MPI-2

Introduction
Dynamic Process Creation

Based on notes by Sathish Vadhiyar, Rob Thacker, and David Cronk

- http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm
- Using MPI2: Advanced Features of the Message-Passing Interface. http://www-unix.mcs.anl.gov/mpi/usingmpi2/

MPI History

- Standardization started (1992)
- MPI-1 completed (1.0) (May 1994)
- Clarifications (1.1) (June 1995)
- MPI-2 (started: 1995, finished: 1997)

  >> MPI 1.2 issued in 1995 (minor corrections/clarifications)

MPI-2

- Dynamic process creation and management
- One sided communications
- Extended collective operations
- Parallel I/O
- Miscellany

Process Creation and Management

- MPI-1 application is static
- MPI-2 : process creation after MPI application has started
- Motivation:
  - Task farming applications
  - Useful in assembling complex distributed applications
  - Of questionable value with batch queuing systems and resource managers
  - Processes are started during runtime
  - serial applications executing parallel codes
  - To be friendly to the PVM users
  - Standard manner of starting processes in PVM

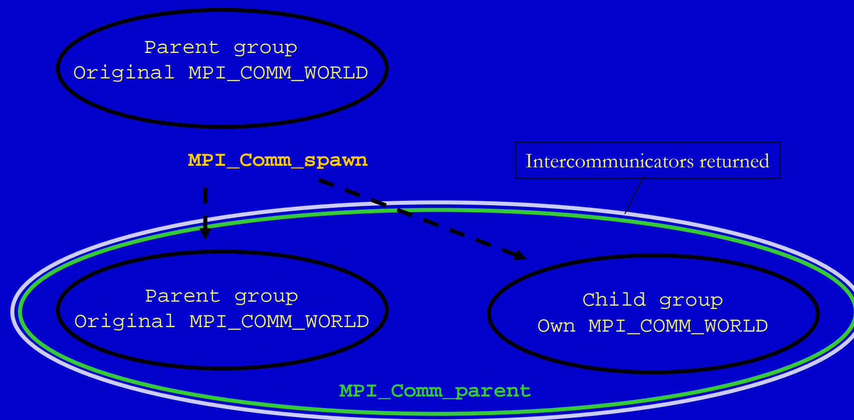
Process Creation - Features

- Creation and cooperative termination
- Communication between new processes and existing application
- Communication between 2 MPI applications

Dynamic Process Management

- Spawning new processes
  - Collective over a group
  - New processes form their own group with their own MPI_COMM_WORLD
  - Spawning processes are returned an intercommunicator with the spawning processes part of the local group and spawned processes part of the remote group
  - Spawned processes can get this intercommunicator with call to MPI_Comm_get_parent

Parent/Children communicators



Process Creation - API

MPI_Comm_ spawn(char *command, char *argv[], int maxprocs, MPI_info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])

IN command

IN argv

IN maxprocs

IN info - a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root) (MPI_INFO_NULL can be used)

IN root - rank of process in which previous arguments are examined (integer)

IN comm - intracommunicator containing group of spawning processes

OUT intercomm - intercommunicator between original group and the newly spawned group (handle)

OUT array_of_errcodes

API

MPI_Comm_get_parent(MPI_Comm *parent)
OUT parent - the parent communicator

MPI_Comm_spawn_ multiple(int count, char *array_of_commands[], char **array_of_argv[], int array_of_maxprocs[], MPI_Info array_of_info[], int root, MPI_Comm comm, MPI_Comm *intercomm, int array_of_errcodes[])
- for starting multiple binaries

MPI_UNIVERSE_SIZE

- Attribute of MPI_COMM_WORLD
  - Best guess from system of how many processes could exist
- Is not required to be defined by MPI implementation
  - Main problem – need to relate to scheduler via library of some sort
  - May be set by an environment variable on some systems
  - Best way to do things is to have value set via a call to the start-up program
    - e.g. mpiexec –n 1 –universe_size 10 my_prog

Master example

```
/* manager */
#include "mpi.h"
int main(int argc, char *argv[])
{
   int world_size, universe_size, *universe_sizep, flag;
   MPI_Comm everyone;           /* intercommunicator */
   char worker_program[100];

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);

   if (world_size != 1) error("Top heavy with management");

   MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
&universe_sizep, &flag);
   if (!flag) {
     printf("This MPI does not support UNIVERSE_SIZE. How many\n\
processes total?");
     scanf("%d", &universe_size);
   } else universe_size = *universe_sizep;
   if (universe_size == 1) error("No room to start workers");
```

