The Message Passing Interface (MPI) and its integration with the EGEE Grid

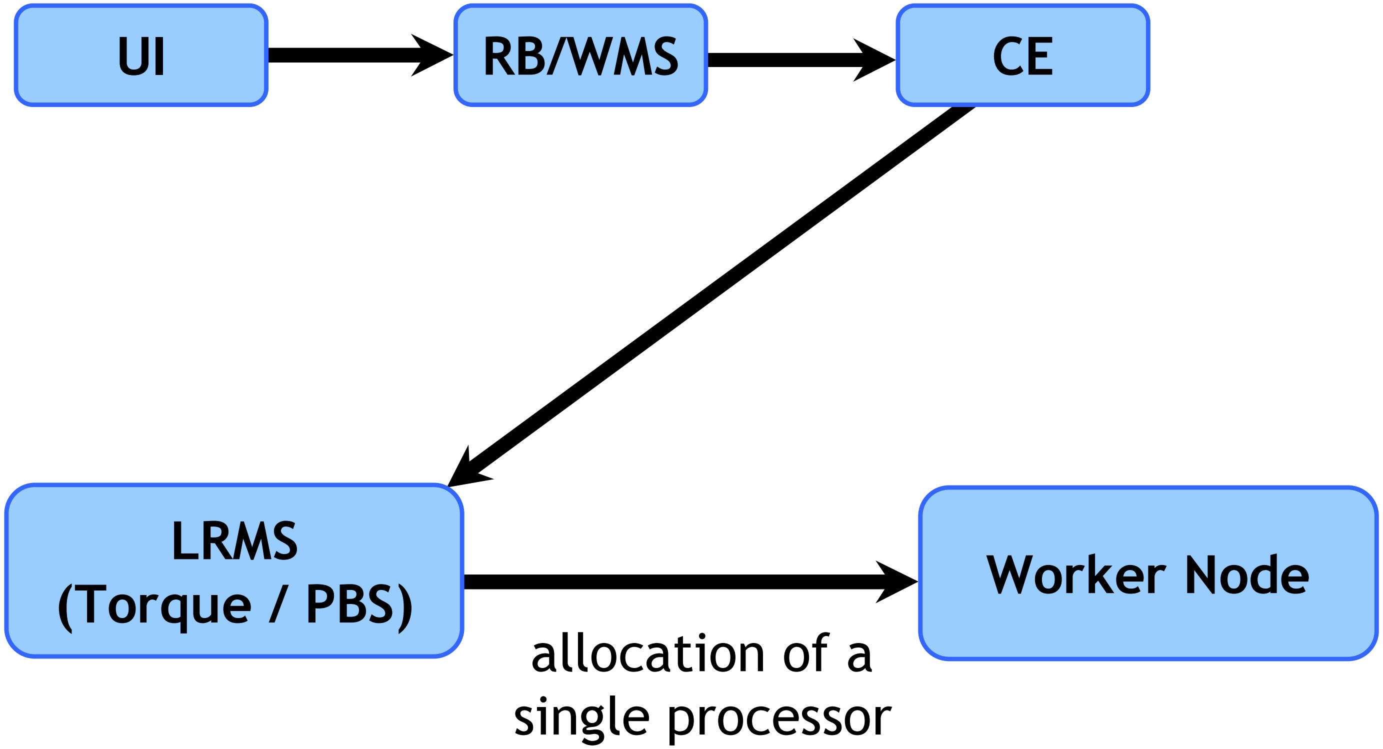
Vangelis Koukis
HG-01-GRNET and HG-06-EKT admin team
vkoukis@cslab.ece.ntua.gr

Issues on Grid Technologies, TEI of Western Macedonia, Kastoria, 2009/03/12

Presentation Outline

- Parallel Programming
    - Parallel architectures
    - Parallel programming models and MPI
- Introduction to basic MPI services
- MPI demonstration on a dedicated cluster
- Integration of MPI jobs on the EGEE Grid
- MPI job submission to HG-01-GRNET
- Discussion / Q&A Session

The lifetime of a serial job on the Grid



UI

RB/WMS

CE

LRMS
(Torque / PBS)

Worker Node

allocation of a
single processor

The need for MPI apps on the Grid

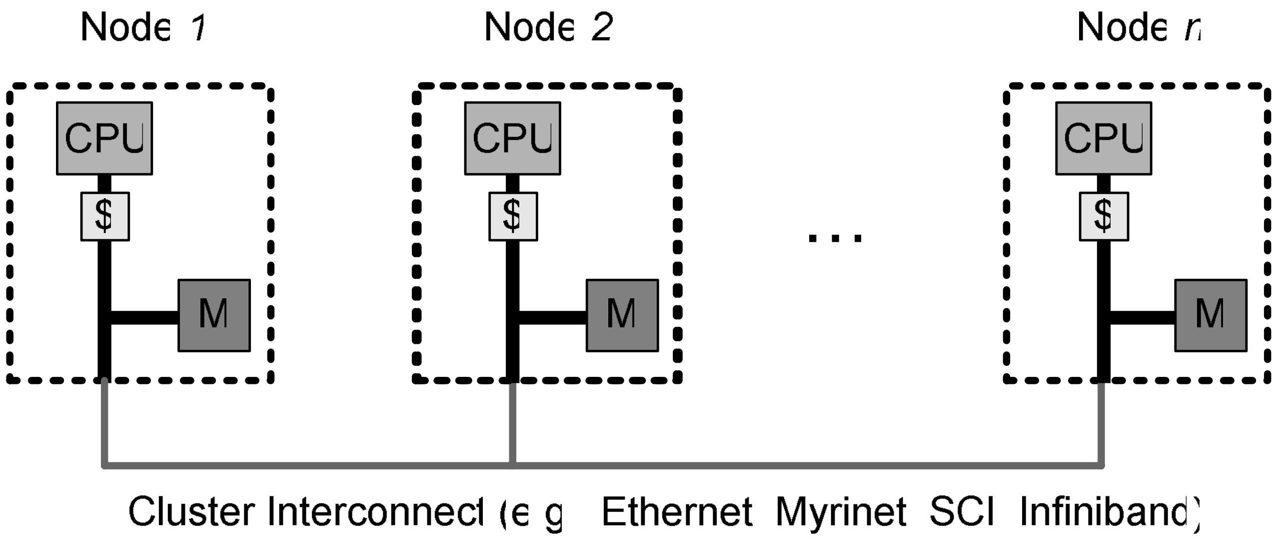
- The Grid offers very large processing capacity: How can we best exploit it?
  - Thousands of processing elements / cores
- The easy way: The EP way
  - Submit a large number of independent (serial) jobs, to process distinct parts of the input workload concurrently
- What about dependencies?
  - What if the problem to be solved is not “Embarassingly Parallel”?

