Deep Computing

Introduction to MPI Workshop

Nov 6-7, 2008 Part I

Kirk E Jordan
Emerging Solutions Executive

Deep Computing
Systems & Technology Group

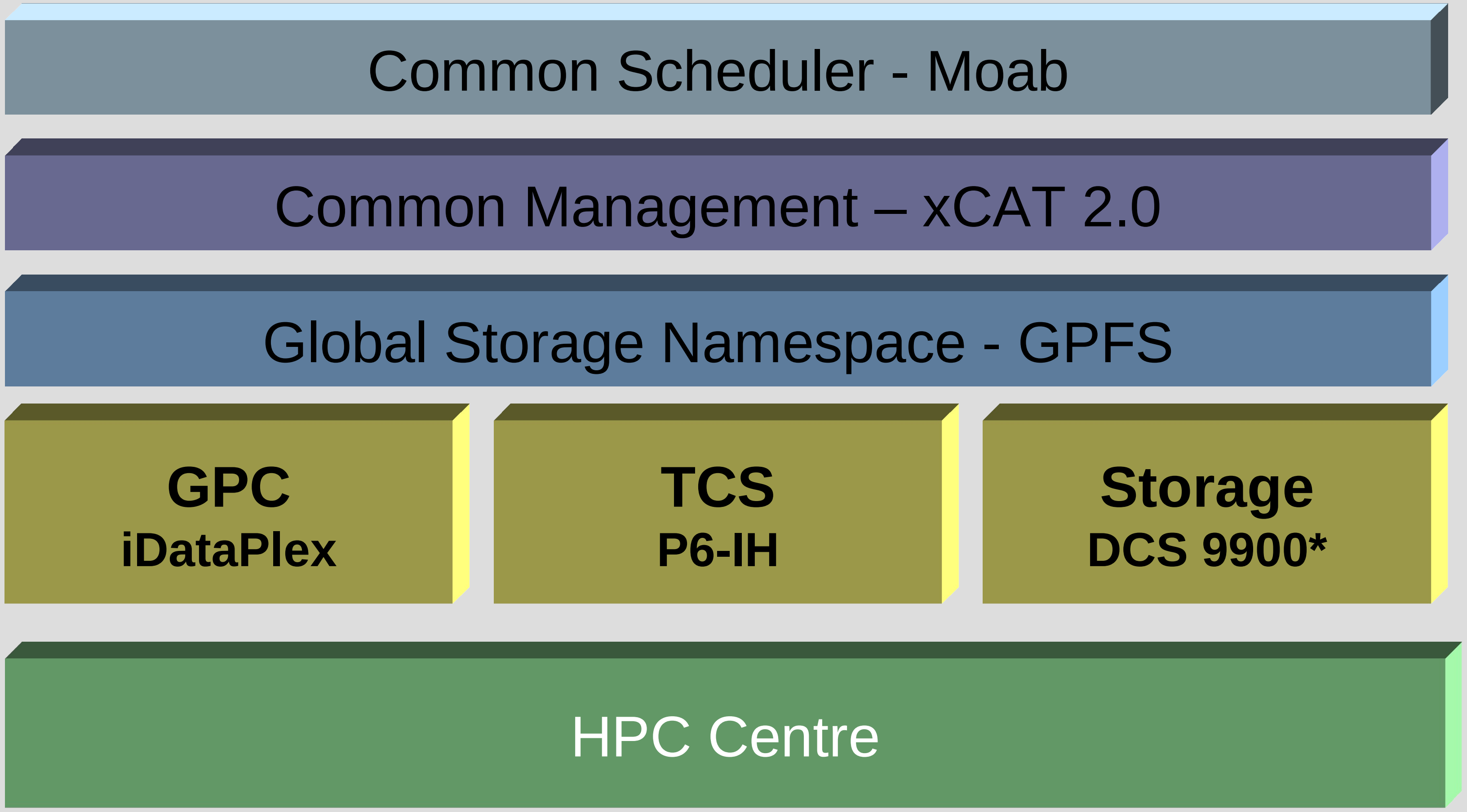
Computational Science Center
T.J. Watson Research Center

kjordan@us.ibm.com

Outline for Part 1 (Goal – get basic background)

- **Some basic user information/background**
- **Characteristics of the hardware**
- **Software overview**
- **User Environment – setup/site dependent**
- **Run an application**
- **What is MPI?**
- **Getting Started**
- **Essential Management Routines**
- **Run an MPI Program**
- **Summary**

Overview Systems
Integrated by design



*Storage was proposed in partnership with Datadirect Networks, with the intention of moving to an IBM branded solution if possible.

Solution Overview & Segments

30k Cores

300 TFlops Peak

5PB Disk

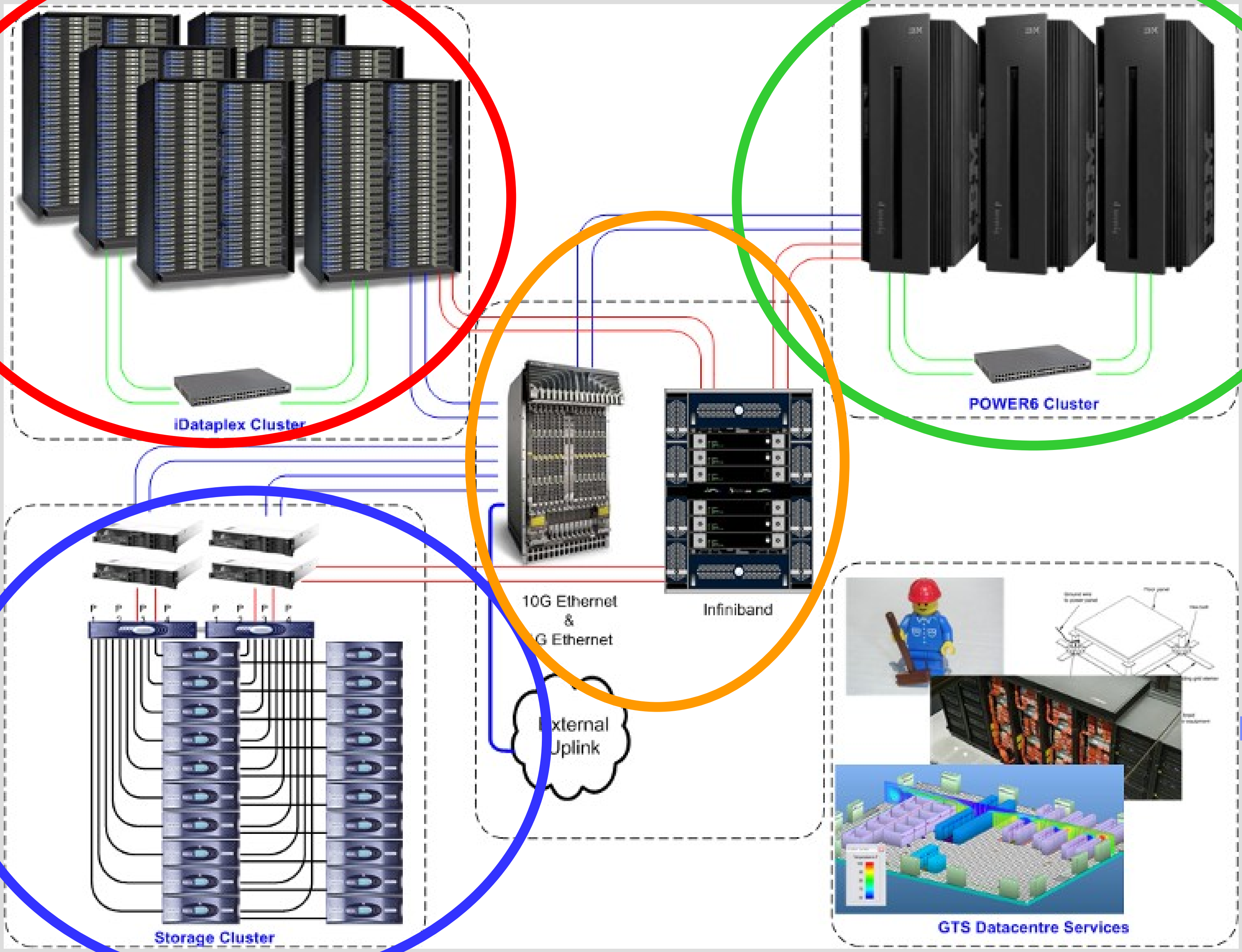
Highest Density

3.3k Cores

60 TFlops Peak

2.4MW
2.5k sqft
Datacenter

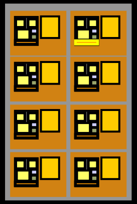
1.16 PUE



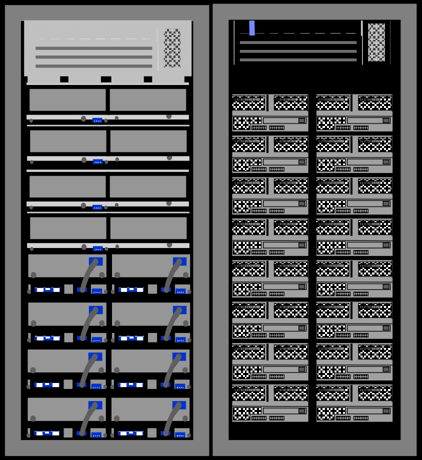
Hardware Overview

- Core:

- Processors:



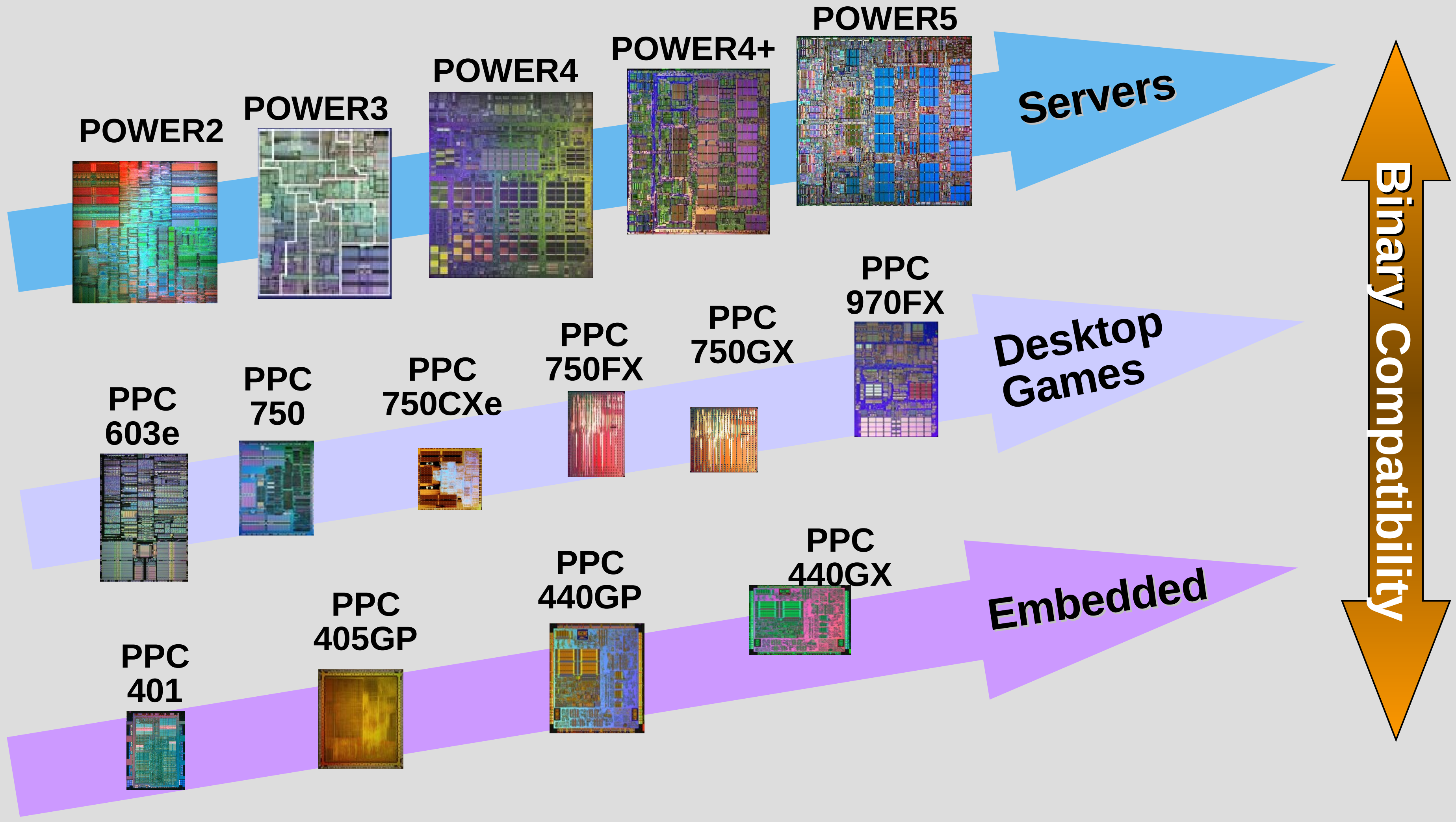
- Nodes:



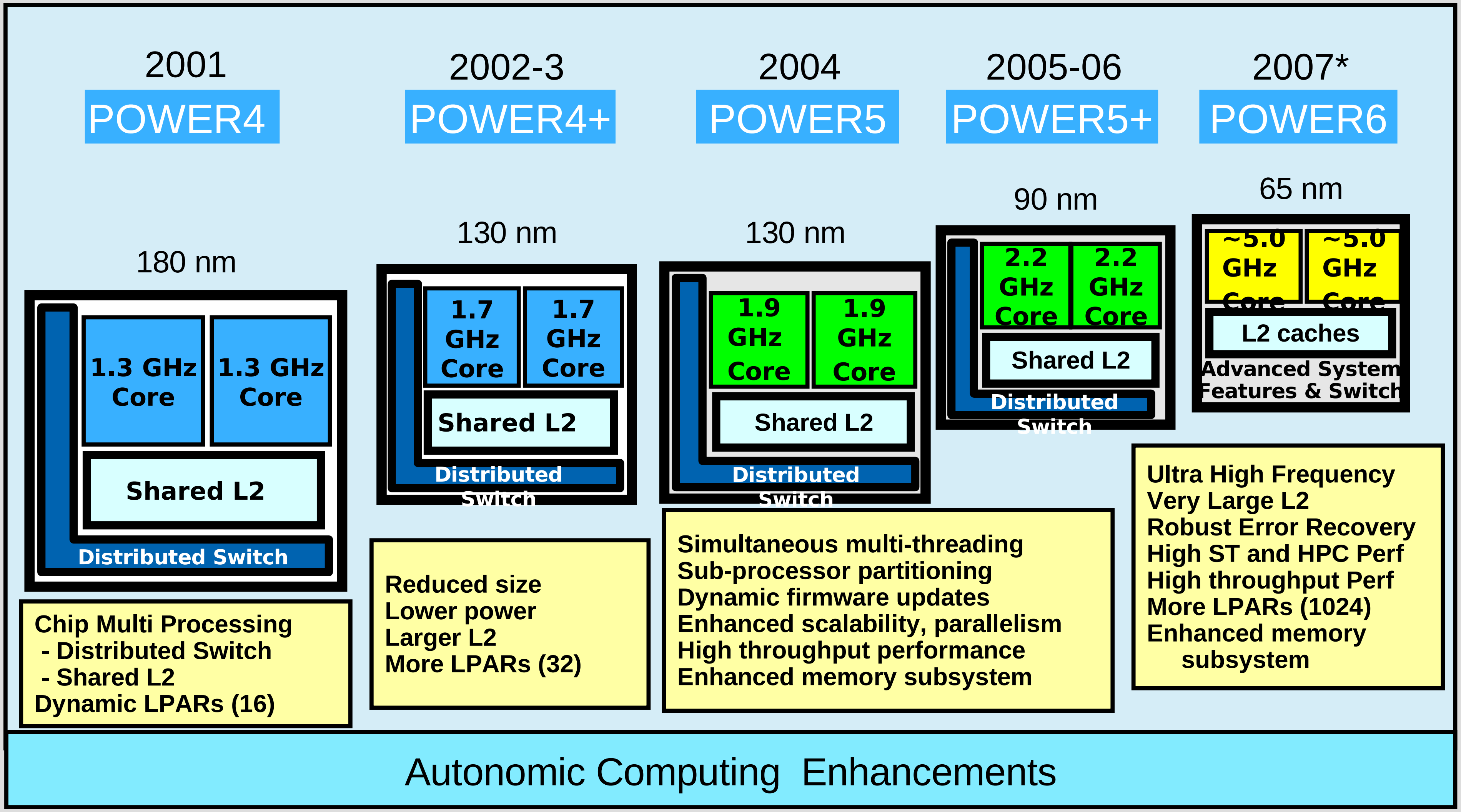
- Clusters:

p575
POWER6

POWER : The Most Scaleable Architecture



POWER Server Roadmap



Chip Multi Processing
- Distributed Switch
- Shared L2

Dynamic LPARs (16)