Master example

```
/* * Now spawn the workers. Note that there is a run-time determination of
what type of worker to spawn, and presumably this calculation must be done
at run time and cannot be calculated before starting the program. If
everything is known when the application is first started, it is generally better
to start them all at once in a single MPI_COMM_WORLD. */

  choose_worker_program(worker_program);
  MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
MPI_ERRCODES_IGNORE);

/* * Parallel code here. The communicator "everyone" can be used to
communicate with the spawned processes, which have ranks
0,..,MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
"everyone". */

  MPI_Finalize();
  return 0; }
```

Worker code

```
/* worker */

#include "mpi.h"
int main(int argc, char *argv[])
{
  int size;
  MPI_Comm parent;
  MPI_Init(&argc, &argv);
  MPI_Comm_get_parent(&parent);
  if (parent == MPI_COMM_NULL) error("No parent!");
  MPI_Comm_remote_size(parent, &size);
  if (size != 1) error("Something's wrong with the parent");

/* * Parallel code here. The manager is represented as the process
with rank 0 in (the remote group of) MPI_COMM_PARENT. If the
workers need to communicate among themselves, they can use
MPI_COMM_WORLD. */

  MPI_Finalize();
  return 0; }
```


Info Object for MPI_Spawn

- MPI_Info_create(MPI_Info *info)
- MPI_Info_set(MPI_Info info, char *key, char *value)

Relevant {key, value} for MPI_Spawn:
{soft, a}
{soft, a:b}
{soft, a:b:c}

Can be used to say which hosts to use, and whether to return if it can’t start all processes e.g.
MPI_Info_set(hostinfo, “file”, “spawnhostfile”);
MPI_Info_set(hostinfo, “soft”, soft_limits);

Spawn Example – Master/worker

```
/* master */
#include "mpi.h“
int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  if (world_size != 1) error("Top heavy with management")

  MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
     &universe_sizep, &flag);

  choose_worker_program(worker_program)
  MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
     MPI_INFO_NULL, 0, MPI_COMM_SELF, &everyone,
     MPI_ERRCODES_IGNORE);
…
…
  MPI_Finalize();
  return 0;
}
```

Example continued

```
/* worker */
#include "mpi.h“
int main(int argc, char *argv[]) {

  MPI_Init(&argc, &argv)
  MPI_Comm_get_parent(&parent);
  if (parent == MPI_COMM_NULL) error("No parent!");

  MPI_Comm_remote_size(parent, &size);
  if (size != 1) error("Something's wrong with the parent");
 …
…
…

  MPI_Finalize();
  return 0;
}
```

Communication between applications

- Communication between 2 independently started applications
- One program announces a willingness to accept connections while another attempts to establish a connection
- Once a connection has been established, the two parallel programs share an intercommunicator
- Follows client/server type of communication
- Collective operation
- Operates by means of port names and/or service names

Connecting different MPI Applications

- Climate models are a good example of two separate applications that need to share data
  - Atmosphere needs input from ocean model
    - e.g. effect of large currents on warming
  - Ocean model needs input from atmosphere
    - Atmospheric temperature affects evaporation rates
- Secondary example – visualization of an active application
  - May want to pass information to a visualization engine that is implemented in a separate program
- We'll assume that such programs cannot be spawned within the MPI environment

Establishing Connection

- Similar to sockets but for groups of processes
- Ports, IP addresses, name servers
- Port name (MPI_Open_port, MPI_Comm_accept)
- Service name (MPI_Lookup_name)
- Portability varies

Mediating communication

- Necessary to modify one program to accept a connection from another
  - Must open a port
    - MPI_Open_port(MPI_INFO_NULL,port_name)
  - The connecting application still has to be allowed to connect to communicator
    - MPI_Comm_accept(port_name,MPI_INFO_NULL,0, MPI_COMM_WORLD,client)
    - Collective over comm in 4th argument
    - client returns a new comm (intercommunicator) to the connecting application
    - Can now send to the remote application

API

- MPI_Open_port(MPI_info info, char *port_name) - server
- MPI_Close_port(char *port_name) – server
- MPI_Comm_Accept(char *port_name, MPI_info info, int root, MPI_Comm comm, MPI_Comm newcomm) – server accepts connection and returns a new socket
- MPI_Comm_connect(char *port_name, MPI_info info, int root, MPI_Comm comm, MPI_Comm newcomm) – client connects to server socket
- Also, MPI_Comm_Join(fd, MPI_Comm intercomm)

Example 1

Server:

```
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;
/* ... */
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);
MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF,
    &intercomm);
/* do something with intercomm */
```

Client:

```
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
    &intercomm);
```

Accessing the port

- To connect to the front end application the client must be given the port name
  - Port names are usually character strings
  - The connect to port by calling
    - MPI_Comm_connect(port_name,info,root,comm,newcomm)
    - Returns new communicator
  - Can determine remote communicator size via
    - MPI_Comm_remote_size(newcomm,procs)
  - Communication must concur with that executed in the server program
  - Finally, disconnected from the port
    - MPI_Comm_disconnect(newcomm)