Presentation Outline

- Parallel Programming
  - Parallel architectures
  - Parallel programming models and MPI
- Introduction to basic MPI services
- MPI demonstration on a dedicated cluster
- Integration of MPI jobs on the EGEE Grid
- MPI job submission to HG-01-GRNET
- Discussion / Q&A Session

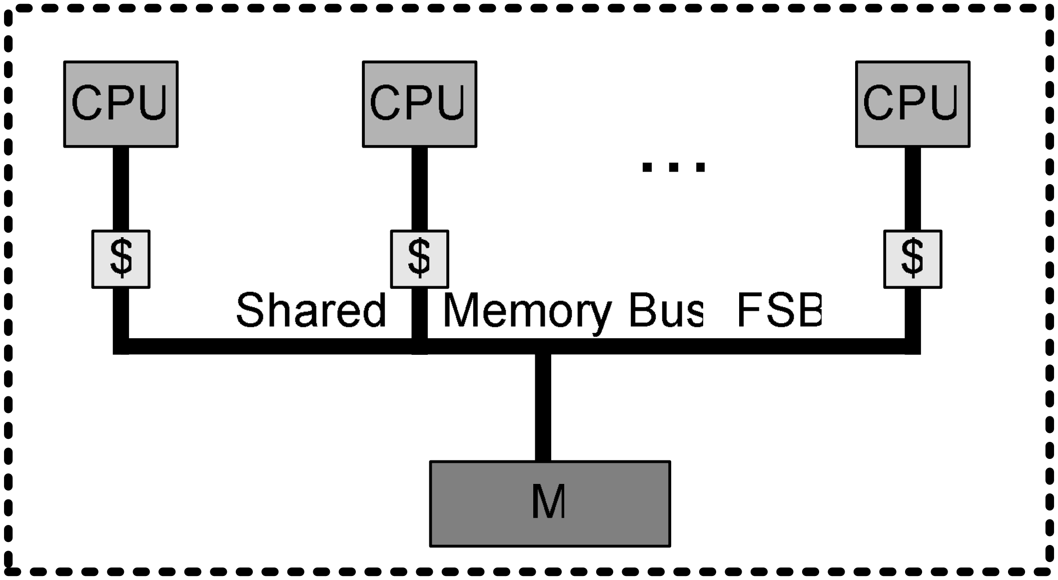
Parallel Architectures (1)

- Distributed Memory Systems
  (e.g., Clusters of Uniprocessor Systems)



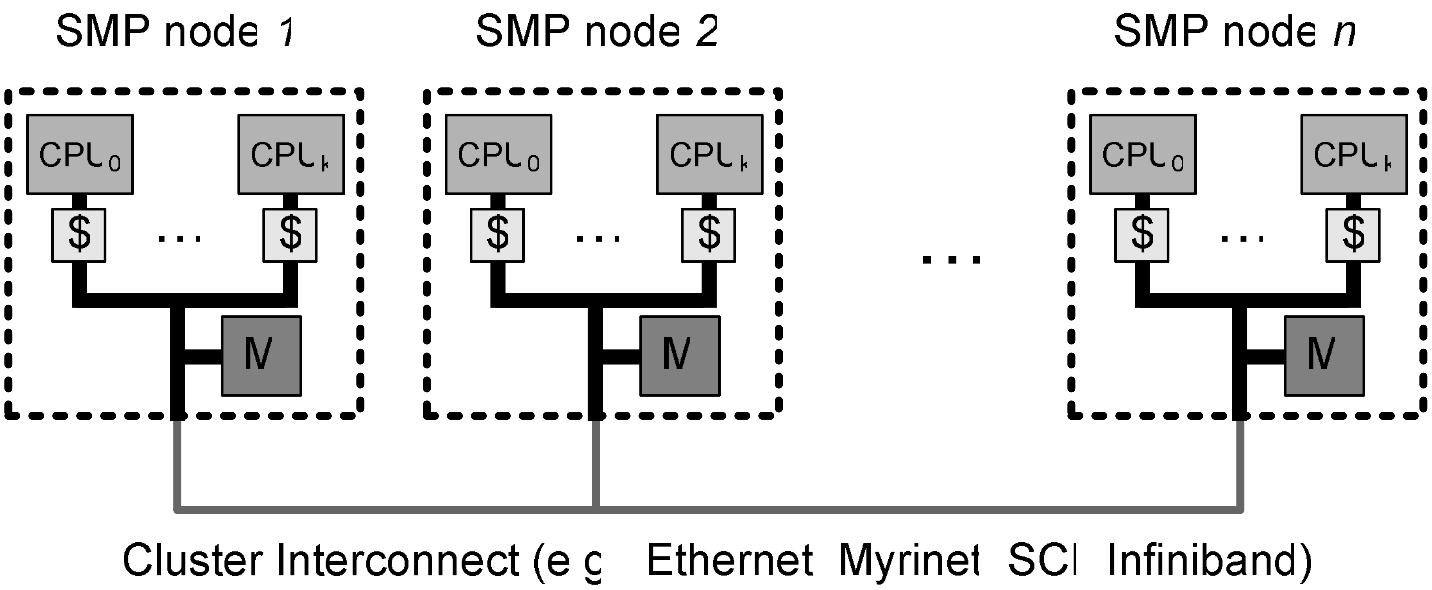
Parallel Architectures (2)