Reduced size
Lower power
Larger L2
More LPARs (32)

Simultaneous multi-threading
Sub-processor partitioning
Dynamic firmware updates
Enhanced scalability, parallelism
High throughput performance
Enhanced memory subsystem

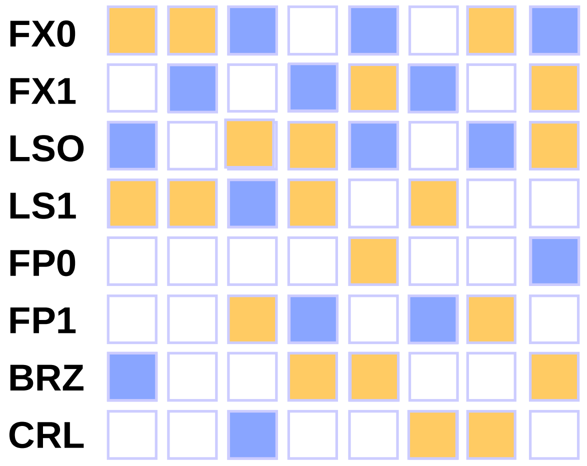
Ultra High Frequency
Very Large L2
Robust Error Recovery
High ST and HPC Perf
High throughput Perf
More LPARs (1024)
Enhanced memory subsystem

Autonomic Computing Enhancements

*Planned to be offered by IBM. All statements about IBM's future direction and intent are subject to change or withdrawal without notice and represent goals and objectives only.

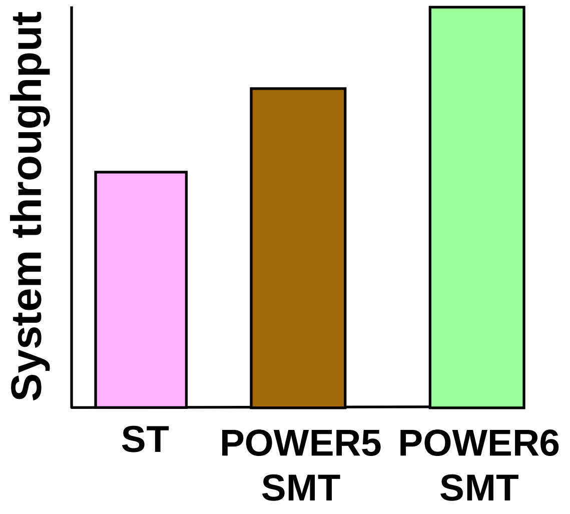
POWER6: Simultaneous Multithreading

POWER5 Simultaneous Multithreading



🟨 **Thread0 active**
⬜ **No thread active**
🟦 **Thread1 active**

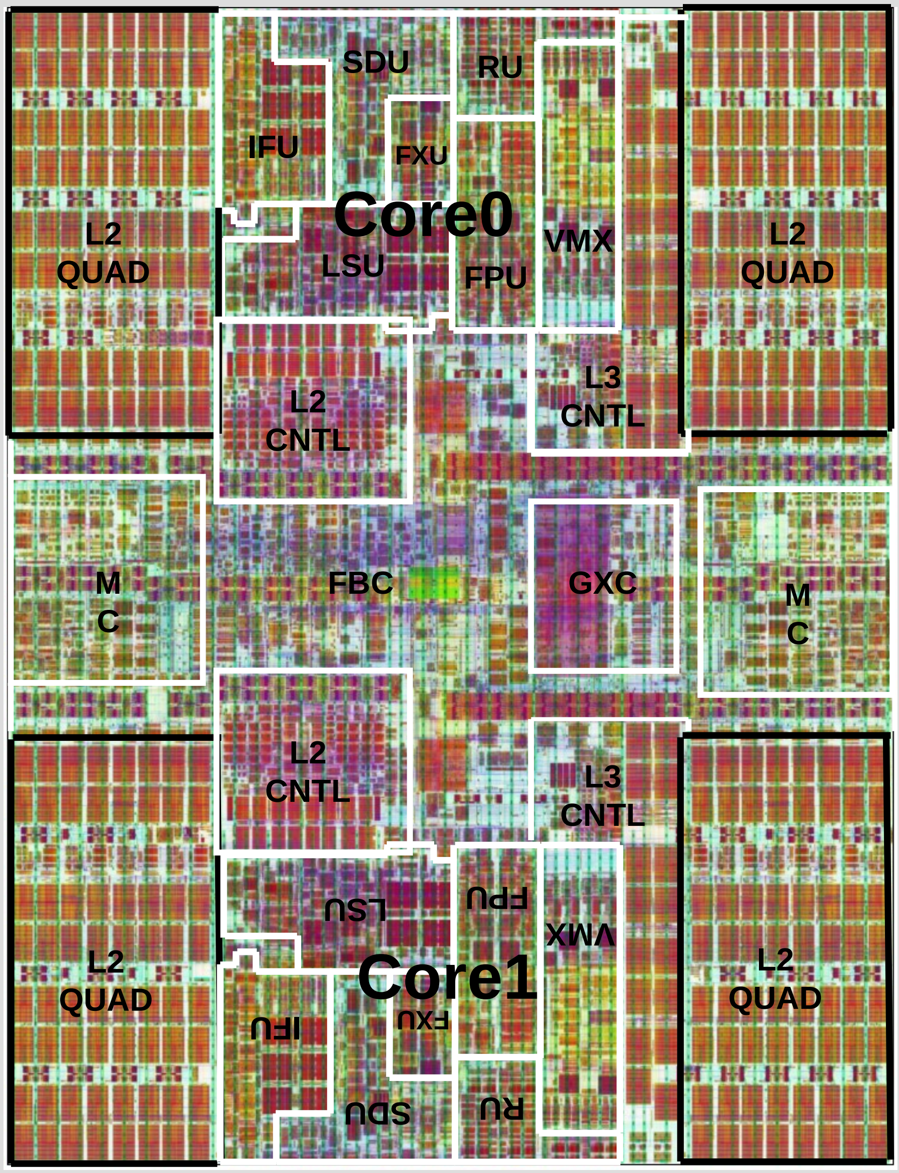
Appears as four CPUs per chip to the operating system (AIX V5.3 and Linux)



- 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**

HPC Performance

- **Collectively, HPC emphasizes (almost) everything in the microarchitecture**

- **Latencies are usually the bottleneck (as opposed to lack of a resource)**
  - Within the pipeline (recursive math)
  - From cache/memory – cache misses
  - From other processors – interventions

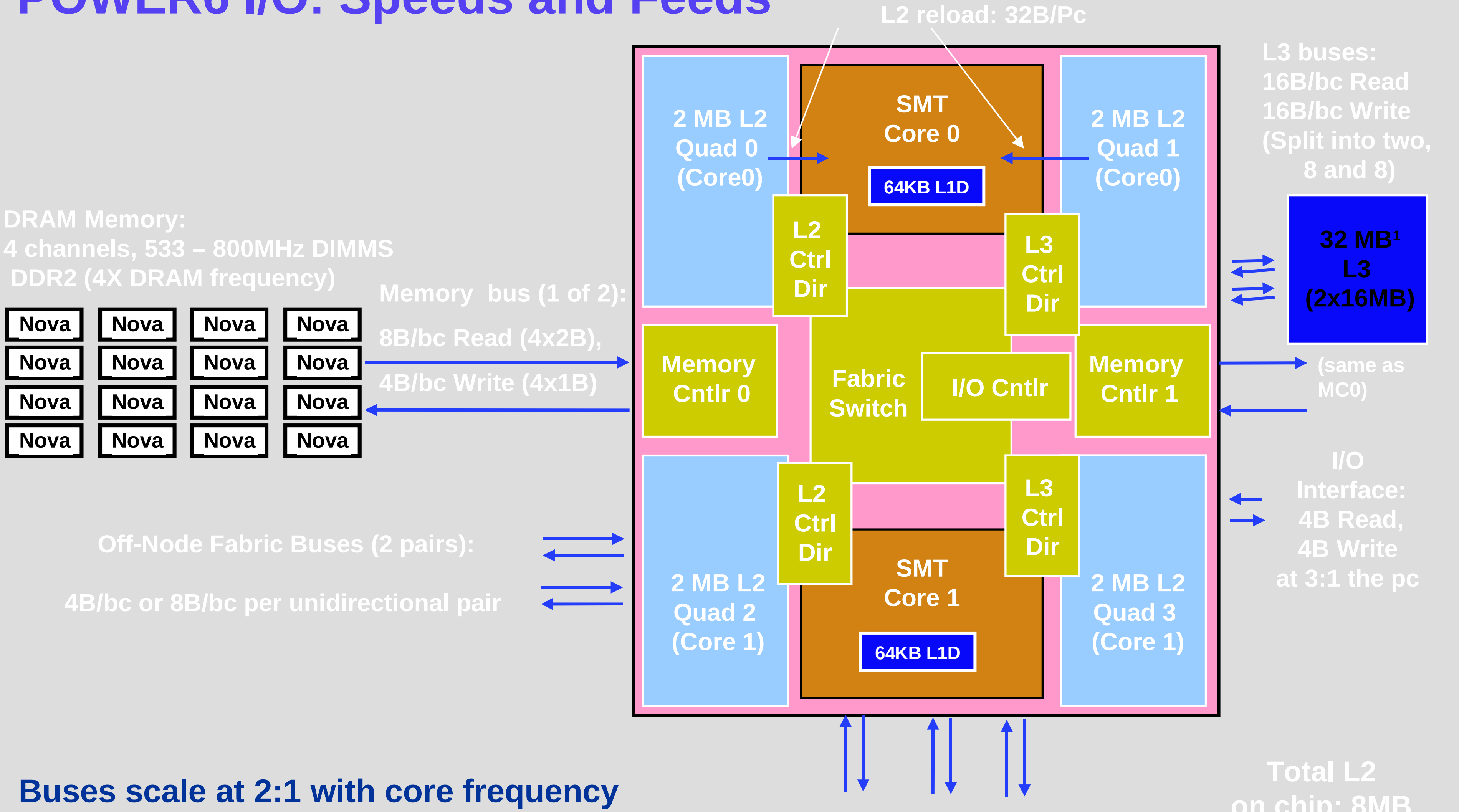
- **Trade-offs abound and everything matters (from each pipeline stage to the application, compiler, and developer)**

