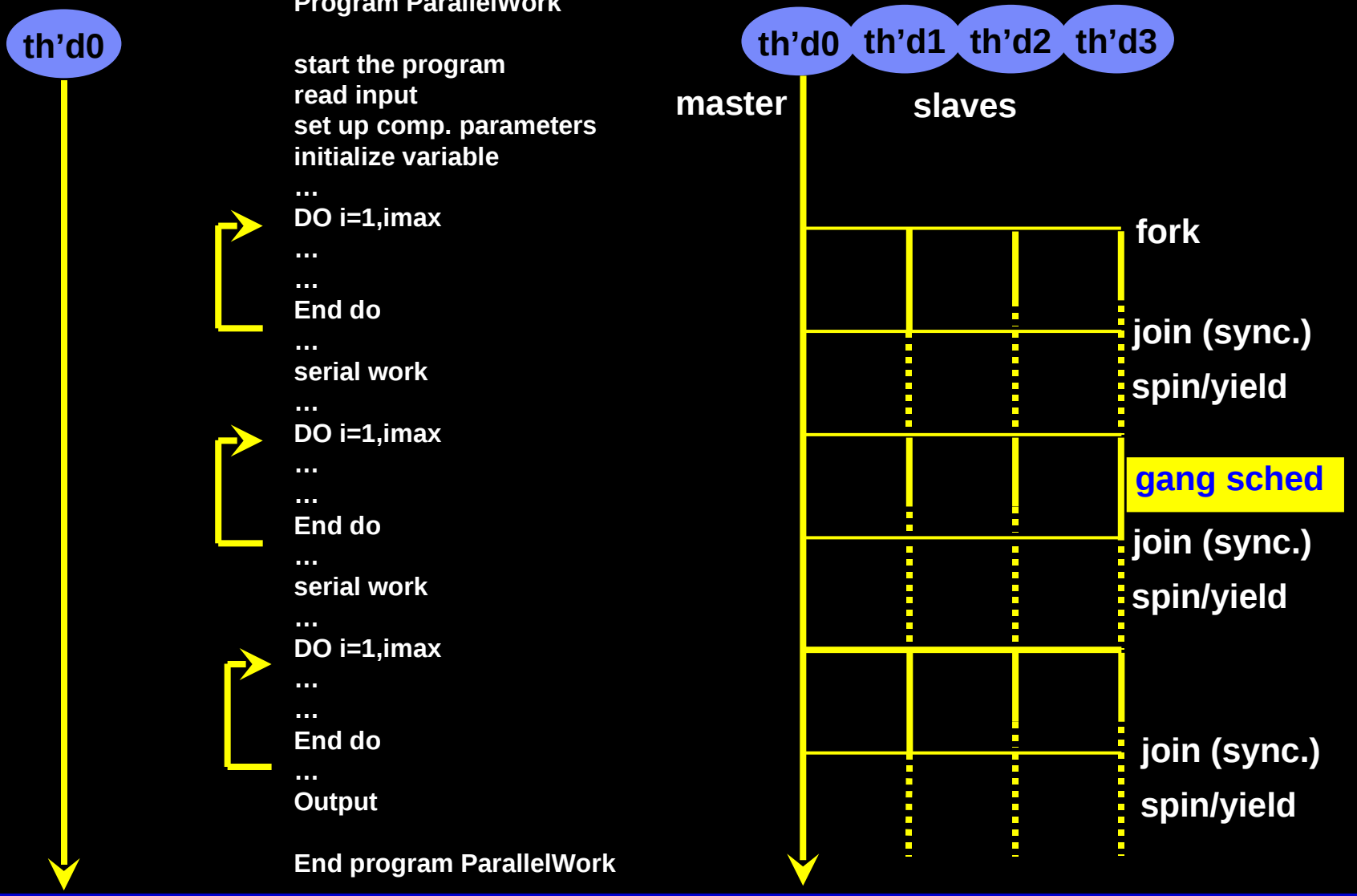


Schematic Flow of an MPI Code

```
Program ParallelWork

start the program
Call MPI_Init(ierr)
read input
set up comp. parameters
…
DO i=1,imax
…
…
End do
…
serial work
…
DO i=1,imax
…
…
End do
…
serial work
…
DO i=1,imax
…
…
End do
…
Output
Call MPI_Finalize
End program ParallelWork
```