- Shared Memory Architectures
  (e.g., Symmetric Multiprocessors)

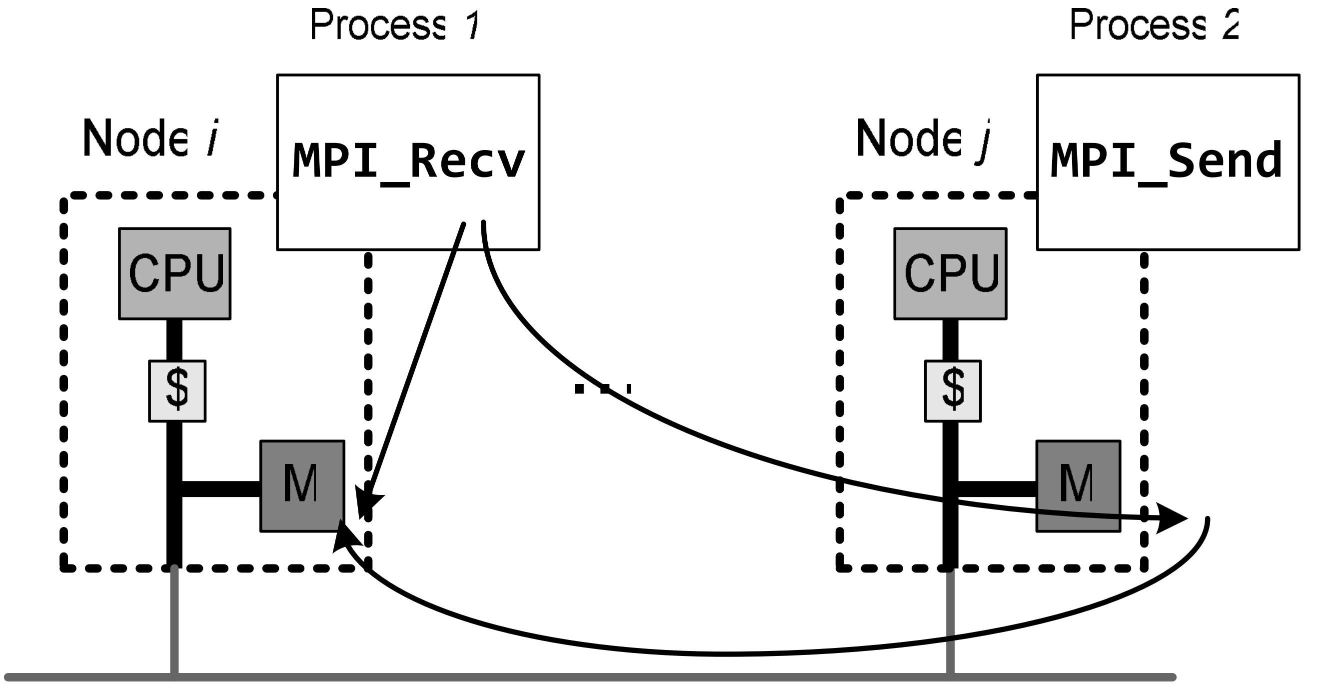


Parallel Architectures (3)

- Hybrid – Multilevel Hierarchies (e.g., Clusters of SMPs, Multicore/SMT Systems)



One model: The Message-Passing Paradigm



Presentation Outline

- Parallel Programming
  - Parallel architectures
  - Parallel programming models and MPI
- Introduction to basic MPI services
- MPI demonstration on a dedicated cluster
- Integration of MPI jobs on the EGEE Grid
- MPI job submission to HG-01-GRNET
- Discussion / Q&A Session

What Is MPI?

- A *standard*, not an *implementation*
- An app library for message-passing
- Following a *layered* approach
- Offering standard language bindings at the *highest level*
- Managing the interconnect at the *lowest level*
- Offers C, C++, Fortran 77 and F90 bindings

Lots of MPI implementations

- MPICH
  http://www-unix.mcs.anl.gov/mpi/mpich
- MPICH2
  http://www-unix.mcs.anl.gov/mpi/mpich2
- MPICH-GM
  http://www.myri.com/scs
- LAM/MPI
  http://www.lam-mpi.org
- LA-MPI
  http://public.lanl.gov/lampi
- Open MPI
  http://www.open-mpi.org
- SCI-MPICH
  http://www.lfbs.rwth-aachen.de/users/joachim/SCI-MPICH
- MPI/Pro
  http://www.mpi-softtech.com
- MPICH-G2
  http://www3.niu.edu/mpi

Single Program, Multiple Data (SPMD)

- Multiple peer *processes* executing the same *program image*
- A number, called rank is used to tell each of the processes apart
  - Each process undertakes a specific subset of the input workload for processing
  - Execution flow changes based on the value of rank
- The basic rules of parallel programming
  - Effort to maximize parallelism
  - Efficient resource management (e.g., memory)
  - Minimization of communication volume
  - Minimization of communication frequency
  - Minimization of synchronization

Processes and Communicators

- Peer processes are organized in groups, called *communicators*. At program start, there is MPI_COMM_WORLD
- Each process is assigned a single rank in the range of 0...P-1, where P is the number of processes in a communicator
- We're referring to *processes*, not *processors* (what about time-sharing?)