Alternative methods of publishing the port name

- MPI_Publish_name(service,info,port)
  - Allows a given program (specified by service) to post the port name to all MPI infrastructure
  - Call MPI_Unpublish_name to remove data from system before finalizing
- Client then calls MPI_Lookup_name(service,info,port)
  - Returns desired port name
- Possible problem:
  - Two programmers may choose same service name (unlikely)

API – Service names

- MPI_Publish_name(char *service_name, MPI_Info info, char *port_name) – publish a port name for a service
- MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name)
- MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name) – find port name of a service
  - defined by implementation (not required to provide usable name service)

Example 2 – Publishing names

Server:

```
MPI_Open_port(MPI_INFO_NULL, port_name);
MPI_Publish_name("ocean", MPI_INFO_NULL, port_name);
MPI_Comm_accept(port_name, MPI_INFO_NULL, 0, MPI_COMM_SELF,
    &intercomm);
/* do something with intercomm */
MPI_Unpublish_name("ocean", MPI_INFO_NULL, port_name);
```

Client:

```
MPI_Lookup_name("ocean", MPI_INFO_NULL, port_name);
MPI_Comm_connect( port_name, MPI_INFO_NULL, 0,
    MPI_COMM_SELF, &intercomm);
```

Example: 2d Poisson model

```
character*(MPI_MAX_PORT_NAME) port_name
integer client
…
if (myid .eq. 0) then
  call MPI_Open_port(MPI_INFO_NULL, port_name,ierr)
  write(*,*) port_name
endif
call MPI_Comm_accept(port_name,MPI_INFO_NULL,0, &
                       MPI_COMM_WORLD,client,ierr)
!Must send information to the connecting program
call MPI_Gather(mesh_size,1,MPI_INTEGER, &
                MPI_BOTTOM,0,MPI_DATATYPE_NULL, &
                0, client, ierr)
…
! After each iteration process 0 sends its info
if (myid.eq.0) then
  call MPI_Send(it,1,MPI_INTEGER,0,0,client,ierr)
end if
call MPI_Gatherv(mesh,mesh_size,MPI_DOUBLE_PRECISION, &
                 MPI_BOTTOM,0,0, &
                 MPI_DATATYPE_NULL, 0, client, ierr)
…
if (myid .eq.0) then
   call MPI_Close_port(port_name,ierr)
end if
call MPI_Comm_disconnect(client,ierr)
call MPI_Finalize(ierr)
```

Connecting application must also execute this gather

These messages are matched in The connecting application

MPI-2 & threads

- MPI processes are usually conceived as being single threaded
  - Threads carry their own pc, register set and stack
  - Individual threads are not considered to be visible outside the process
- Threads have the potential to improve performance in codes where polling is required
  - Polling thread operates at lower overhead than stopping entire process to poll
  - Equivalent to non-blocking receive
- However, to work effectively the MPI implementation must be *thread safe*
  - Multiple threads can execute message passing calls without causing problems
  - MPI-1 is not guaranteed to be thread safe (some implementations are)

Determining what type of threading is allowed

- MPI_Init_thread(required,provided,ierr)
  - A non-thread safe library will return
    - MPI_THREAD_SINGLE
  - Several user threads supported but only one may make messaging calls (standard interpretation)
    - MPI_THREAD_FUNNELED
  - When all threads make calls, but only one at a time
    - MPI_THREAD_SERIALIZED
  - Finally, for all threads executing messaging at any time “thread compliant”
    - MPI_THREAD_MULTIPLE
- This function can be used to start the program instead of MPI_Init
  - C binding MPI_Init_thread(int *argc, char *** argv, int required, int provided)
- Note only the thread which called the initialization may perform the finalize

Ensuring all processes agree on the number of threads

- MPI does not require that environment variables be propagated to all processes
  - Problematic for OpenMP where number of threads is usually specified by environment variable
  - Most often rank 0 gets the environment but others do not
- Code on right may be used to propagate the thread count to all other processes

```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0) {
  nthreads_str = getenv("OMP_NUM_THREADS");
  if (nthreads_str)
   /* convert string to integer*/
    nthreads = atoi( nthreads_str);
  else
    nthreads = 1;
}
MPI_Bcast(&nthreads,1,MPI_INT,0,
           MPI_COMM_WORLD);
omp_set_num_threads(nthreads);
```

Summary

- Dynamic process management is supported in MPI-2 – MPI_Comm_spawn
- Different MPI applications may connect to one another
  - Important for a class of applications where different solvers require parts of the same dataset
  - Communication is mediated by ports
- Level of thread safety can be determined at compile time