- **SMT helps to fill in the holes**

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 I/O: Speeds and Feeds



Buses scale at 2:1 with core frequency

pc = processor clock
bc = bus clock
2 pc = 1 bc

¹May be a single 32MB L3 chip with 8B buses

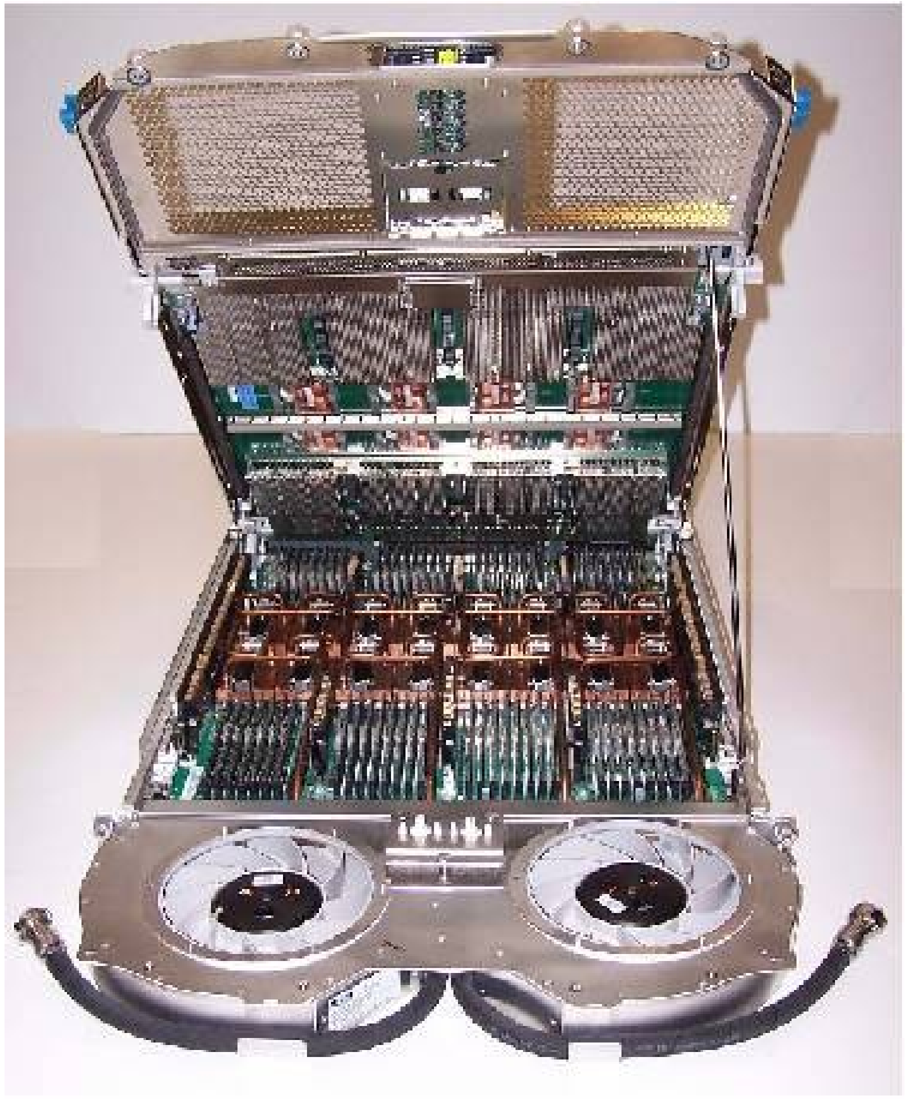
POWER6 p575 Node

AIX 6 AIX V5.3 Linux

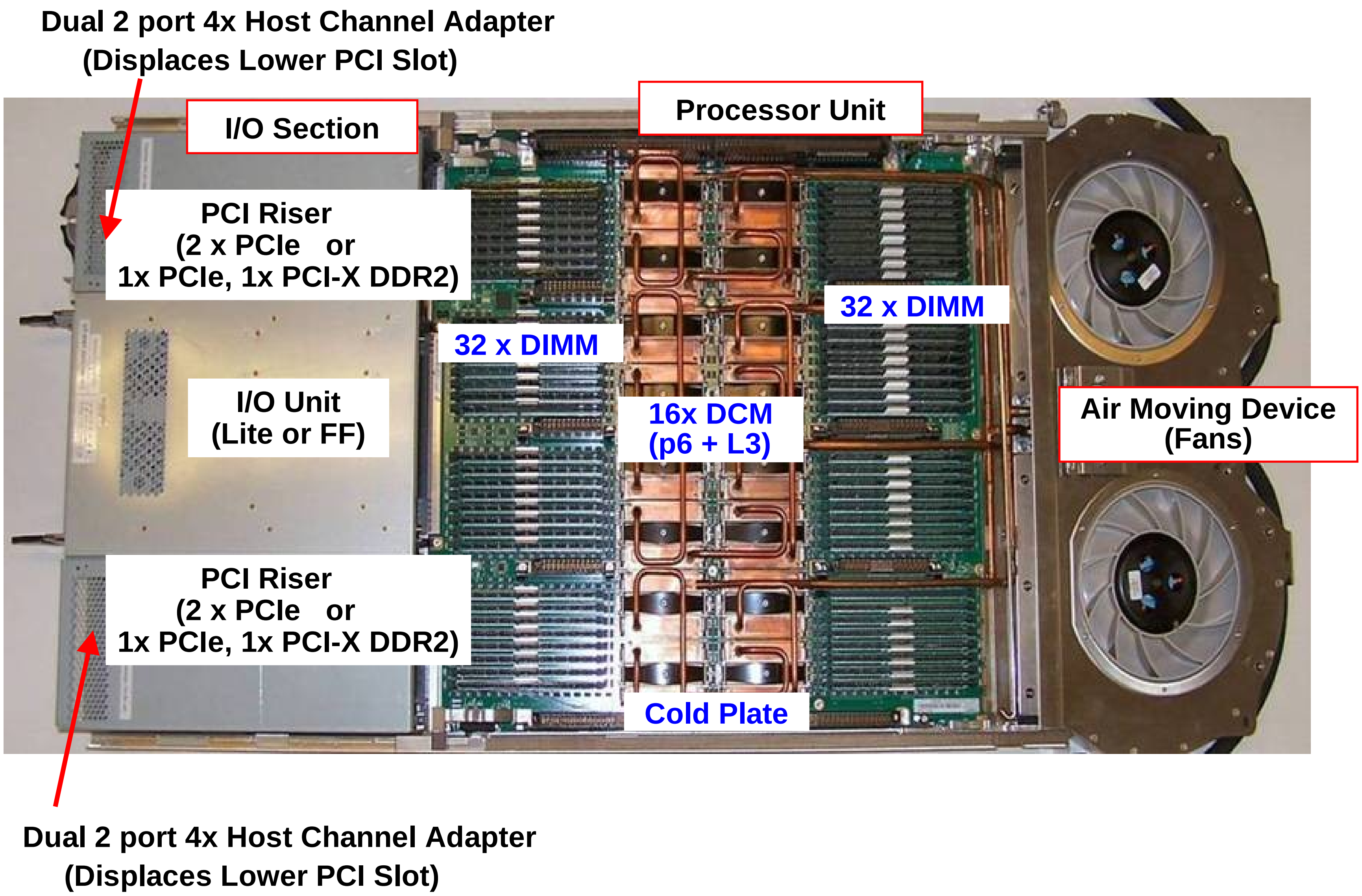
Compute Node	
Architecture	32-core node 1 – 14 nodes / rack (448 Cores) 4.7 GHz
Cache	L3: 32MB / chip
DDR2 Memory	4 to 256 GB (Buffered)
DASD / Bays	2 SAS DASD (2.5”)
Expansion	PCIe / PCI-X support
IVE	Yes
Integrated SAS	Yes
Expansion Slots	Dual GX Bus Adapters
Integrated Ethernet	Two Dual 10/100/1000 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)

POWER6 575 Water Cooled Node

Photo of p575 Mechanical Model

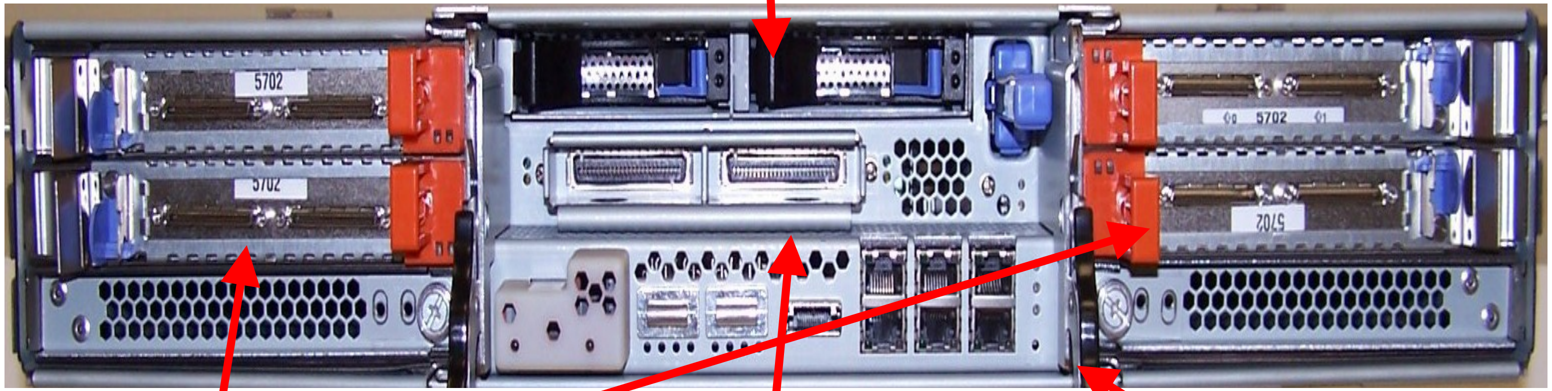


Top View


Dual 2 port 4x Host Channel Adapter
(Displaces Lower PCI Slot)
I/O Section
Processor Unit
PCI Riser
(2 x PCIe or
1x PCIe, 1x PCI-X DDR2)
32 x DIMM
32 x DIMM
I/O Unit
(Lite or FF)
16x DCM
(p6 + L3)
Air Moving Device
(Fans)
PCI Riser
(2 x PCIe or
1x PCIe, 1x PCI-X DDR2)
Cold Plate
Dual 2 port 4x Host Channel Adapter
(Displaces Lower PCI Slot)

P6 p575   Rear View…

SAS DASD
- Dual Drives: 73 or 146 GB



I/O
- PCIe   0 / 2 / 4 slots
- PCI-X 0 / 2 slots
- Expansion slots
  - Two GX++ slots

IO Interconnect
- 12X
- Single IO Drawer

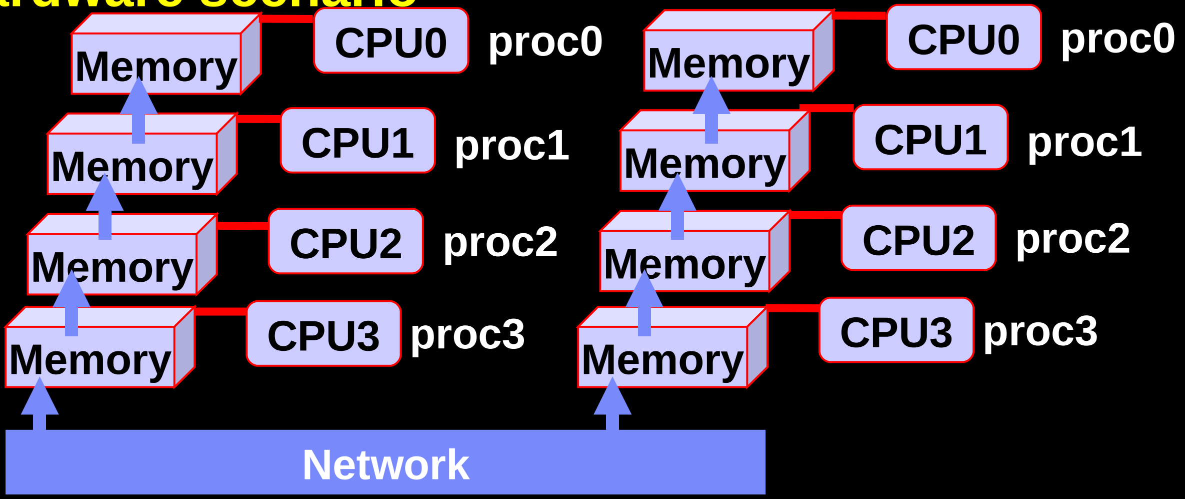
Ethernet Support (IVE)
- Two Dual 10/100/1000
- Optional Dual 10Gb

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



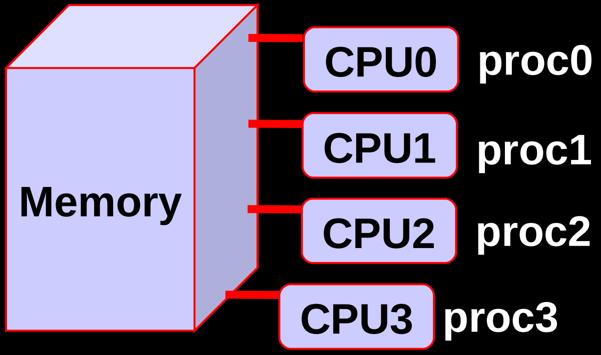
Distributed Memory Program Architecture
New characteristics in late 1990s

- 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

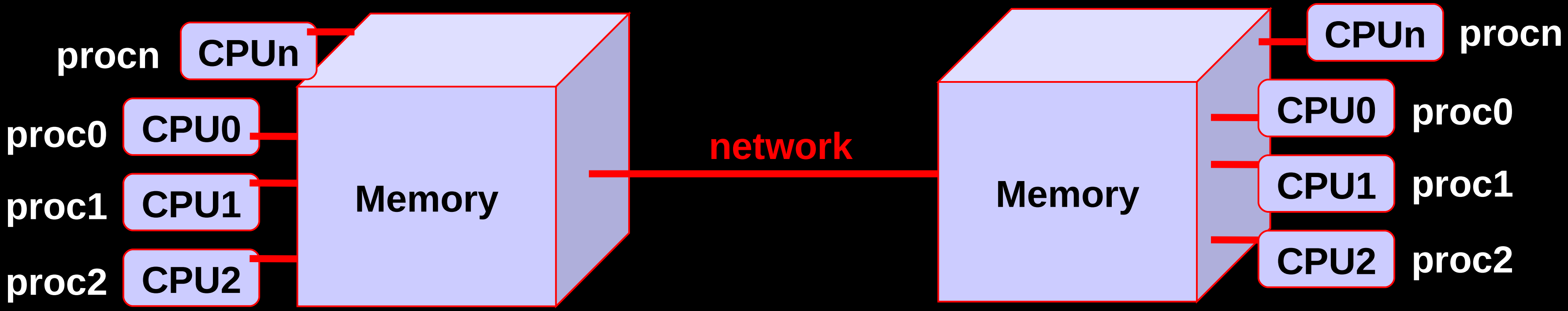
```
export MP_SHARED_MEMORY=yes
```



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

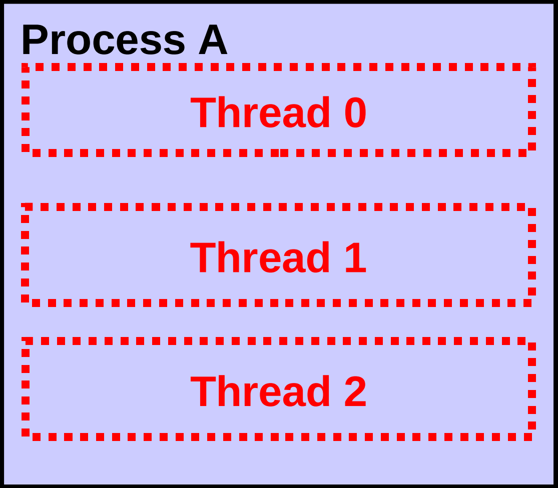
- We can still use shared memory for message passing between processes within each SMP system
- There's no need to change MPI standard for this scenario