Typical MPI code structure

```
#include <mpi.h>

int main(int argc, char *argv[])
{
     ...
     /* Initialization of MPI support */
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     ...
     /* MPI Finalization, cleanup */
     MPI_Finalize();
}
```

Basic MPI services (1)

- MPI_Init(argc,argv)
  - Library Initialization
- MPI_Comm_rank(comm,rank)
  - Returns the rank of a process in communication *comm*
- MPI_Comm_size(comm,size)
  - Returns the size (the number of processes) in *comm*
- MPI_Send(sndbuf,count,datatype,dest,tag,comm)
  - Sends a message to process with rank *dest*
- MPI_Recv(rcvbuf,count,datatype,source,tag, comm,status)
  - Receives a message from process with rank *source*
- MPI_Finalize()
  - Library Finalization

Basic MPI Services (2)

```
int MPI_Init(int* argc, char*** argv)
```

- Initializes the MPI environment
- Usage example:

```
int main(int argc,char *argv[])
{
        …
        MPI_Init(&argc,&argv);
        …
}
```

Basic MPI Services (3)

```
int MPI_Comm_rank (MPI_Comm comm, int* rank)
```

- Returns the *rank* of the calling process in communicator *comm*
- Usage example:

```
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Basic MPI Services (4)

```
int MPI_Comm_size (MPI_Comm comm, int* size)
```

- Returns the *size* (number of processes) in communicator *comm*
- Usage example:

```
int size;

MPI_Comm_size(MPI_COMM_WORLD,&size);
```

Basic MPI Services (5)

```
int MPI_Send(void *buf, int count, int dest,
int tag, MPI_Datatype datatype, MPI_Comm
comm)
```

- The calling process sends a message from *buf* to the process with rank *dest*
- Array *buf* should contain *count* elements of type *datatype*
- Usage example:

```
int message[20],dest=1,tag=55;

MPI_Send(message, 20, dest, tag, MPI_INT,
     MPI_COMM_WORLD);
```

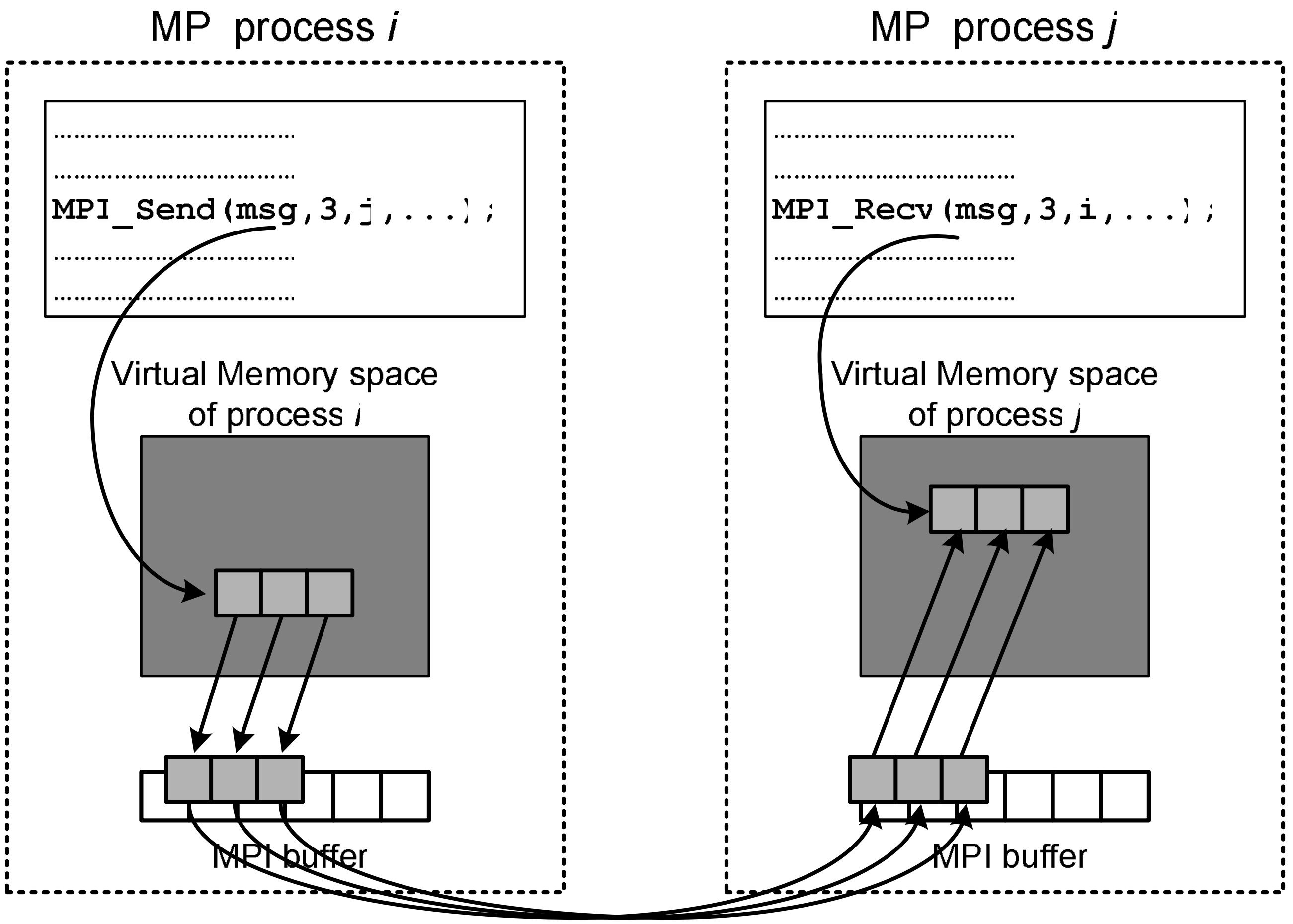
Basic MPI Services (6)