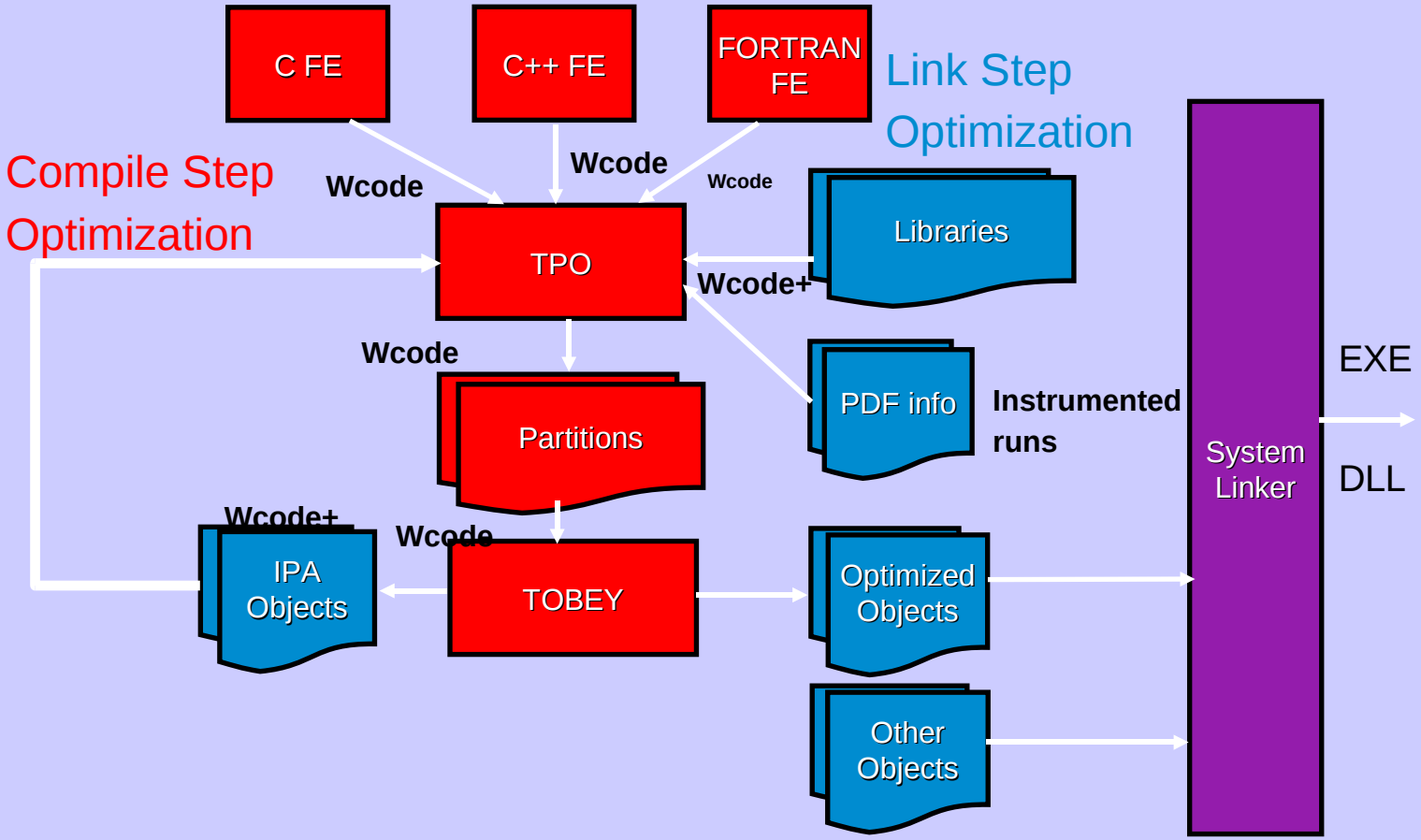
Review: Schematic Flow of an SMP Code



MPI options

- IBM Parallel Environment
  - POE
  - Highly optimized for IBM processors, adapters, and networks
  - Have to purchase license
- MPICH
  - Uses TCP/IP protocol
  - Free
- LAM MPI
  - Free
- OpenMPI
  - Free, but new…
