MPI-2

Process creation and management

MPI and process creation

So far we have seen MPI where the number of processes is static during the course of the program.

The number of processes is defined by the parameter supplied when mpirun was launched.

MPI-2 offers the possibility to create additional processes during the course of program execution.

Creating additional processes

Pros: better load balancing, better utilization of available resources

Cons: hard to accomodate on clusters managed by schedulers. Usually the scheduler gives each job a fixed amount of resources which does not vary during execution.

When to spawn processes? Simple example

Possible example: a serial program that runs for 100 minutes, and spends last 50 minutes of its time in a library function.

Parallel version of this function exists, executing it 10 processors takes 5 minutes.

Therefore, simple parallelization gives a program that runs in 55 minutes on 10 processors. In this case the program is using 10 processors during the entire duration of the run, even though 9 processors have no computation to do for 50 minutes.

It would be better to launch the program with 1 process initially, then spawn 9 additional processes after 50 minutes of runtime when the library function is called.

Caveat: careful when spawning processes on scheduled clusters

Schedulers typically assume that pure MPI jobs have a static number of processes.

On SHARCNET, run test programs on development nodes.

Preliminaries: two useful functions

MPI_Get_processor_name

MPI_Get_version

Allow us to better monitor what is happening in our program

MPI_Get_processor_name

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

name - unique specifier for actual node. This must be an array of size at least MPI_MAX_PROCESSOR_NAME.

resultlen - length (in characters) of the name

This function returns the name of node/computer on which the MPI process that calls it is running.

Example output on orca: orc129

This may be highly useful to diagnose problems that arise from a problem node with your code but with a particular node you are using

MPI_Get_version

```
int MPI_Get_version(int *version, int *subversion)

version - version of MPI

subversion - subversion of MPI
```

This function usefully checks which version of MPI the program is running on. In particular, it reports whether one is running version 2.+ , which is required to use MPI-2 commands successfully.

It’s not very likely these days to run into systems with only version 1.+ on them.

Spawning processes

MPI-2 approach: an existing process is selected to launch additional processes.

Additional processes start running a specified new program from the beginning. This program need not be same as the launching program.

These new processes are grouped in a new communicator, which does not contain the launching process. Hence this is an intercommunicator on the launching process.

Why not just fork the launching process (i.e. clone it), like with threads? MPI runs on distributed memory systems so this may not be possible.

MPI_Comm_spawn_multiple

Same as previous, but can start several different binary executables, or same executable with different arguments.

Simply replace arguments in MPI_Comm_spawn with appropriate arrays

MPI_Comm_spawn

```
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info
info,  int root, MPI_Comm comm, MPI_Comm *intercomm,
int array_of_errcodes[])
```

comment - name of program to be spawned (string, significant only at root)

argv - arguments to command (array of strings, s.o.a.t.)

maxprocs - maximum number of processes to start (s.o.a.t)

info - set of key-value pairs telling the runtime system where and how to start the processes (MPI_INFO_NULL is not needed)

root - rank of process where previous arguments examined

comm - intracommunicator containing group of spawning processes

intercomm - intercommunicator between original group and the newly spawned group

array_of_errcodes - array of integers, one code per processe (MPI_ERRCODES_IGNORE if not used)

Intra vs. Inter

Intra-communications - what we have seen so far, involves communication between processes that are members of the same group/communicator.

Inter-communications - communications between processes that are members of different groups/communicators

Example: MPI_Send
If process i calling MPI_Send belongs to communicator COM1 and sends data to process j also in communicator COM1, then COM1 is an intra-communicator on both processes

MPI_Send(...,j,tag,COM1)

If process i calling MPI_Send belongs to communicator COM1 and sends data to process j in a communicator COM2 to which it does not belong, then COM2 is an inter-communicator on process i

MPI_Send(...,j,tag,COM2)

Syntax of point-to-point communication is the same both for inter and intra communicators. Target process addressed by rank in COM2.

Collective communications on intercommunicators

Are these possible? They are not allowed in MPI-1, but are in MPI-2.