```
int MPI_Recv(void *buf, int count, int
source, int tag, MPI_Datatype datatype,
MPI_Comm comm, MPI_Status *status)
```

- Receives a message from process with rank *source* and saves it in *buf*
- At most *count* elements of type *datatype* are to be received (MPI_Get_count used to get the precise count)
- Wildcards
  - MPI_ANY_SOURCE, MPI_ANY_TAG
- Usage example:

```
int message[50],source=0,tag=55;
MPI_Status status;
MPI_Recv(message, 50, source, tag,
     MPI_INT, MPI_COMM_WORLD, &status);
```

Basic MPI Services (7)



MP process *i*

```
……………………………
……………………………
MPI_Send(msg,3,j,...);
……………………………
……………………………
```

Virtual Memory space of process *i*

MPI buffer

MP process *j*

```
……………………………
……………………………
MPI_Recv(msg,3,i,...);
……………………………
……………………………
```

Virtual Memory space of process *j*

MPI buffer

Basic MPI Services (8)

```
int MPI_Finalize()
```

- Finalizes MPI support
- Should be the final MPI call made by the program

A simple example

```
/* Computes f(0)+f(1) in parallel */
#include <mpi.h>

int main(int argc,char** argv){
   int v0,v1,sum,rank;
   MPI_Status stat;
   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   if(rank==1) {
```

Process *1*

```
      v1=f(1);
      MPI_Send(&v1,1,0,50,MPI_INT,MPI_COMM_WORLD);
   else if(rank==0){
      v0=f(0);
      MPI_Recv(&v1,1,1,50,MPI_INT,MPI_COMM_WORLD,&stat);
      sum=v0+v1;
   }
```

Process *0*

```
   MPI_Finalize();
}
```

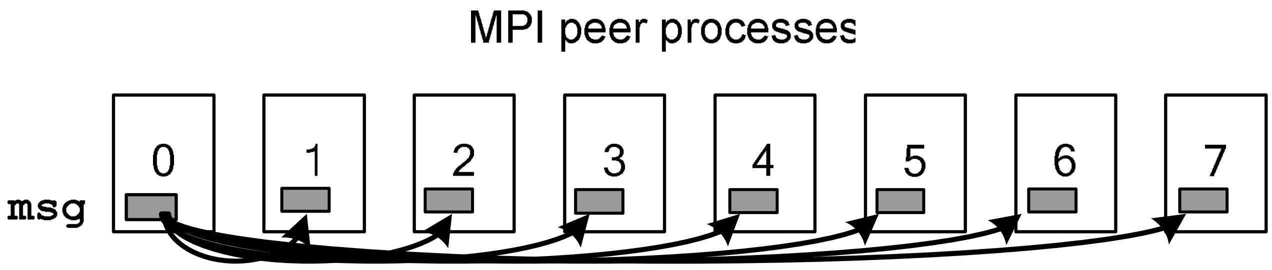
Different Communication Semantics

- Point-to-point / Collective Communication
- Synchronous, buffered or ready
  - With different buffering and synchronization semantics
- Blocking or non-blocking calls
  - Depending on when MPI returns control to the calling process

Collective Communication (1)

Example: Process *0* needs to send *msg* to processes *1-7*

```
if (rank == 0)
  for (dest = 1; dest < size; dest++)
    MPI_Send(msg,count,dest,tag,MPI_FLOAT,MPI_COMM_WORLD);
```

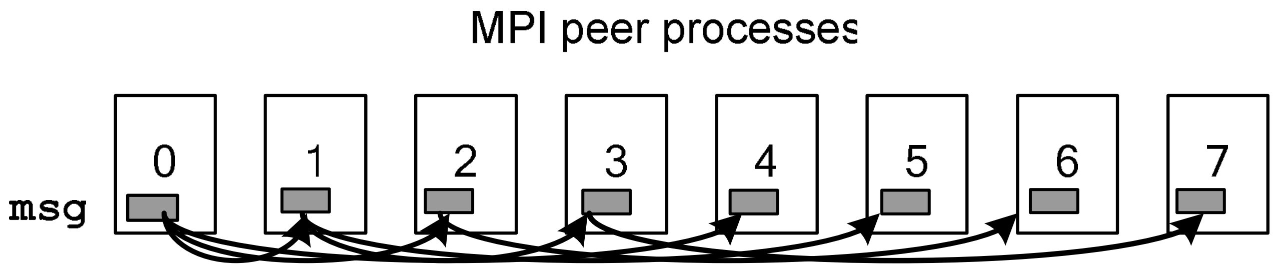


In general: *p - 1* communication steps needed for *p* processes

Collective Communication(2)

Example: Process 0 needs to send msg to processes 1-7

```
MPI_Bcast(msg,count,MPI_FLOAT,0,MPI_COMM_WORLD);
```



In general: $\lceil \log_2 p \rceil$ communication steps needed for *p* processes

Collective Communication (3)

```
int MPI_Bcast(void* message, int count,
MPI_Datatype datatype, int root, MPI_Comm
comm)
```

- Message in *message* is broadcast from process *root* to all processes in communicator *comm*
- Memory at *message* should contain *count* elements of type *datatype*
- Called by all processes in *comm*

Collective Communication (4)

```
int MPI_Reduce(void* operand, void*
result, int count, MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm comm)
```

- All data in *operand* pointers contributed to reduction operation *op*, and the result is retrieved by *root* in *result*
- Needs to be called by all processes in *comm*
- MPI_Op: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, etc.
- An MPI_Allreduce variant is also available

Collective Communication (5)