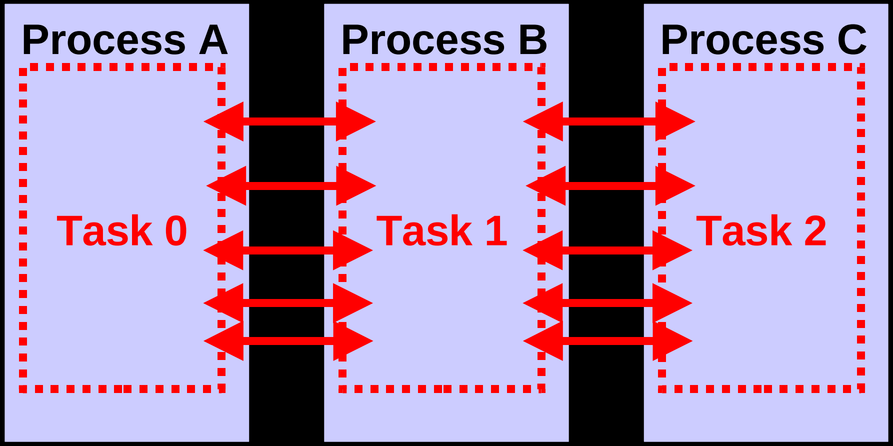
User shared-memory for message passing

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

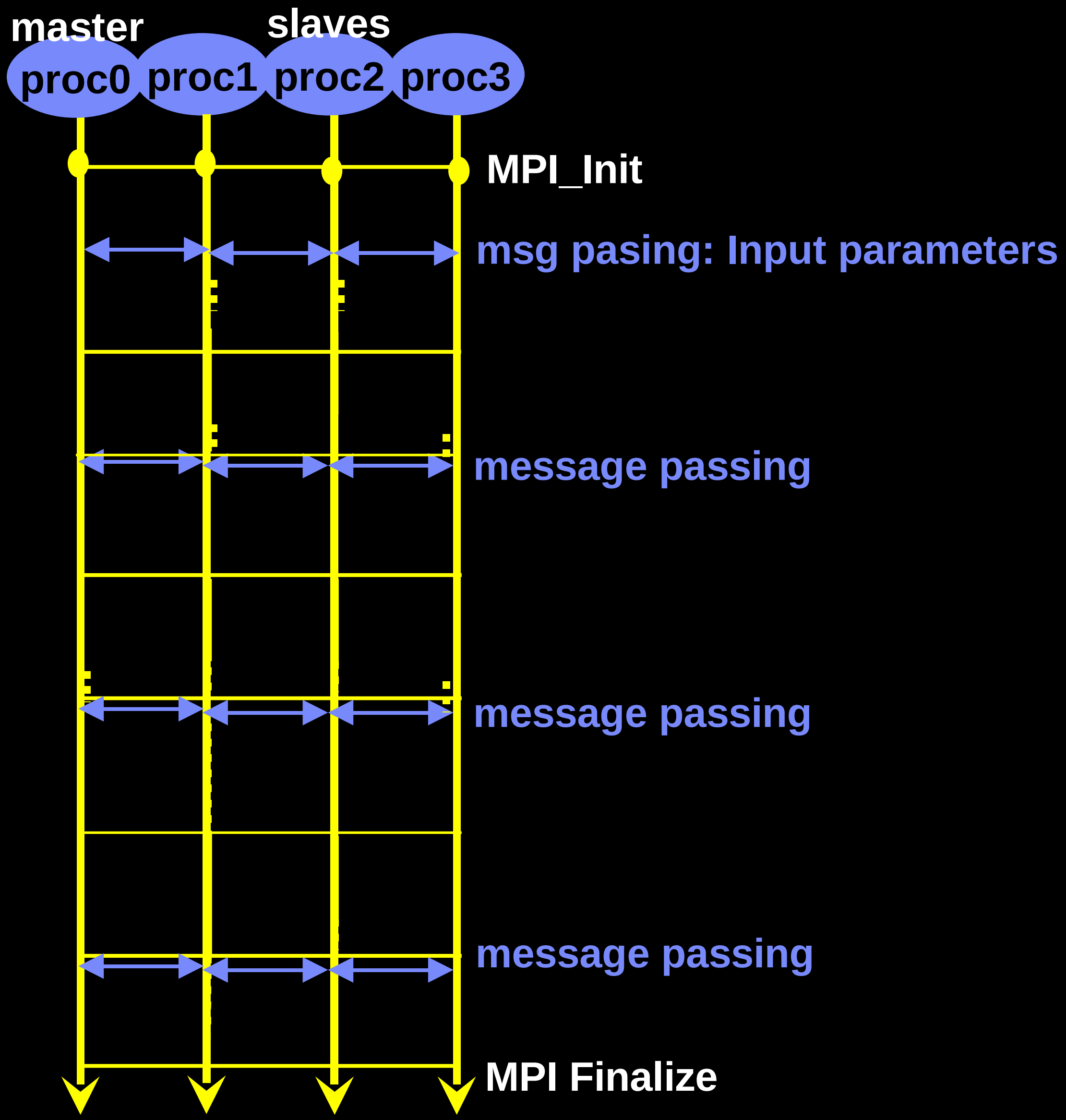
- Multiple steps in applying MPI technology to solve a Sci&Eng problem
  - 1. Divide workload to multiple processes (domain decomposition)
  - 2. Execute your MPI program
  - 3. Collect and process the output data
- Questions
  - Which filesystem should I use for my input, scratch and output files? What 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: FLENT, 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.

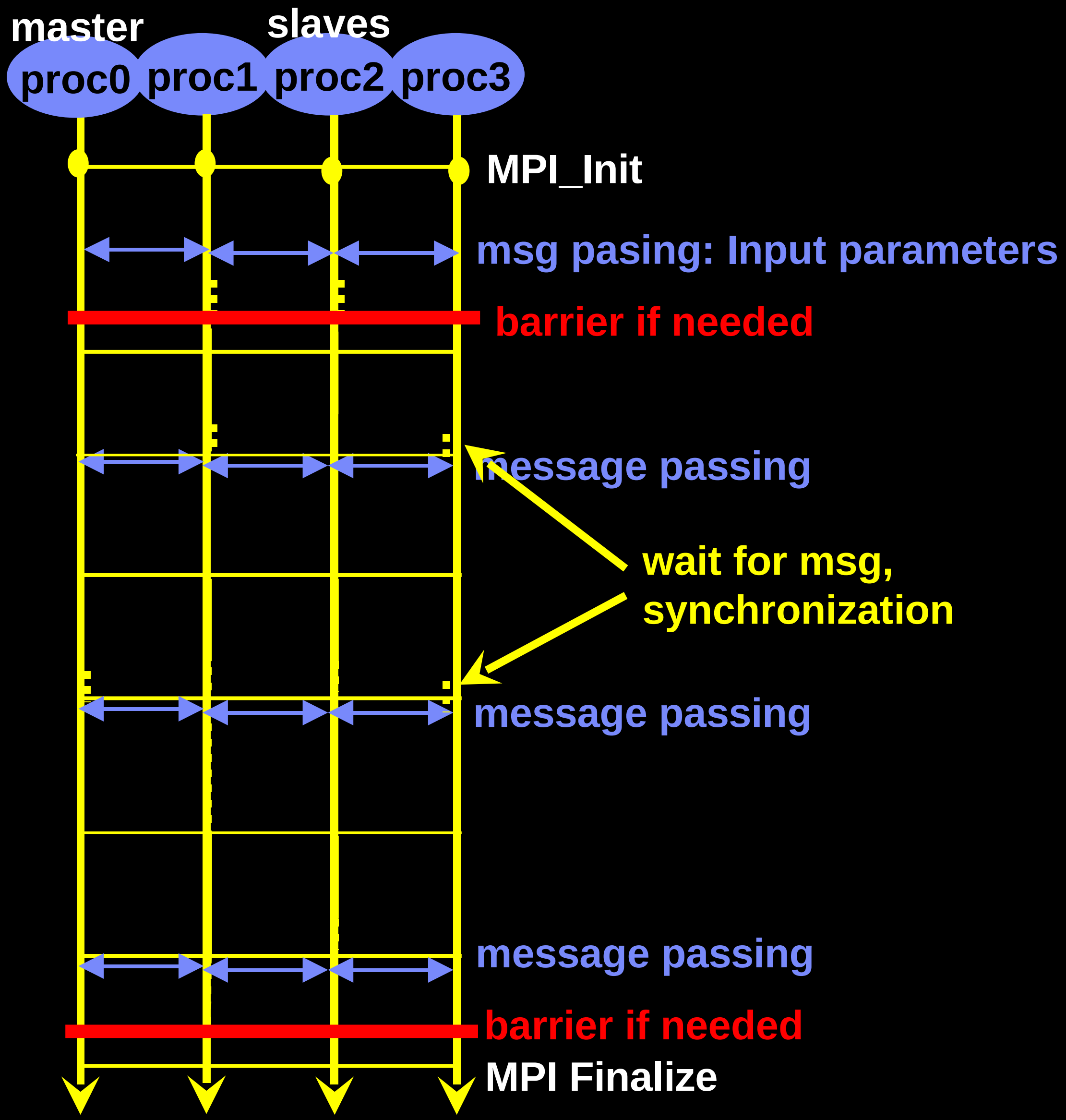
Schematic Flow of an MPI Code

```
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
```

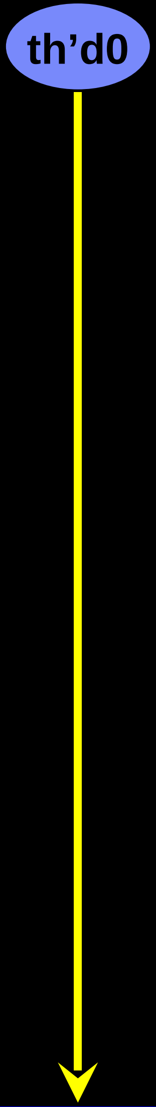


Schematic Flow of an MPI Code

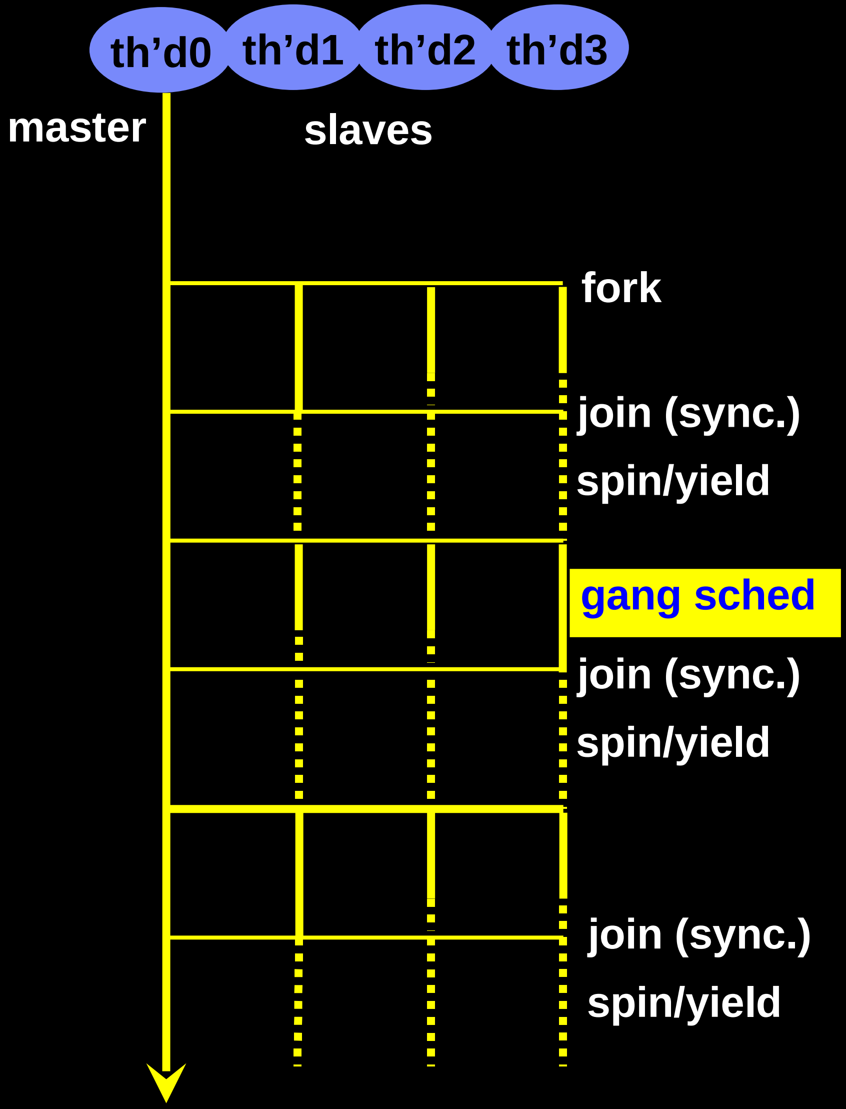
```
Program ParallelWork

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



```
Program ParallelWork

start the program
read input
set up comp. parameters
initialize variable
…
DO i=1,imax
…
…
End do
…
serial work
…
DO i=1,imax
…
…
End do
…
serial work
…
DO i=1,imax
…
…
End do
…
Output