- etc., etc.

IBM XL compiler architecture



Some Environment Hints

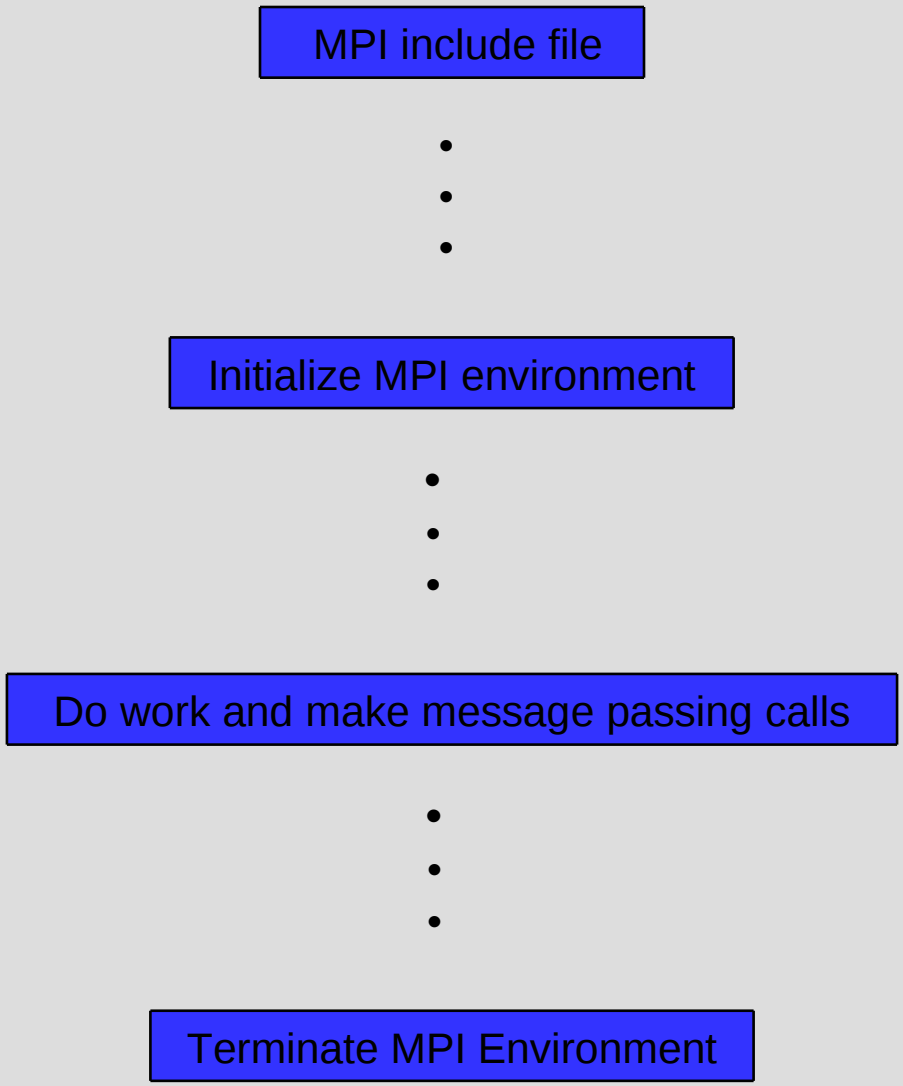
- **If you get mpcc:**
  - Command not found. or something similar, your PATH doesn't contain the location of the MPI commands.
  - You may need something like (for the MPICH implementation)