```
/* Compute f(0)+f(1) + … + f(n) in parallel */
#include <mpi.h>

int main(int argc,char *argv[]){
   int sum,rank;
   MPI_Status stat;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   /* Assumes values have been computed in f[] */
   MPI_Reduce(&f[rank],&sum,1,MPI_INT,MPI_SUM,0,
            MPI_COMM_WORLD);
   MPI_Finalize();
}
```

Collective Communication (6)

```
int MPI_Barrier(MPI_Comm comm)
```

- Synchronizes execution of processes in communicator *comm*
- Each process blocks until *all* participating processes reach the barrier
- Reduces the degree of attainable parallelism

Collective Communication (7)

```
int MPI_Gather(void* sendbuf, int sendcnt,
MPI_Datatype sendtype, void* recvbuf, int
recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
```

- Data in *sendbuf* are gathered in memory belonging to process with rank *root* (in increasing rank)
- Results stored in *recvbuf*, which contains meaningful data only for *root*
- Also available as an MPI_Allgather variant
- The reverse project: MPI_Scatter

Synchronous – Buffered - Ready

- Different completion semantics for send and receive operations
- Available in blocking as well as non-blocking variants
- A simple MPI_Send can be synchronous or buffered, depending on implementation

Synchronous – Buffered – Ready (2)

- `int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Returns successfully only when operation has completed on the receiver side - safe
- `int MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Returns as soon as possible, performs intermediate buffering and schedules sending over the network – may fail later on
- `int MPI_Rsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Returns as soon as possible, but requires guarantee that a receive operation has already been posted on the remote side - uncertain

Synchronous – Buffered – Ready (3)

MPI_Bsend	MPI_Ssend	MPI_Rsend
Completes locally	Syncs with remote	Completes locally
2 memory copies	1 memory copy	1 memory copy
May fail later due to resource constraints	Returns only if send successful	Returns only if send successful
No need for outstanding receive	No need for outstanding receive	Will fail if no receive is outstanding on the remote

Non – Blocking Communication

- MPI returns control immediately to the calling process, *but*
- It is not safe to reuse provided buffers before the posted operations have completed
- Two ways to check for operation completion:
  - `int MPI_Test (MPI_Request* request,int* flag, MPI_Status* status)`
  - `int MPI_Wait (MPI_Request* request,MPI_Status* status)`

Non – Blocking Communication (2)

- Each blocking function has a non-blocking counterpart:
  - MPI_Isend (corresponds to MPI_Send)
  - MPI_Issend (corresponds to MPI_Ssend)
  - MPI_Ibsend (corresponds MPI_Bsend)
  - MPI_Irsend (corresponds MPI_Rsend)
  - MPI_Irecv (corresponds MPI_Recv)