End program ParallelWork
```



MPI options

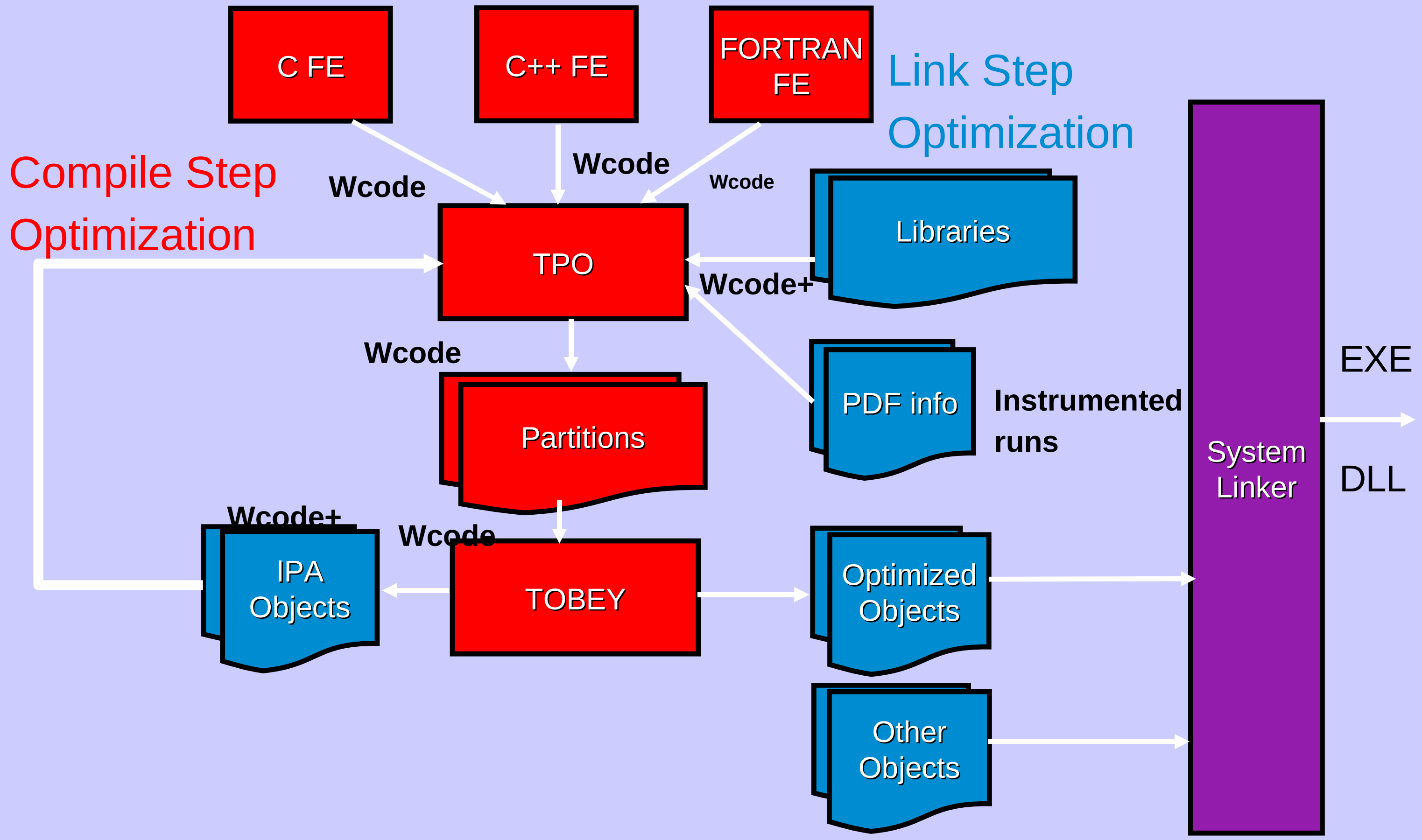
- IBM Parallel Environment
  - POE
  - Highly optimized for IBM processors, adapters, and networks
  - Have to purchase license
- MPICH
  - Uses TCP/IP protocol
  - Free
- LAM MPI
  - Free
- OpenMPI
  - Free, but new…
- etc., etc.

Software Environment

(brief Interlude - go to SW Env 02)

Some Useful System Commands

IBM XL compiler architecture



Some Environment Hints

- **If you get mpicc:**
  - 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 mpirun for a different version of MPI.**
  - Give the command

    ```
    which mpirun
    ```

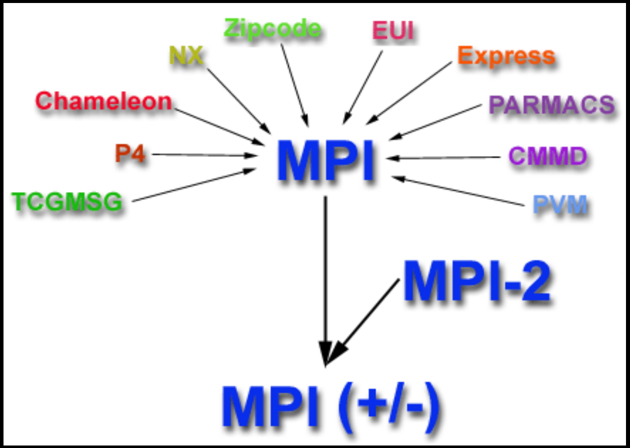
    - Make sure that the PATH given matches the one that cooresponds to the MPI implementation that you are using.

Distributed memory Programming: Message Passing Interface (MPI)

What Is MPI?

- **<u>Message Passing Interface (MPI)</u>:**
  - A specification for message passing libraries, designed to be a standard for distributed memory, message passing, parallel computing.
- **The goal of the Message Passing Interface:**
  - provide a widely used standard for writing message-passing programs.
  - establish a practical, portable, efficient, and flexible standard for message passing.
- **The MPI standard can be obtained from http://www-unix.mcs.anl.gov/mpi/standard.html**

Historical Development of MPI



- **1980-early 1990: distributed memory parallel computing application develops and calls for a standard**
- **1992: MPI Forum established**
- 1993: draft MPI standard presented at SC'93
- May, 1994: MPI-1 final version released, 115 routines defined
- 1996; MPI-2 finalized, which picked up "difficult" issues that MPI-1 intentionally left off.
- Most vendors have full implementation of MPI-1, but partial implementation of MPI-2

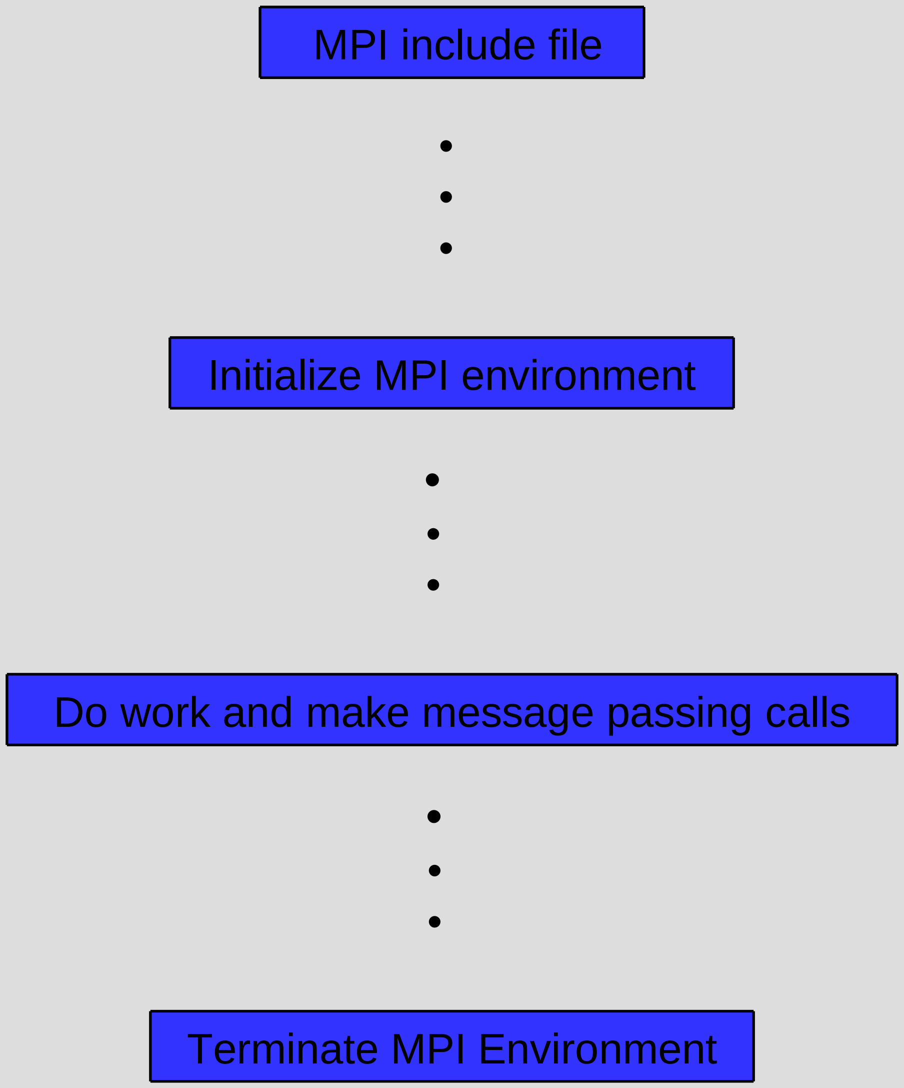
Reasons for using MPI

- **Standardization - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms.**
- **Portability - there is no need to modify your source code when you port your application to a different platform which supports MPI.**
- **Performance - vendor implementations should be able to exploit native hardware features to optimize performance.**
- **Functionality (over 115 routines in MPI-1, more in MPI-2)**
- **Availability - a variety of implementations are available, both vendor and public domain.**

General Remarks

- Target platform is a distributed memory system including massively parallel machines, SMP clusters, workstation clusters and heterogenous networks.
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing the resulting algorithm using MPI constructs.
- The number of tasks dedicated to run a parallel program is static. New tasks can not be dynamically spawned during run time. (MPI-2 is attempting to address this issue).
- Able to be used with C and Fortran programs in MPI-1. C++ and Fortran 90 language bindings are in MPI-2.

Overview of MPI Program Structure


MPI include file
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment

Communicators and Groups

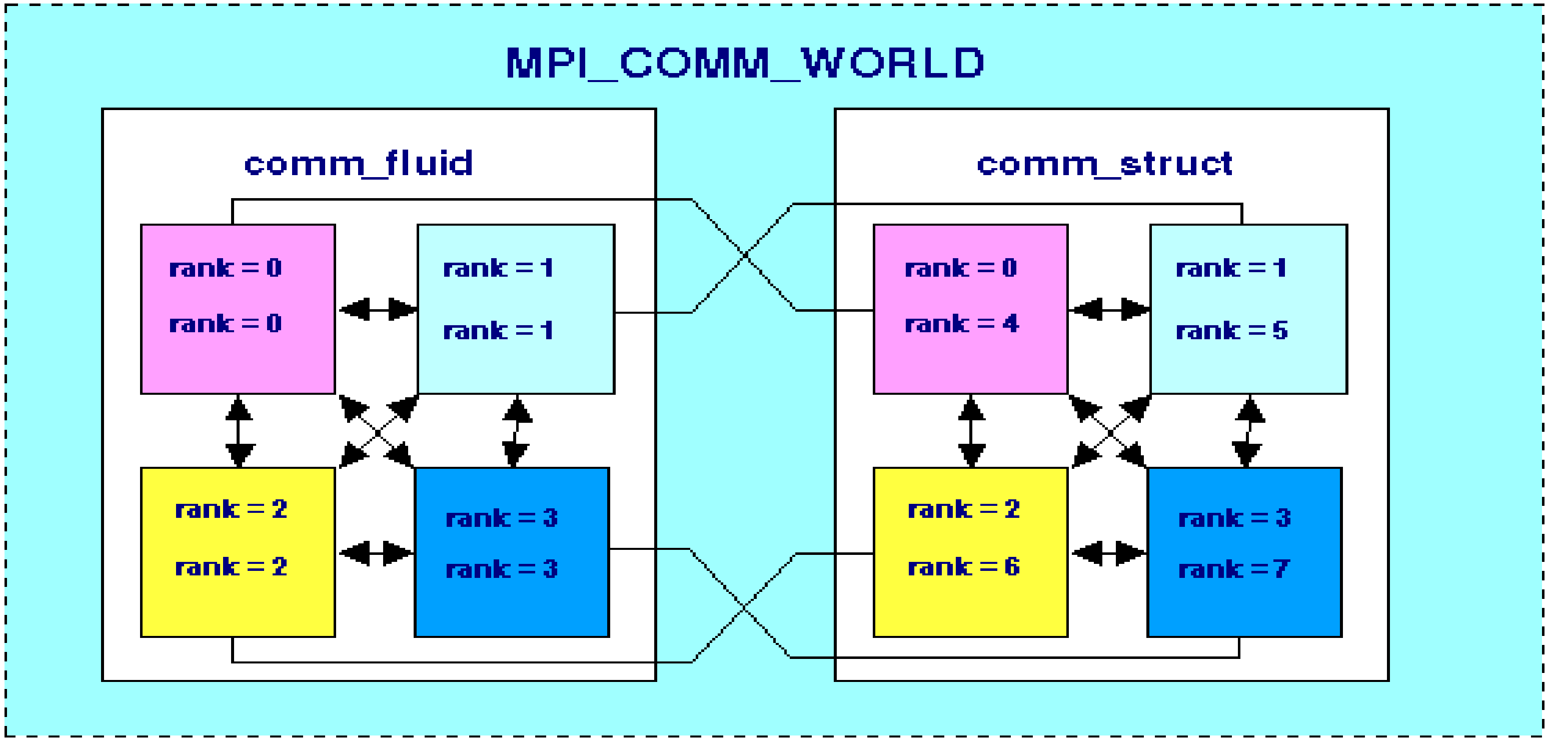
- **MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument.**
- **Simply use MPI_COMM_WORLD whenever a communicator is required - it is the predefined communicator which includes all of your MPI processes.**

Rank

- Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes.
- A rank is sometimes also called a "process ID".
- Ranks are contiguous and begin at zero.

- Programmer uses to specify the source and destination of messages.
- Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).

Multiple Communicators



Describe (briefly) 3 Classes of MPI Routines

- **Environment Management Routines - setup and query the environment**
- **Point to Point Communication Routines - provide message passing between 2 processors**
- **Collective Communication Routines - involve all processors in scope of communicator**

- **(Other routines but another session)**

MPI has many routines – focus on a few

- **Many routines in MPI - may seem overwhelming - all operations can be reduced to a much smaller set of primitives.**
- **These primitives should be the focus of a first exposure to MPI**
  - MPI_Init MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Isend
  - MPI_Irecv
  - MPI_Iprobe
  - MPI_Test

General MPI Program Structure

MPI include file

Initialize MPI environment

Do work and make message passing calls

Exit MPI

```
program hello
implicit none
include 'mpif.h'
integer   ::       myrank, nprocs, n, islave, master
integer   ::       status(MPI_STATUS_SIZE)
integer   ::       ierr, resultlen, tag
character (LEN=MPI_MAX_PROCESSOR_NAME) :: hostname
!-----------------------------------------------------
call MPI_Init(ierr)
call MPI_Comm_Rank(MPI_COMM_WORLD, myrank, ierr)
call MPI_Comm_Size(MPI_COMM_WORLD, nprocs, ierr)
call MPI_Get_processor_name(hostname, resultlen, ierr)
call MPI_Barrier (MPI_COMM_WORLD, ierr)
!-----------------------------------------------------
write (*,*) "Hello! --- Rank ", myrank, " out of ", nprocs,  &
  " processes running on ", hostname(1:index(hostname,".")-1)
!-----------------------------------------------------
call MPI_Barrier (MPI_COMM_WORLD, ierr)
write (*,*) "slave ", myrank, ": ", t2-t1, " seconds"
call MPI_Finalize(ierr )

end program hello
```

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. */
}
```

Need to compile, link and execute -

- **Compile & Link:**
  - mpcc -o minim minim.c
- **Via Loadlever, submit for execution:**
  - llsubmit minim.cmd
- **Execution line:**
  - mpirun -np 512 -cwd `pwd` -exe minim.x

Summary: MPI Subroutines

Functionality category	Functionality Description	Examples
Environment management & misc.	Env. Initialization, termination, etc	MPI_Init, MPI_Wtime, MPI_Error_class, MPI_Finalize MPI_Pack, MPI_Pcontrol
Point-to-point communication - Blocking	Msg passing between two different tasks - Success acknowledged	MPI_Send, MPI_Recv MPI_Ssend, MPI_Bsend MPI_Buffer_attach, MPI_Wait
Point-to-point communication - Non-Blocking	Msg passing between two different tasks - Success not acknowledged	MPI_ISend, MPI_IRecv MPI_ISsend, MPI_IBsend MPI_Testsome, MPI_Iprobe
Collective Communication	Communication involving all processes	MPI_Barrier, MPI_Bcast, MPI_Gather, MPI_Scatter,
Group and Communicator Management	Managing MPI group, a set of ordered processes which is always associated with a communicator object	MPI_Group_incl, MPI_Group_union, MPI_Comm_group, MPI_Comm_size, MPI_Comm_rank, MPI_Intercomm_merge
Derived Types	Create data types for other MPI routines to use	MPI_Type_size, MPI_Type_struct MPI_hvector, MPI_Type_lb
Virtual Topology	Mapping/ordering MPI processes into a geometric "shape	MPI_Cart_create, MPI_Cart_shift MPI_Graph_create, MPI_Graph_get

Calling MPI Routines in C or Fortran

- Include file: required for all programs/routines which make MPI library calls

C include file	Fortran include fle
# include “mpi.h”	Include ‘mpif.h’

- Format of MPI calls

Language	C	Fortran
Format	Rc=MPI_Xxxxx(parameters, …)	CALL MPI_Xxxxx(parameter, …, ierr)
Example	rc=MPI_Bsend(&buf,count,type,dest,tag,comm)	CALL MPI_Bsend(buf,count,type ,dest,tag,comm,ierr)
Error code	Returned as “rc” or MPI_SUCCESS	Returned as ierr paremeter, or MPI_SUCCESS

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
```

Invoking the MPI Compiler

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

Simple example program

```
program  hej

  IMPLICIT NONE
  include "mpif.h"

  character(LEN=MPI_MAX_PROCESSOR_NAME):: name
  character(LEN=MPI_MAX_PROCESSOR_NAME), &
       allocatable :: all_names(:)
  integer:: rank, nproc, lname, ierr, i

  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
```

```
 call MPI_Get_processor_name(name,lname,ierr)
 allocate(all_names(nproc))

 call MPI_Gather(name,len(name), MPI_CHARACTER,  &
      all_names,len(all_names(1)), MPI_CHARACTER,  &
      0, MPI_COMM_World, ierr)

 if(rank==0)then
   write(*,'(i4,": ",a)') (i-1,trim(all_names(i)),
   i=1,nproc)
 endif

 call MPI_Finalize(ierr)
end program
```

Example program execution

```
Starting program at Sun Aug 26 17:18:37 CEST
   2007

   Using /bgl/BlueLight/ppcfloor/bglsys/bin/mpir
   un -shell /pdc/vol/openssh/4.5p1/bin/ssh
   -verbose 1 -cwd /gpfs/scratch/s/smeds/test
   -mode VN -env BGLMPI_MAPPING=TXYZ -env
   MPIP='-c' /gpfs/scratch/s/smeds/test/hej-
   traced "arg 1"  "arg 2"
...
<Aug 26 17:18:38.878163> FE_MPI (Info) : Waiting
   for job to terminate
mpiP: Found MPIP environment variable ['-c']
mpiP: mpiP V3.1.1 (Build Aug 21 2007/15:21:35)
mpiP: Direct questions and errors to mpip-
   help@lists.sourceforge.net
 0: Processor <0,0,0,0> in a <4, 4, 2, 2> mesh
 1: Processor <0,0,0,1> in a <4, 4, 2, 2> mesh
...
62: Processor <3,3,1,0> in a <4, 4, 2, 2> mesh
63: Processor <3,3,1,1> in a <4, 4, 2, 2> mesh
```