    setenv PATH /usr/local/mpi/lib/sun4/ch_p4:/usr/local/mpi/bin:$PATH

    rehash
    - Or something similar.
    - The exact path will depend on your MPI installation and the devices that you are using.
    - The MPI standard does not specify how MPI programs are compiled or run; this is up to the implementation. The examples here are for the MPICH implementation.
- **If your program runs, but runs with only one processor, you may be accessing an mpexec for a different version of MPI.**
  - Give the command

    which mpiexec
    - Make sure that the PATH given matches the one that cooresponds to the MPI implementation that you are using.

Overview of MPI Program Structure



Invoking the MPI Compiler

Language	Compiler
Fortran 77	mpxlf
Fortran 90	mpxlf90
Fortran 95	mpxlf95
C	mpcc
C++	mpCC

Need to compile, link and execute -

- **Compile & Link:**
  - mpcc -o minim minim.c
  - mpxlf –o minim minim.f
- **Via Loadlever, submit for execution:**
  - llsubmit minim.cmd
- **Execution line:**
  - mpiexec -n 5 -cwd `pwd` -exe minim.x

How to submit jobs at SciNet

- **Submission process - Loadleveler**
- **Overview of queues**
- **Other environment setup**

C Language File - minim.c

```
/*
==========================================================
=
    .                    Minim:                              .
    .                    -----                             .
    . MINIMAL program construction.                      .
    . Functions used:                                      .
    .                                                          .
    .          MPI_Init                                .
    .          MPI_Comm_size                             .
    .          MPI_Comm_rank                              .
    .          MPI_Finalize                            .
        . This is a minimal program that starts up, does simple I/O and then quits, all to
illustrate the .
    . basic initializing and finalizing calls under
MPI.                                                       .

==========================================================
= */
#include <stdio.h>
#include "mpi.h"
main(int argc, char **argv)
{
  int nnode;   /* Number of processor.        */
  int inode;   /* This specific processor. */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nnode);
  MPI_Comm_rank(MPI_COMM_WORLD, &inode);
  /* Print only from node 0.   */
  if (inode == 0) {
     printf(" Running program %s\n", argv[0]);
     printf(" The total number of nodes is %d \n", nnode);
            }
  /* Print from all nodes. */
  printf(" Hello from node %d\n", inode);
  MPI_Finalize();   /* Clean-up. */
}
```

Env Routines - C Language - simple.c

```
#include "mpi.h"
  #include <stdio.h>
  int main(argc,argv)
  int argc;
  char *argv[]; {
  int       numtasks, rank, rc;

  rc = MPI_Init(&argc,&argv);

  if (rc != 0) {
        printf ("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
        }

   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);
   /*******   do some work *******/

   MPI_Finalize();
   }
```

Env Routines - Fortran Language - simple.f

```
      program simple
          include 'mpif.h'
          integer numtasks, rank, ierr, rc

          call MPI_INIT(ierr)

          if (ierr .ne. 0) then
                print *,'Error starting MPI program. Terminating.'
                call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
          end if
          call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
          call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

          print *, 'Number of tasks=',numtasks,' My rank=',rank
C ****** do some work ******

          call MPI_FINALIZE(ierr)
          end
```

Summary

- **Brief overview of system**
- **Comment on basics of parallel programming**
- **Compiling and linking a program – to get started**
  - Minimum MPI program
  - Simple program
- **Loadleveler – job scheduler**

Note on Core files

- **Core files are text files. Look at the core file with a text editor, focus on the function call chain; feed the hex addresses to addr2line.**
  - addr2line -e your.x hex_address
  - tail -n 10 core.511 | addr2line -e your.x
- **Use grep and word-count (wc) to examine large numbers of core files:**
  - grep hex_address “core.*” | wc -l