Non – Blocking Communication (3)

- Why use non-blocking operations?
  - Enables overlapping computation with communication for efficiency:

Blocking	**Non-blocking**
`MPI_Recv();`	`MPI_Irecv();`
`MPI_Send();`	`MPI_Isend();`
`Compute();`	`Compute();`
	`Waitall();`

MPI Datatypes

MPI_CHAR: 8-bit character
MPI_DOUBLE: 64-bit floating point value
MPI_FLOAT: 32-bit floating point value
MPI_INT: 32-bit integer
MPI_LONG: 32-bit integer
MPI_LONG_DOUBLE: 64-bit floating point value
MPI_LONG_LONG: 64-bit integer
MPI_LONG_LONG_INT: 64-bit integer
MPI_SHORT: 16-bit integer
MPI_SIGNED_CHAR: 8-bit signed character
MPI_UNSIGNED: 32-bit unsigned character
MPI_UNSIGNED_CHAR: 8-bit unsigned character
MPI_UNSIGNED_LONG: 32-bit unsigned integer
MPI_UNSIGNED_LONG_LONG: 64-bit unsigned integer
MPI_UNSIGNED_SHORT: 16-bit unsigned integer
MPI_WCHAR: 16-bit unsigned integer

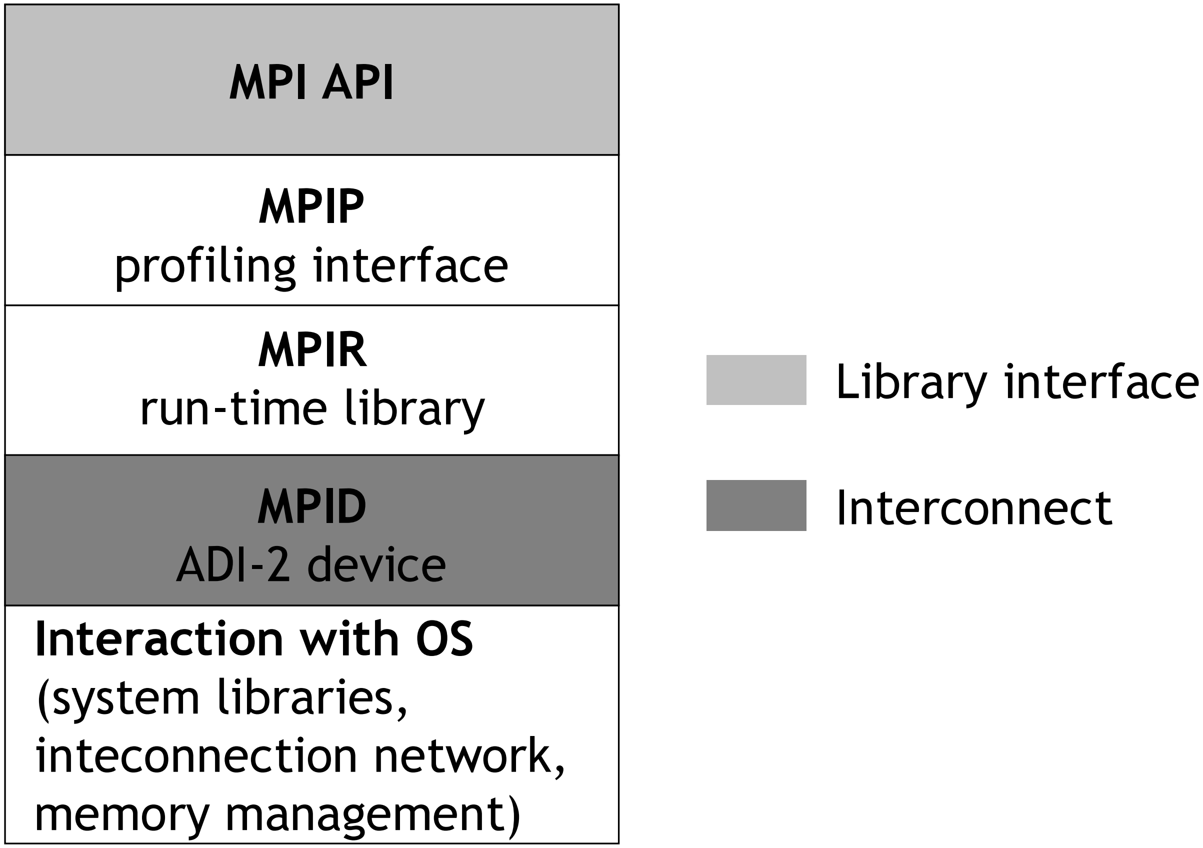
MPI Datatypes (2)

- MPI data packing for communication needed for complex datatypes
- *count* parameter (for homogeneous data in consecutive memory locations)
- MPI_Type_struct (derived datatype)
- MPI_Pack(), MPI_Unpack() (for heterogeneous data)

The MPI-2 Standard

- Support for Parallel I/O
- Dynamic process management, runtime process spawning and destruction
- Support for remote memory access operations
  - One-sided RDMA operations

The MPICH implementation



MPI API

MPIP
profiling interface

MPIR
run-time library

MPID
ADI-2 device

Interaction with OS
(system libraries,
inteconnection network,
memory management)

▒ Library interface

▓ Interconnect

The MPICH Implementation (2)

- 1 send message queue, 2 receive queues per process
  - posted + unexpected
- Underlying device selection based on the destination rank
  - p4, shmem
- Protocol selection based on message size
  - Short < 1024 bytes, rendezvous > 128000 bytes, eager protocol for sizes in-between
- Flow control
  - 1MB buffer space for the eager protocol per pair of processes

Presentation Outline

- Parallel Programming
  - Parallel architectures
  - Parallel programming models and MPI
- Introduction to basic MPI services
- MPI demonstration on a dedicated cluster
- Integration of MPI jobs on the EGEE Grid
- MPI job submission to HG-01-GRNET
- Discussion / Q&A Session

MPI program execution (1)

- The traditional, HPC way: running directly on a dedicated PC Cluster
- Linux cluster of 16 multicore nodes (clone1…clone16)
- Program compilation and execution
  - Appropriate PATH for a specific MPI implementation
    - `export PATH=/usr/local/bin/mpich-intel:…:$PATH`
  - Compile and link with the relevant MPI-specific libraries
    - `mpicc test.c -o test -O3`
  - Program execution
    - `mpirun -np 16 test`

Demo time!

- Run a simple “Hello World” 16-process MPICH job on dedicated cluster (clones)

MPI program execution (2)

- Which machines do the peer processes run on?
  - Machine file