Say we have N processes in COM1, M process in COM2
Want to broadcast data located on process i in COM1 to all processes in COM2

Here is how this is done (requires use of special keyword)

```
process i in COM1 calls
MPI_Bcast(&data,1,MPI_INT,MPI_ROOT,COM2)

othe processes in COM1 call
MPI_Bcast(&data,1,MPI_INT,MPI_PROC_NULL,COM2); // no data broadcast

and processes in COM2 call

MPI_Bcast(&data,1,MPI_INT,i,COM1)
```

MPI_Comm_test_inter

```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
```

comm - communicator to test

flag - true if this is an inter-communicator

If you are not sure if a communicator is intra or inter, can check with this function.

Functions to call on child processes

Child processes need to find information about the parent process.

First, child process needs to determine that it is in fact a child process, and not one launched in standard way via mpirun

Also, an inter-communicator must be retrieved to communicate with the previously existing process (the process that did the spawning and others in its communicator)

MPI_Comm_get_parent

```
int MPI_Comm_get_parent(MPI_Comm *parent)
```

parent - parent communicator

Function returns parent communicator for the process it is called on.

If process was not spawned (i.e. is a standard process launched by mpirun), function returns MPI_COMM_NULL

If parent communicator is freed or disconnected, function returns MPI_COMM_NULL

MPI_Comm_remote_size

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

comm - communicator (input)

size - number of processes in the remote group of comm (output)
```

This function determines the size of the remote group associated with an inter-communicator

MPI_UNIVERSE_SIZE

```
MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, &universe_sizep,
&attr_flag)
```

Special attribute available to the MPI implementation which may define the limit on the total number of processes that may be launched.

Simple example of spawning processes

Launch mpirun with 1 process an executable created obtained from compiling the manager program (mpi2_manager.c)

This process will in turn spawn additional child processes, each running the executable obtained form compiling the worker program (mpi2_worker.c) called mpi2_worker

The child processes will send information to the manager process

```
/* MPI2 Manager Code */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int rank, size, namelen, version, subversion, universe_size;
  MPI_Comm family_comm;
  char processor_name[MPI_MAX_PROCESSOR_NAME],
  worker_program[100];
  int rank_from_child,ich;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  MPI_Get_processor_name(processor_name, &namelen);
  MPI_Get_version(&version, &subversion);
  printf("I’m manager %d of %d on %s running MPI %d.%d\n", rank, size,
processor_name, version, subversion);

  if (size != 1) printf("Error: Only one manager process should be
running, but %d were started.\n", size);
```

```
/* set total number of process to be running, including manager */
  universe_size = 4;

  strcpy(worker_program, "./mpi2_worker");
  printf("Spawning %d worker processes running %s\n", universe_size-1,
worker_program);

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

/* communicate with child processes */
  for(ich=0;ich<(universe_size-1);ich++){
   MPI_Recv(&rank_from_child,1,MPI_INT,ich0,family_comm,MPI_STATUS_IGNORE);
   printf("Received rank %d from child %d \n",rank_from_child,ich);
  }

  MPI_Bcast(&rank,1,MPI_INT,MPI_ROOT,family_comm);
  MPI_Comm_disconnect(&family_comm);
  MPI_Finalize();
  return 0;
}
```

```
/* MPI2 Worker Code */

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char** argv)
{
  int rank, size, namelen, version, subversion, psize;
  MPI_Comm parent;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  MPI_Get_processor_name(processor_name, &namelen);
  MPI_Get_version(&version, &subversion);

  printf("I’m worker %d of %d on %s running MPI %d.%d\n", rank, size,
processor_name, version, subversion);
```

```
MPI_Comm_get_parent(&parent);

if (parent == MPI_COMM_NULL) {
  printf("Error: no parent process found!\n");
  exit(1);
}

MPI_Comm_remote_size(parent, &psize);

if (psize != 1) {
  printf("Error: number of parents (%d) should be 1.\n", psize);
  exit(2);
}
```

```
/* communicate with parent process */
  int sendrank = rank;
  printf("Worker %d: Success!\n", rank);

  MPI_Send(&rank,1,MPI_INT,0,0,parent);