```
mpiP:
mpiP: Storing mpiP output in [./Unknown.
   64.0.1.mpiP].
mpiP:
<Aug 26 17:18:44.400667> BE_MPI (Info) : Job
   3350 switched to state TERMINATED ('T')
<Aug 26 17:18:44.400704> BE_MPI (Info) : Job
   successfully terminated
...
<Aug 26 17:18:44.970018> FE_MPI (Info) : == Exit
   status:   0 ==
Program finished Sun Aug 26 17:18:44 CEST 2007
    Program exit code: 0
```

MPI Performance Considerations
Dive Deeper …

- Dive Deeper into MPI
- Terminology – MPI performance related
- Factors affecting MPI performance
- IBM Environment variables that may improve performance

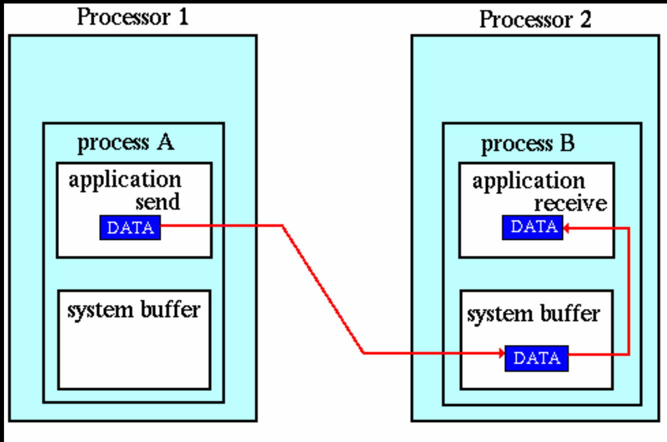
Dive Deeper into MPI: P-to-P communications

- Two MPI tasks send ←→ receive messages
- Multiple types of send and receive routines
  - Synchronous send
  - Blocking send/blocking receive
  - Non-blocking send/non-blocking receive
  - Buffered send
  - Combined send/receive
  - “Ready” send
- Any type of send can be paired with any type of recv
- Several routines associated with send-receive operations

Dive Deeper into MPI P-to-P Communications: Need for Buffering

- Only in perfect world, every send is perfectly in sync with its matching receive
- Need for buffer:
  - Scenario 1: a send operation occurs 5 sec before the receive is ready – where to place the message?
  - Scenario 2: multiple sends arrive at the same receiving task which can only accept one send at a time – where to place the backing up messages
- Vendor implement solutions for these situations using system buffer. This is not defined by MPI standard.

Dive Deeper into MPI P-to-P Communications: System Buffer and Application Buffers



Path of a message buffered at the receiving process

- **System buffer**
  - **Allows asynchronous send-receive, thus may improve performance**
  - **Managed entirely by the MPI library, opaque to programmer**
  - A finite resource that can be easy to exhaust
  - Often mysterious and not well documented
  - May exist on sending side, receiving side, or both
  - Help to improve program performance
- Application buffer – user managed address space

Dive Deeper into MPI P-to-P Communication: Blocking vs. Non-blocking

- Blocking send
  - will only return after it is safe to modify the application buffer (your send data)
    - Safe: modifications will not affect the data intended for the receiving task
    - Safe: no guarantee the data was actually received
  - can be synchronous – handshaking occurred with the receive task to confirm a safe send
  - can be asynchronous when system buffer is used to hold the data for eventual delivery
- Blocking receive
  - only returns after the data has arrived and is ready for the program to use
- Non-blocking send and receive
  - Simply request the MPI library to perform the operation when it is able.
  - Return almost immediately, without wait for any communication events to complete
  - User is responsible to know when the application buffer is safe to be modified – use wait routines when needed
  - Use no-blocking routines to overlap communication with computation. – to explore possible performance gains.

Dive Deeper into MPI P-to-P Communication: Order and Fairness

- Order: MPI guarantees that messages will not overtake each other
  - Between a pair of sender and receiver, “first” message sent will arrive first, and “first” requested message will arrive first.
  - Order rule does not apply when more than 2 tasks participate in communications
- Fairness:
  - MPI does not guarantee fairness.
  - Programmer needs to prevent “racing condition”: if task0 and task1 both send a competing message to task2, only one send will complete.

Dive Deeper into MPI: Collective Communication

- Must involve all processes in the scope of communicator
- Types of collective operations
  - Synchronization – i.e. barrier,
  - Data movement – i.e. broadcast, scatter/gather, all to all
  - Reduction – i.e. one member collects data from the others and performs an operation (min, max, add, etc)
- Restrictions
  - Collective op. are blocking
  - No message tag argument needed
  - To apply collective op. to a subset of processes, first partition the subset into new groups and attach the new groups to new communicators.
  - Can only be used with MPI predefined data types, not with MPI derived data types.

Dive Deeper into MPI: Predefined Primitive MPI Data Types

Data Type	C	Fortran
Signed char	MPI_CHAR	MPI_CHARACTER
Signed short int	MPI_SHORT	
Signed int	MPI_INT	MPI_INTEGER
Signed long int	MPI_LONG	
Unsigned char	MPI_UNSIGNED_CHAR	
Unsigned short int	MPI_UNSIGNED_SHORT	
Unsigned int	MPI_UNSIGNED_INT	
Unsigned long int	MPI_UNSIGNED_LONG	
Float / Real	MPI_FLOAT	MPI_REAL
Double / Double Precision	MPI_DOUBLE	MPI_DOUBLE_PRECISION
Long double	MPI_LONG_DOUBLE	
complex		MPI_COMPLEX
Double complex		MPI_DOUBLE_COMPLEX
logical		MPI_LOGICAL
8 binary digits	MPI_BYTE	MPI_BYTE
Data packet or unpacket with MPI_Pack()/MPI_Unpack	MPI_PACKED	MPI_PACKED

Dive Deeper into MPI: Group and Communicator

- A group is an ordered set of processes
  - Each process has it unique integer rank, starting at 0 and goes to N-1
  - A group is represented A group is always associated with a communicator object
- A communicator is a group of processes that may communicate with each other
  - It can be considered as an extra “tag” required by every MPI calls.
- Both group and communicator are represented within system memory as objects, accessible to programmers only by “handles”.
  - The handle for the communicator that comprises all tasks is MPI_COMM_WORLD
- Primary purposes of group and communicator objects

Terminology – MPI Performance Related

- **Latency: overhead associated with sending a 0-byte message**
- **Bandwidth:**
- **Application buffer: user program space which holds the data that is to be sent or received**
- **System buffer: system address space for storing messages – need it to enable async communication**
- **Blocking communication: a communication is blocking if its completion depends on certain "events".**
- **non-blocking:**
- **Synchronous: a synchronous send operation is complete only after receiving acknowledgement from receiving process**
- **Asynchronous:**

Two Types of MPI communications

- Point-to-point communication routines: for data exchange between a send task and a receive task.
  - Blocking (7 routines)
  - Non-blocking (5)
  - Persistent communications (7)
  - Completion/testing (4)
- Collective communication routines: for all tasks within the communicator participate in a communication operation
  - All are blocking (14)