```
$ cat <<EOF >machines
clone4
clone7
clone8
clone10
EOF

$ mpiCC test.cc -o test -O3 -static -Wall
$ mpirun -np 4 -machinefile machines test
```

MPI program execution (3)

- Implementation details
  - ➡How are the needed processes created? An implementation- and OS-specific issue
    - passwordless rsh / ssh, cluster nodes trust one another and share a common userbase
    - Using daemons, (“lamboot” for LAM/MPI)
- What about file I/O;
  - ➡Shared storage among all cluster nodes
    - NFS in the most common [and slowest] case
    - Deployment of a parallel fs, e.g., PVFS, GFS, GPFS

Presentation Outline

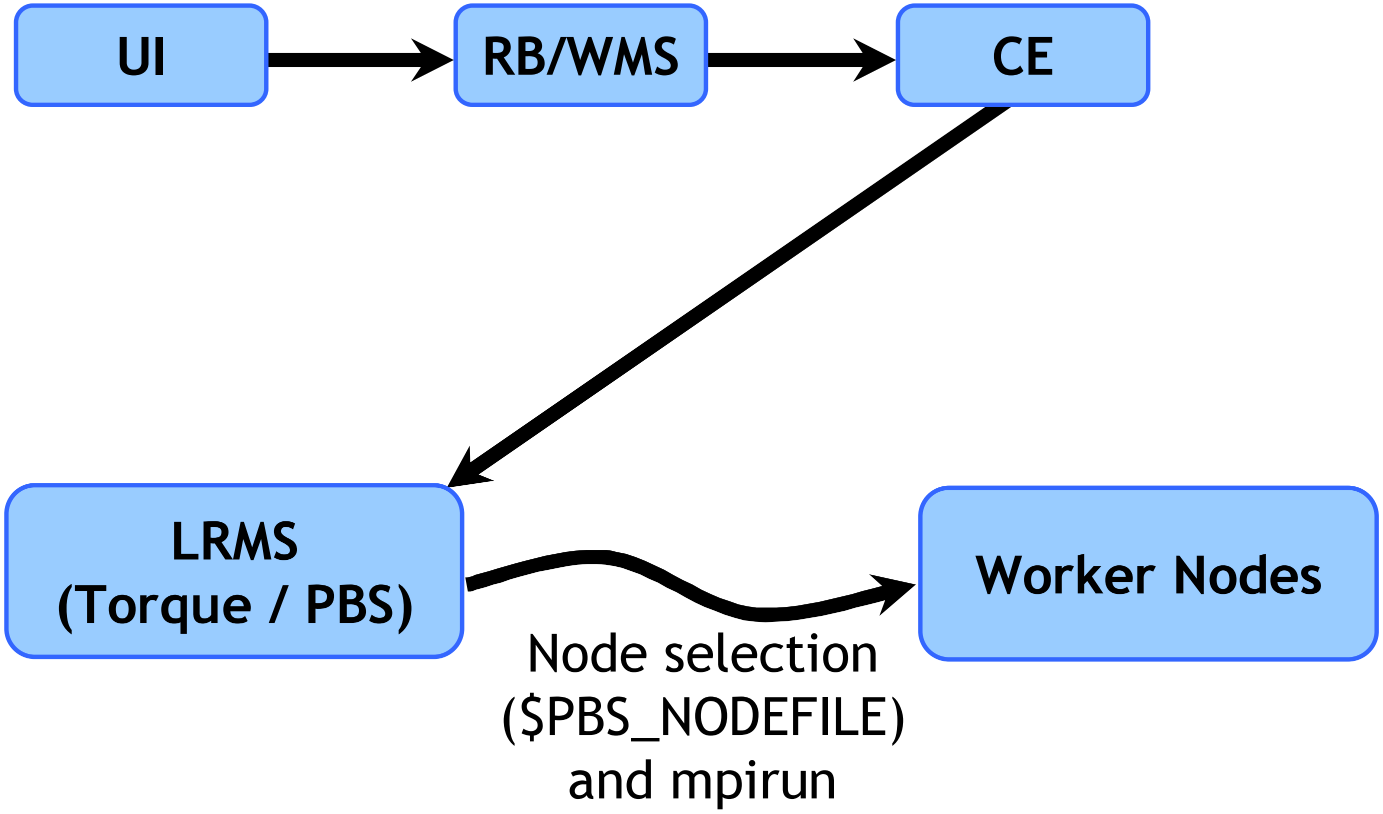
- Parallel Programming
  - Parallel architectures
  - Parallel programming models and MPI
- Introduction to basic MPI services
- MPI demonstration on a dedicated cluster
- Integration of MPI jobs on the EGEE Grid
- MPI job submission to HG-01-GRNET
- Discussion / Q&A Session

MPI jobs in the Grid environment

- Submission of MPICH-type parallel jobs

```
        Type = "job";
        JobType = "MPICH";
        NodeNumber = 64;
        Executable = “mpihello";
        StdOutput = "hello.out";
        StdError = "hello.err";
        InputSandbox = {“mpihello"};
        OutputSandbox = {"hello.out","hello.err"};
        #RetryCount = 7;
        #Requirements = other.GlueCEUniqueID ==
"ce01.isabella.grnet.gr:2119/jobmanager-pbs-short"
```

The lifetime of an MPI job on the Grid



UI

RB/WMS

CE

LRMS
(Torque / PBS)

Worker Nodes

Node selection
($PBS_NODEFILE)
and mpirun

Presentation Outline

- Parallel Programming
  - Parallel architectures
  - Parallel programming models and MPI
- Introduction to basic MPI services
- MPI demonstration on a dedicated cluster
- Integration of MPI jobs on the EGEE Grid
- MPI job submission to HG-01-GRNET
- Discussion / Q&A Session

Demo time!

- Submission of a “Hello World” 4-process MPICH job to HG-01-GRNET

Questions – Issues - Details

- Who is responsible for calling mpirun;
  - On which nodes? How are they selected?
- Shared homes / common storage?
- Process spawing and destruction? Accounting?
  - MPICH-specific solutions, based on rsh / ssh
  - mpiexec to integrate process creation with Torque
  - CPU Accounting for multiple processes per job
- Support for different Interconnects and/or MPI implementations?
  - Where does compilation of the executable take place?

Now and in the future...

- Grid support for MPI jobs is a Work In Progress
  - Support for MPICH over TCP/IP (P4 device)
  - Possible problems with other devices, since P4-specific hacks are used
- Need for pre/post-processing scripts
  - Compilation of the executable on the remote Worker Nodes?

EGEE MPI Working Group

- Aims to provide standardized, generic support for different MPI implementations
  - ➡http://egee-docs.web.cern.ch/egee-docs/uig/development/uc-mpi-jobs_2.html
- Proposes implementation guidelines for the compilation and execution of parallel jobs

Other Issues

- Processor selection and allocation to processes, packing of processes to nodes
  - What about message latency?
  - Per-node memory bandwidth
  - Available memory per node
- Support for hybrid architectures
  - Combine MPI with pthreads / OpenMP to better adapt to the underlying architecture

Bibliography – Online sources

- Writing Message-Passing Parallel Programs with MPI (Course Notes – Edinburgh Parallel Computing Center)
- Using MPI-2: Advanced Features of the Message-Passing Interface (Gropp, Lusk, Thakur)
- http://www.mpi-forum.org (Definition of the MPI 1.1 and 2.0 standards)
- http://www.mcs.anl.gov/mpi (home of the MPICH implementation)
- comp.parallel.mpi (newsgroup)

Presentation Outline

- Parallel Programming
  - Parallel architectures
  - Parallel programming models and MPI
- Introduction to basic MPI services
- MPI demonstration on a dedicated cluster
- Integration of MPI jobs on the EGEE Grid
- MPI job submission to HG-01-GRNET
- Discussion / Q&A Session