  MPI_Bcast(&parent_rank,1,MPI_INT,0,parent);

  printf("Rank received from parent is %d \n",parent_rank);
  MPI_Comm_disconnect(&parent);
  MPI_Finalize();
  return 0;
}
```

Program output

```
ppomorsk@orc129:~/edu_stuff/] mpirun -np 1 ./a.out
I’m manager 0 of 1 on orc129 running MPI 2.1
Spawning 3 worker processes running ./mpi2_worker
I’m worker 2 of 3 on orc129 running MPI 2.1
I’m worker 0 of 3 on orc129 running MPI 2.1
I’m worker 1 of 3 on orc129 running MPI 2.1
Received rank 0 from child 0
Received rank 1 from child 1
Received rank 2 from child 2
Rank received from parent is 0
Rank received from parent is 0
Worker 2: Success!
Worker 0: Success!
Rank received from parent is 0
Worker 1: Success!
```

MPI_Comm_disconnect

int MPI_Comm_disconnect(MPI_Comm *comm)

comm - disconnect process from communicator, value of comm becomes MPI_COMM_NULL

Waits for all pending communications to complete.

Cannot be applied to MPI_COMM_WORLD or MPI_COMM_SELF

This function needs to be called on all processes that are connected through the communicator

Disconnected groups of processes can perform MPI_Finalize separately (as this is a collective operation on all processes)

So need to use this function if you would like only the spawned processes to be terminated, but the manager process to continue (and possibly due something else)

Communicators can be manipulated further

MPI_Intercomm_merge - creates an intracommunicator from an intercommunicator

MPI_Intercomm_create - creates an intercommunicator from two intracommunicators

MPI_Intercomm_merge

```
int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm
*newintracomm)
```

intercomm - intercommunicator
high - used to order groups withing comm (logical) when creating the new communicator. Boolean value. Group that sets true has its process ordered after the group that sets it to false. If all processes provide same value, the choice of which group is first is arbitrary.

newintracomm - created intracommunicator

In our example code, let’s do this to join the old and new processes into a single new intracommunicator. Very convenient.

```
 /* mpi2_manager */
...
  MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, universe_size-1,
MPI_INFO_NULL, 0, MPI_COMM_SELF, &family_comm, MPI_ERRCODES_IGNORE);

  MPI_Intercomm_merge(family_comm, 1, &allcomm);
  MPI_Comm_rank(allcomm, &globalrank);
  printf("manager: global rank is %d , rank is %d \n",globalrank,
rank);
  MPI_Allreduce(&globalrank,&sumrank,1,MPI_INT,MPI_SUM,allcomm);
  printf("sumrank after allreduce on process %d is %d \n", rank,
sumrank);
  MPI_Finalize();
  return 0;
}
```

```
 /* mpi2_worker */
...

  MPI_Intercomm_merge(parent, 1, &allcom);
  MPI_Comm_rank(allcom, &globalrank);
  printf("worker: global rank is %d , rank is %d \n",globalrank, rank);

  MPI_Allreduce(&globalrank,&sumrank,1,MPI_INT,MPI_SUM,allcom);
  printf("sumrank after allreduce on process %d is %d \n", rank,
sumrank);

  MPI_Finalize();
  return 0;
}
```

Program output

```
[ppomorsk@orc130:~/edu_stuff/] mpirun -np 1 ./a.out
I’m manager 0 of 1 on orc130 running MPI 2.1
Spawning 3 worker processes running ./mpi2_worker
I’m worker 2 of 3 on orc130 running MPI 2.1
Worker 2: Success!
I’m worker 0 of 3 on orc130 running MPI 2.1
Worker 0: Success!
I’m worker 1 of 3 on orc130 running MPI 2.1
Worker 1: Success!
worker: global rank is 2 , rank is 1
worker: global rank is 3 , rank is 2
worker: global rank is 1 , rank is 0
manager: global rank is 0 , rank is 0
sumrank after allreduce on process 0 is 6
sumrank after allreduce on process 2 is 6
sumrank after allreduce on process 1 is 6
sumrank after allreduce on process 0 is 6
```