  - MPI-2 specifies non-blocking corollaries for these routines.

Factors Affecting MPI Performance

- Platform / Architecture Related
  - CPU, memory subsystem, Network adapters, OS,
- Network related
  - Hardware, Protocols, configurations, network tuning, network contention
- Application related
  - Algorithm, communication/computation ratio, load balance, memory usage pattern, IO, message size used, types of MPI routines used
- MPI implementation related
  - Message buffering
  - Message passing protocols – eager, rendezvous, order
  - Send-receiving synchronization – polling, interrupt
  - Routine internals – efficiency of algorithm used to implement routines

Message Buffering

- A temporary space to store the data being sent to receiver
  - System buffer: provided by the system and not visible by the user. MPI standard is purposefully vague.
  - User buffer: explicitly declared and managed by the programmer
- 4 ways to implement standard send
  - 1. buffer at the sending side
  - 2. buffer at the receiving side
  - 3. no buffer at all.
  - 4. buffer under some condition and not others. i.e. eager vs. rendezvous protocols.
- Using user buffer:

• MPI_Buffer_attach	- allocates user buffer space
• MPI_Buffer_detach	- frees user buffer space
• MPI_Bsend	- buffer send, blocking
• MPI_Ibsend	- buffer send, non-blocking

- Advantage: permits comm. to be asynch with computation.
- Disadvantages:
  - Buffer exhaustion/overflow can cause program failure
  - It can be hard for programmer to know when and how to use buffer
  - Potential portability problem if

Introducing MPI Message Passing Protocols Eager vs. Rendezvous

- These are internal methods and polices an MPI implementation employs to accomplish message deliver. No MPI standard here.
- Two common protocols:
  - 1. Eager – an asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive
  - 2. Rendezvous – a synchronous protocol that requires an acknowledgement from a matching receive
- An implementation can use a combination of the 2 protocols: eager protocol for small messages, and rendezvous protocol for large message.
  - In conjunction with message buffer

MPI Message Passing Protocols
Eager

- Assumption:
  - Send process assumes the receiver can store the message if it is sent. It is receiver's responsibility to buffer the message upon arrival
  - implementation's guarantee of certain amount of buffer space on the receive process.
  - Used for smaller messages (upto KB), and small number of MPI tasks
- Advantage:
  - Reduce sync delay;
  - Simple programming: only need to use MPI_Send
- Disadvantages:
  - Not scalable. Larger buffer required when the number of senders increases
  - program error when receive buffer is exceeded
  - Buffer wastage when there's only small amount of msgs
  - Consume CPU cycle to copy data and manger the buffer

MPI Message Passing Protocols Rendezvous

- Used when assumption of Eager Protocol can not be made, or when Eager Limit is exceeded.
- Requires "handshaking" between sender and receiver:
  - 1. Sender process sends mgs envelope to destination process
  - 2. Envelop received and stored by destination process
  - 3. When buffer space is available, destination process replies to sender that requested data can be sent
  - 4. Sender receives reply and then sends data.
  - 5. Destination process receives data
- Advantages:
  - Scalable, robust, envelope can be small
  - May eliminate a data copy – user space to user space direct.
- Disadvantages:
  - Synch delay due to the handshaking requirement
  - More programming complexity

Default Eager Limits

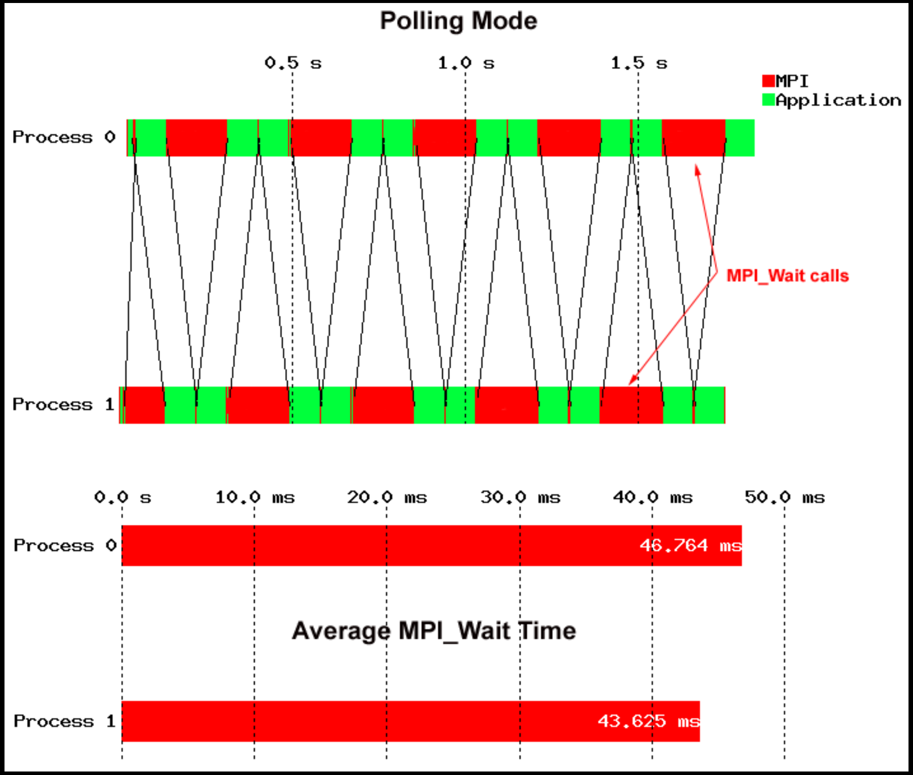
- Small Message (MP_EAGER_LIMIT)
  - Send header and message
- Large Message
  - Send header
  - Acknowledge
  - Send message

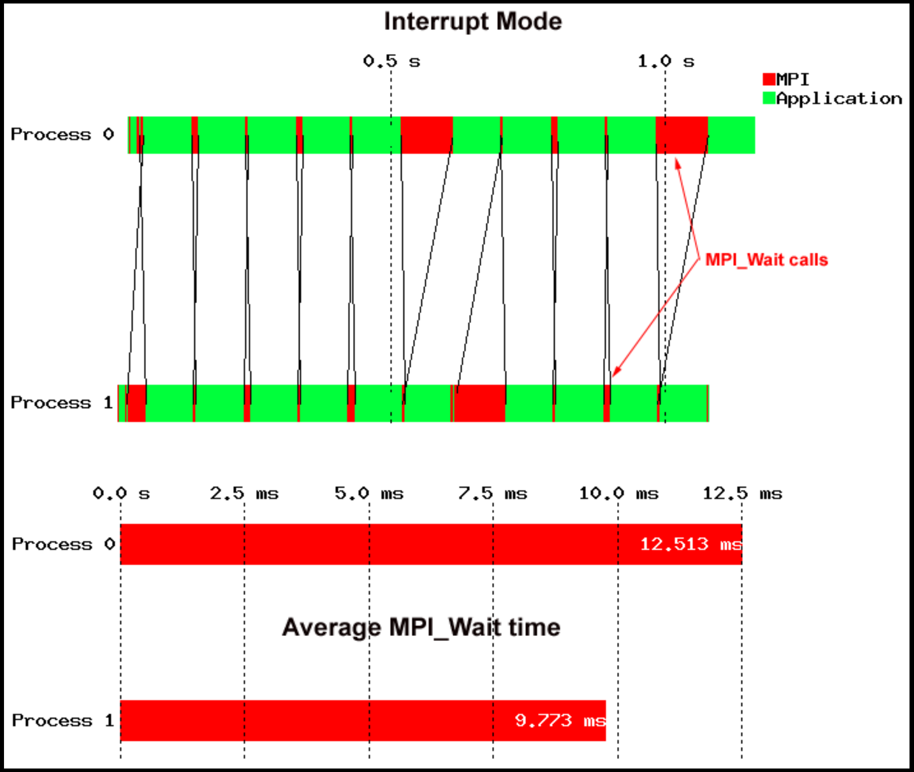
No. Tasks	MP_EAGER_LIMIT (default, bytes)
1 - 256	32768
257 – 512	16384
513 – 1024	8192
1025 – 2048	4096
2049 – 4096	2048
4097 - 8192	1024

Sender-Receiver Synchronization
polling or interrupt

- Cooperation between sending and receiving tasks are required for synchronous MPI communication
  - Polling mode: user MPI tasks check for and service communication events at regular intervals
  - Interrupt mode: user MPI tasks be interrupted by the system for communication events when they occur
- Interrupt mode has advantages in the following situations:
  - 1. apps that use nonblocking send or receive
  - 2. apps that have non-synchronized set of send or receive pairs
  - 3. apps that do not issuewaits for nonblocking send or receive

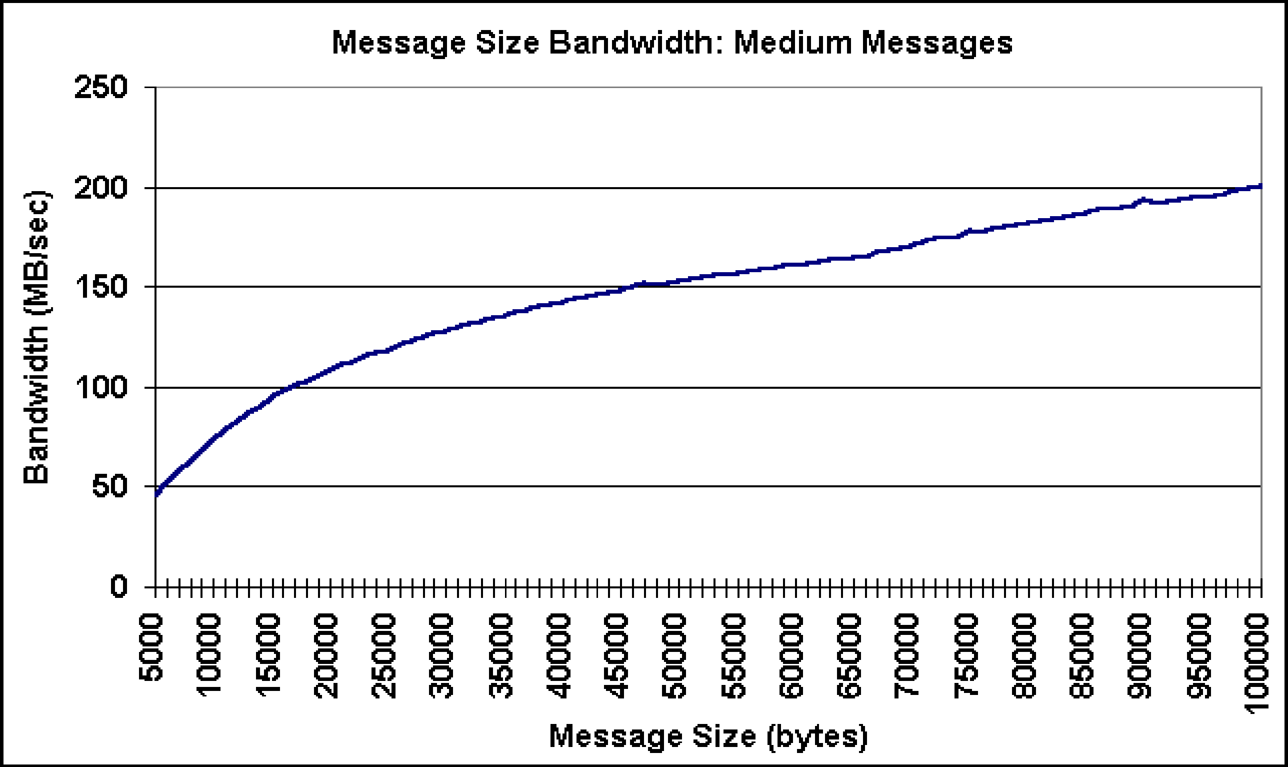
Example: Polling and Interrupt





Message Size

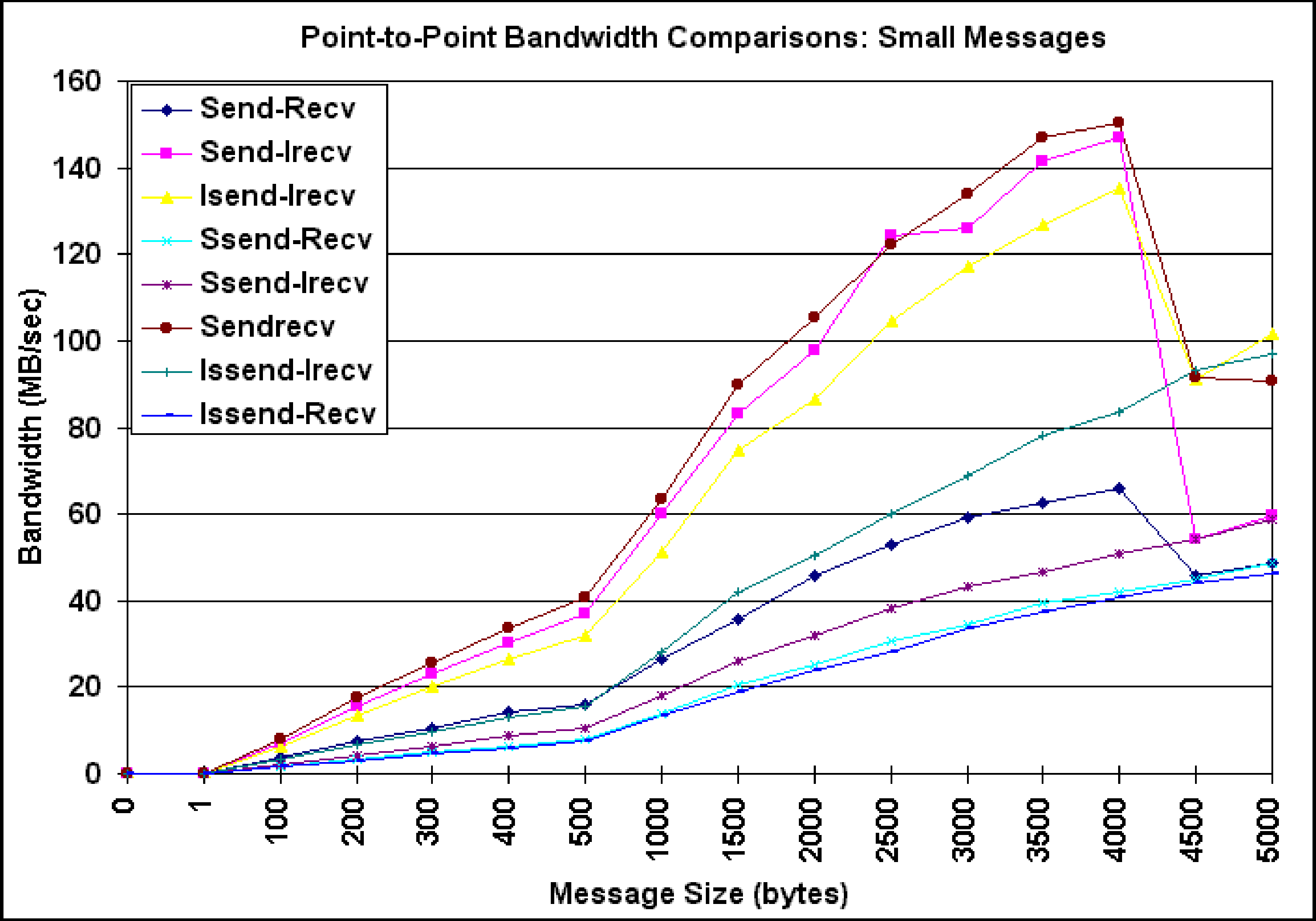
- Typically, increasing the message size will yield better performance.



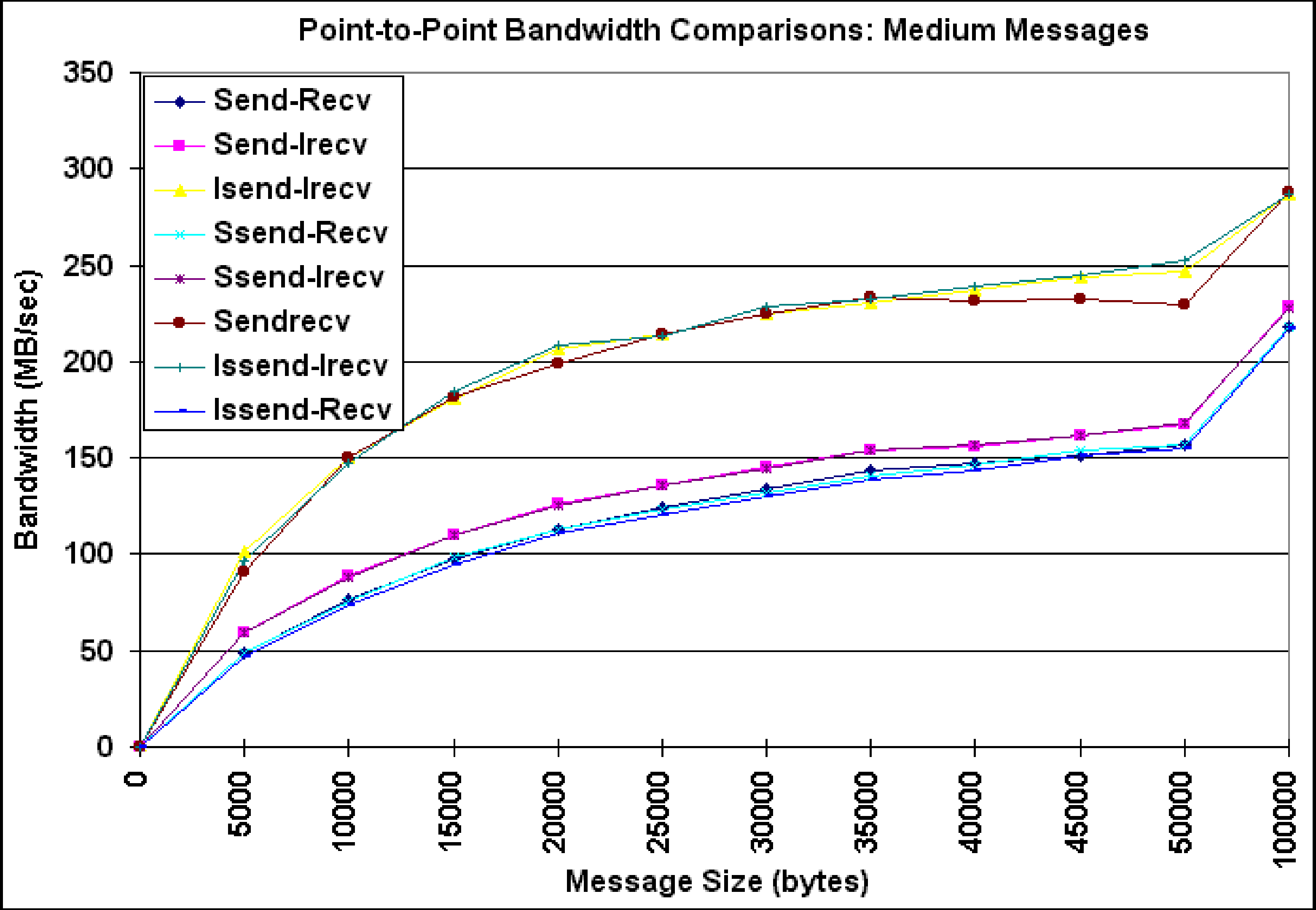
Point-to-Point Communications

- Many way to mix and match send and receive operations
- Send routines (match any receive, probe; non-blocking can match any completion/testing)
  - Blocking – standard, buffered, ready, sync
  - non-blocking – standard, buffered, ready, sync
  - Persistent – standard, buffered, ready, sync
- Receiving routines (match any send)
  - Blocking
  - Non-blocking
  - Persistent
- Probe routines (match any send)
  - Blocking
  - Non-blocking
- Completion/testing routines (match any non-blocking send/receive
  - Blocking – one, some , any, all
  - Non-blocking one, some, any, all

Point-to-Point bandwidth comparison: Small Messages



Point-to-Point bandwidth comparison: Small Messages

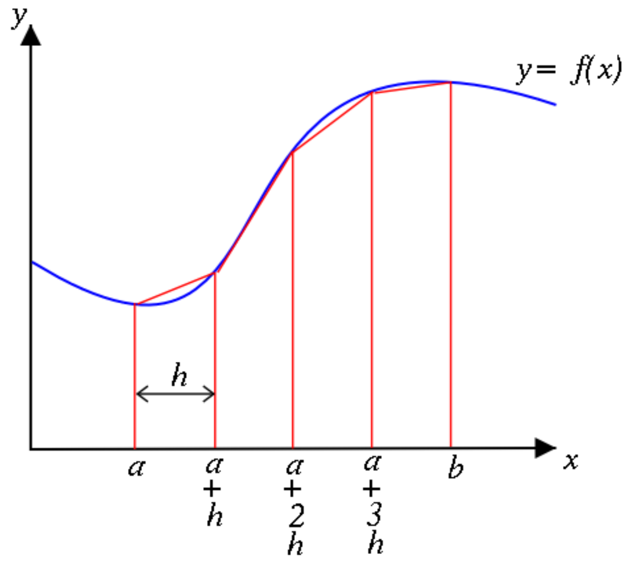


A first application

- Compute a numerical approximation to the definite integral

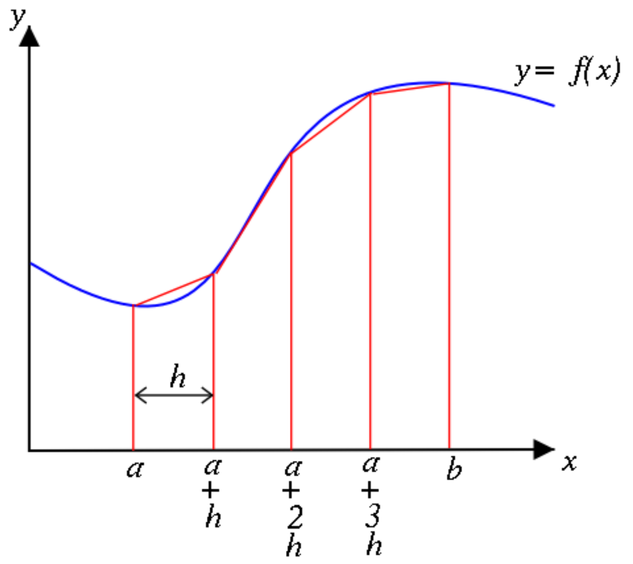
$$\int_a^b f(x)\,dx$$

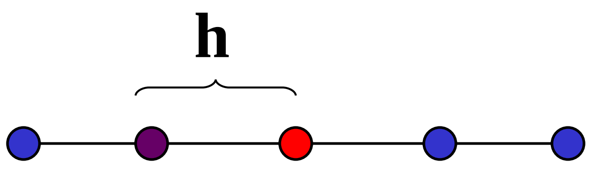
using the trapezoidal rule



How the trapezoidal rule works

- Divide the interval [a,b] into n segments of size h=1/n
- Approximate the area under an interval using a trapezoid
- Area under the ith trapezoid
  $\frac{1}{2}$ (f(a+i×h)+f(a+(i+1)×h)) ×h
- Area under the entire curve
  ≈ sum of all the trapezoids





a+i*h **a+(i+1)*h**

Reference material

- For a discussion of the trapezoidal rule

http://metric.ma.ic.ac.uk/integration/techniques/definite/numerical-methods/trapezoidal-rule

- A applet to carry out integration

http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Numerical/Integration

- Code (from Pacheco hard copy text)

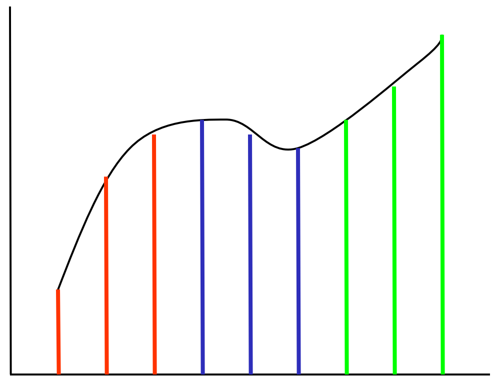
```
    PUB = /export/home/cs260x-public
Serial Code
    PUB/Pacheco/ppmpi_c/chap04/serial.c
Parallel Code
        PUB/Pacheco/ppmpi_c/chap04/trap.c
```

Serial code (Following Pacheco)

```
main() {
  float f(float x)  { return  x*x;  }           // Function we're integrating

  float h = (b-a)/n;                            //   h = trapezoid base width
                                                //   a and b: endpoints
                                                //   n = # of trapezoids

  float integral = (f(a) + f(b))/2.0;

  float  x;   int   i;

  for (i = 1, x=a; i <= n-1; i++) {
     x += h;
     integral = integral + f(x);
  }
  integral = integral*h;
 }
```

The parallel algorithm

- Decompose the integration interval into sub-intervals, one per processor
- Each processor computes the integral on its local subdomain
- Processors combine their local integrals into a global one



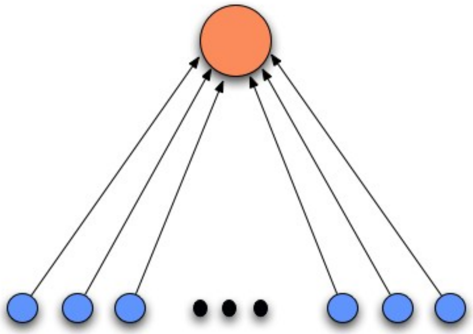
First version of the parallel code

```
 local_n = n/p;                    // Number of trapezoids; assume p divides n evenly
 float local_a = a + my_rank*local_n*h,
       local_b = local_a + local_n*h,
       integral = Trap(local_a, local_b, local_n, h);

if (my_rank == 0) {                // Sum the integrals calculated by all the processes
    total = integral;
    for (source = 1; source < p; source++) {
      MPI_Recv(&integral, 1, MPI_FLOAT, source, tag, WORLD, &status);
      total += integral;
    }
 } else
    MPI_Send(&integral, 1, MPI_FLOAT, dest, tag, WORLD);
```

Improvements

- The result does not depend on the order in which the sums are taken
- We use a linear time algorithm to accumulate contributions, but there are other orderings

```
for (source = 1; source <
{
    MPI_Recv(&integral, 1,
                    MPI_ANY_SOURCE, tag,
              WORLD, &status);
```



```
         total += integral;
```

Fortran - trap.f (1/2)

```
      program trap
      include "mpif.h"
      double precision PI25DT
      parameter (PI25DT = 3.141592653589793238462643d0)
      double precision mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, ierr, sizetype,
     1 sumtype
      f(a)=4.d0/(1.d0+a*a)
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
      sizetype = 1
      sumtype = 2
 10   if ( myid .eq. 0 ) then
         print *, 'Enter the number of intervals: (0 quits)'
         read(*,*) n
      endif
c
c    broadcast n
c

      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
c
c    check for quit signal
c
```

Fortran - trap.f (2/2)

```
      if ( n .le. 0 ) goto 30
c
c     calculate the interval size
         h = 1.0d0/n
         sum = 0.0d0
         do 20 i = myid+1, n, numprocs
              x = h*(dble(i) - 0.5d0)
              sum = sum + f(x)
 20   enddo
         mypi = h*sum
c
         print *, 'myid ', myid, 'mypi ',mypi
c     collect all the partial sums
c
         call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,
     1 MPI_SUM,0,MPI_COMM_WORLD,ierr)
c
c     node 0 prints the answer.
         if ( myid .eq. 0 ) then
              print *, 'pi is ', pi, 'Error is', abs(pi - PI25DT)
         endif
         goto 10
 30   call MPI_FINALIZE(ierr)
         stop
         end
```

Comment on MPI Performance

- **Keep track of communication**
  - Communication performance as a function of P
  - Bulk numbers are usually sufficient
  - MPI_Wait and collective inbalance usually indicates load inbalance and/or serialization
  - HPC Toolkit, mpiP

MPI Program to Approximate PI via Integration

- Apply the Trapezoidal Rule to approximate PI by integrating f(x)= 4/(1+x*x).
- We use collective routines to accomplish this.
- Provide some number of intervals.
- Each processor will contributed a portion to the sum.

Example : MPI Profile (1/2)

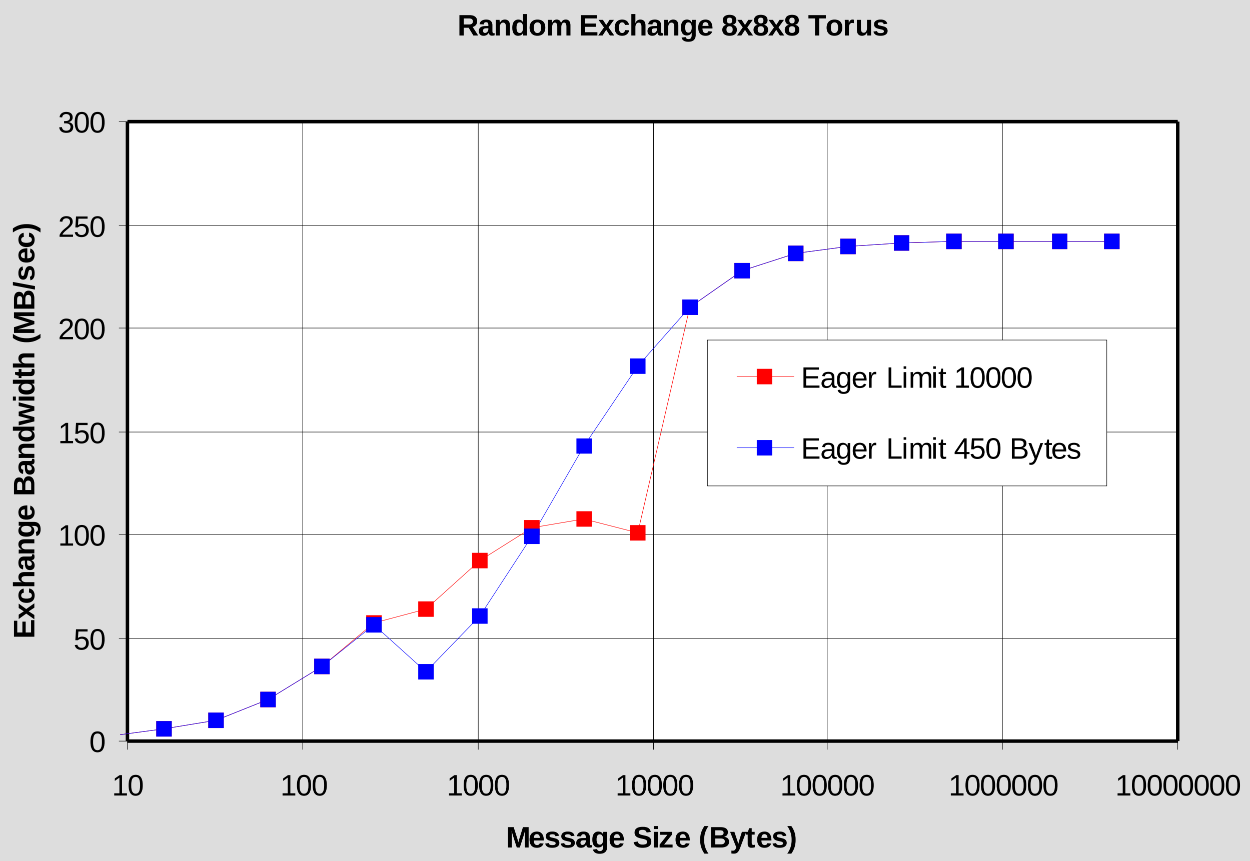
```
elapsed time from clock-cycles using freq = 700.0 MHz
------------------------------------------------------------
MPI Routine               #calls     avg. bytes      time(sec)
------------------------------------------------------------
MPI_Comm_size                  6            0.0          0.000
MPI_Comm_rank                  1            0.0          0.000
MPI_Send                  285196         6694.6         35.545
MPI_Recv                  210284          698.5         20.959
MPI_Probe                  81243            0.0        124.980
MPI_Iprobe                352732            0.0          0.358
MPI_Bcast                      5            4.0          0.000
MPI_Barrier                10000            0.0         85.153
MPI_Gather                 10002            8.0          0.803
MPI_Allgather                  3           14.7          0.001
MPI_Allreduce                  6           17.3          0.660
------------------------------------------------------------
MPI task 0 of 512 had the minimum communication time.
total communication time = 268.460 seconds.
total elapsed time       = 434.298 seconds.
top of the heap address  = 47.293 MBytes.
------------------------------------------------------------
```

Example : MPI Profile (cont 2/2)

```
Message size distributions:
MPI_Send                  #calls    avg. bytes      time(sec)
                           73141          16.0          0.273
                             380          45.9          0.004
                             786          98.9          0.002
                            2325         196.8          0.016
                            5768         379.1          0.082
                          121925         998.1         22.893
                            1023        1600.2          0.011
                             643        4079.7          0.112
                             511        5721.0          0.019
                           77672       10240.0          5.937
                            1022      960000.0          6.198
```

-env BGLMPI_EAGER=900 (BG/L default = 1000) to get adaptive routes for messages of ~1K, and better performance.

Network Exchange



For exchange between random sites on the torus, adaptive routing helps. (December 2005) default was 1000 byte eager limit, with static routes for eager messages.

If all communication is collinear on the torus, adaptive routes can't help, because there is only one minimal route on the linear path, and it may be better to increase the eager limit: BGLMPI_EAGER=10000, for example.

mpiP output

```
> .../mpiP-3.1.1/bin/mpip-insert-src ./hej-traced Unknown.64.0.1.mpiP
@ mpiP
@ Command :
@ Version                  : 3.1.1
@ MPIP Build date          : Aug 21 2007, 15:21:35
@ Start time               : 2007 08 26 15:18:41
@ Stop time                : 2007 08 26 15:18:41
@ Timer Used               : rts_get_timebase
@ MPIP env var             : '-c'
@ Collector Rank           : 0
@ Collector PID            : 0
@ Final Output Dir         : .
@ Report generation        : Single collector task
@ MPI Task Assignment      : 0 Processor <0,0,0,0> in a <4, 4, 2, 2> mesh
@ MPI Task Assignment      : 1 Processor <0,0,0,1> in a <4, 4, 2, 2> mesh
....
```

mpiP - Overall statistics

```
@--- MPI Time (seconds) -----------------------------------
Task    AppTime    MPITime     MPI%
   0    0.00351   0.000305     8.70
   1   3.01e-05   7.93e-06    26.32
   2   5.08e-05   2.76e-05    54.21
...
  62   5.42e-05   3.12e-05    57.52
  63   3.04e-05   7.84e-06    25.78
   *    0.00859    0.00396    46.07

@--- Callsites: 1 -----------------------------------------
 ID Lev File/Address        Line Parent_Funct            MPI_Call
  1   0 hej.f90            18 hej                     Gather
```

mpiP - Top twenty sites...

```
@--- Aggregate Time (top twenty, descending, milliseconds) -----------
Call                     Site       Time    App%    MPI%     COV
Gather                      1       3.96   46.07  100.00    2.36

@--- Aggregate Sent Message Size (top twenty, descending, bytes) -----
Call                     Site      Count      Total       Avrg  Sent%
Gather                      1         64   8.13e+03        127 100.00
```

mpiP - all sites

```
@--- Callsite Time statistics (all, milliseconds): 64 ----------------
Name          Site Rank  Count      Max     Mean      Min   App%   MPI%
Gather           1    0      1    0.305    0.305    0.305   8.70 100.00
...
Gather           1    *     64    0.647   0.0618  0.00743  46.07 100.00
@--- Callsite Message Sent statistics (all, sent bytes) --------------
Name          Site Rank   Count       Max       Mean        Min       Sum
Gather           1    0       1       127        127        127       127
...
Gather           1    *      64       127        127        127      8128
@--- End of Report ---------------------------------------------------
```

MPI on Blue Gene

- **MPI implementation based on MPICH-2 (Argonne)**
- **Include path for <mpi.h>, mpif.h :**
  - -I/bgl/BlueLight/ppcfloor/bglsys/include
- **Libraries to link for MPI:**
  - -L/bgl/BlueLight/ppcfloor/bglsys/lib
  - -lmpich.rts -lmsglayer.rts -lrts.rts -ldevices.rts
- **Sample Makefile:**

```
FC = blrts_xlf
FFLAGS = -g -O -qarch=440 -qmaxmem=64000
MPI_INC = -I/bgl/BlueLight/ppcfloor/bglsys/include
MPI_LIB = -L/bgl/BlueLight/ppcfloor/bglsys/lib \
            -lmpich.rts -lmsglayer.rts -lrts.rts -ldevices.rts
LD = blrts_xlf
LDFLAGS = -g
hello.x : hello.o
   $(LD) $(LDFLAGS) hello.o $(MPI_LIB) -o hello.x
hello.o : hello.f
   $(FC) -c $(FFLAGS) $(MPI_INC) hello.f
```

Submitting jobs with mpirun

- **You can use "mpirun" to submit jobs. The Blue Gene mpirun is in**
  - /bgl/BlueLight/ppcfloor/bglsys/bin
- **Typical use:**
  - mpirun -np 512 -cwd `pwd` -exe your.x
- **common options:**

  -args "list of arguments"
  -env "VARIABLE=value"
  -mode CO/VN (coprocessor/virtual-node)
- **coprocessor mode : one MPI process per node, 512 MB or 1 GB limit per process.**
- **virtual-node mode : two MPI processes per node, 256 MB or 512 MB limit per process; L3 cache, memory, networks, are shared.**
- **More details: mpirun -h (for help)**

- **redbook: Blue Gene/L System Administration   (www.redbooks.ibm.com, sg247178)**
- **Limitations: one job per partition, limited partition sizes**

Remarks

- **Read more about it – IBM REDBOOKS:**
  - **http://www.ibm.com/redbooks**
    - Application Development Guide
    - System Administration Guide
    - Performance Tools

- **This exercise presents a simple program to determine the value of pi. The algorithm suggested here is chosen for its simplicity. The method evaluates the integral of 4/(1+x*x) between 0 and 1. The method is simple: the integral is approximated by a sum of n intervals; the approximation to the integral in each interval is (1/n)*4/(1+x*x). The master process (rank 0) asks the user for the number of intervals; the master should then broadcast this number to all of the other processes. Each process then adds up every n'th interval (x = rank/n, rank/n+size/n,...). Finally, the sums computed by each process are added together using a reduction. You may want to use these MPI routines in your solution:**
  MPI_Bcast MPI_Reduce