MPI-1 Introduction/Tutorial

Jan Hamaekers

Institut für Numerische Simulation
Rheinische Friedrich-Wilhelms-Universität Bonn

Seminar Technische Numerik
7. Januar 2010, Bonn

Outline

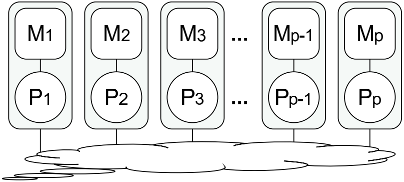
References

This talk is based on tutorials and documents from:

- `http://www.mpi-forum.org/`
- `http://www.mcs.anl.gov/research/projects/mpi/`
- `http://www.mcs.anl.gov/research/projects/mpi/tutorial/`
- `http://www.lam-mpi.org/`
- `http://www.lam-mpi.org/tutorials/nd/`
- `http://www.open-mpi.org/`
- `http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/CommonDoc/MessPass/MPI.html`
- `https://fs.hlrs.de/projects/par/par_prog_ws/`

Parallelization

- Distributed memory



- Serial vs. Parallel
  - Serial: e.g.: $\mathcal{O}(N)$, $\mathcal{O}(N \log(N))$
  - Parallel: e.g.: $\mathcal{O}(N/P)$, $\mathcal{O}(N \log(N)/\sqrt{P})$
- Two primary programming paradigms:
  - SIMD (single instruction multiple data)
    - Write single program that will perform same operation on multiple sets of data
  - MIMD (multiple instruction multiple data)
    - Write different programs to perform different operations on multiple sets of data
- MPI can be used for either paradigm

M P I = Message Passing Interface

- MPI is a standard protocol in terms of user interface.
  - By itself, it is NOT a library - but rather the specification of what such a library should be.
- Why using MPI ?
  - Standardization - MPI is the only message passing library which can be considered a standard.
  - Portability - There is no need to modify your source code when you port your application to a different platform that supports the MPI standard. (in particular: heterogeneous systems)
  - Performance Opportunities - Vendor implementations should be able to exploit native hardware features to optimize performance.
  - Functionality - Over 115 routines are defined in MPI-1 alone.
  - Availability - A variety of implementations are available, both vendor and public domain. (mpich, lam/OpenMPI)

M P I = Message Passing Interface

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs.
- MPI implementations are a combination of MPI-1 and MPI-2.
  - A few implementations include the full functionality of both.
  - This talk: MPI-1
    - Point-to-point communication
    - Collective communication
    - Communicators
  - Next talk: MPI-2
    - spwan new processes
    - one-sided communication
    - parallel I/O
    - debugging

The Six Necessary MPI Commands

- **int** MPI_Init(**int** *argc, **char** **argv)
- **int** MPI_Finalize(**void**)
- **int** MPI_Comm_size(MPI_Comm comm, **int** *size)
- **int** MPI_Comm_rank(MPI_Comm comm, **int** *rank)
- **int** MPI_Send(**void** *buf, **int** count, MPI_Datatype datatype,
  int dest, **int** tag, MPI_Comm comm)
- **int** MPI_Recv(**void** *buf, **int** count, MPI_Datatype datatype,
  int source, **int** tag, MPI_Comm comm, MPI_Status *status)

The Six Necessary MPI Commands

- **int** MPI_Init(**int** *argc, **char** **argv)
  - Initiates MPI
- **int** MPI_Finalize(**void**)
  - Shuts down MPI
- **int** MPI_Comm_size(MPI_Comm comm, **int** *size)
  - Find out number of processes
- **int** MPI_Comm_rank(MPI_Comm comm, **int** *rank)
  - Find out identifier of current process
  - rank is in [0:size-1]

Hello World

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

main(int *argc, char** argv)
{
  int size, rank;

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

  printf("Hello world! I'm  %d  of  %d\n",
         rank+1, size);

  MPI_Finalize();
}
```

Hello World

- mpicc –o hello hello.c
  - ```
    /opt/packages/mpich/bin/mpicc -show
    cc -DUSE_STDARG -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1
       -DHAVE_UNISTD_H=1
       -DHAVE_STDARG_H=1 -DUSE_STDARG=1 -DMALLOC_RET_VOID=1
       -L/opt64/packages/mpich-1.2.7p1/lib
       -lmpich -lpthread -lrt
    ```
- mpiCC –o hello hello.cc or mpicxx –o hello hello.cc
  - ```
    /opt/packages/mpich/bin/mpiCC -show
    g++ -DUSE_STDARG -DHAVE_STDLIB_H=1 -DHAVE_STRING_H=1
        -DHAVE_UNISTD_H=1
        -DHAVE_STDARG_H=1 -DUSE_STDARG=1 -DMALLOC_RET_VOID=1
        -L/opt64/packages/mpich-1.2.7p1/lib
        -lpmpich++ -lmpich -lpthread -lrt
    ```
- Try: /opt/packages/mpich/bin/mpicc –help
  - e.g. mpicc –cc=gcc–4.3 –o hello hello.c

Hello World

- mpirun −np size hello
  - Try: /opt/packages/mpich/bin/mpirun −help
    - e.g. mpirun −np 2 −dbg=gdb hello

```
mpirun -np 4 hello
Hello world! I'm  1 of  4
Hello world! I'm  2 of  4
Hello world! I'm  3 of  4
Hello world! I'm  4 of  4
```

The Six Necessary MPI Commands

- **int** MPI_Init(**int** *argc, **char** **argv)
- **int** MPI_Finalize(**void**)
- **int** MPI_Comm_size(MPI_Comm comm, **int** *size)
- **int** MPI_Comm_rank(MPI_Comm comm, **int** *rank)
- **int** MPI_Send(**void** *buf, **int** count, MPI_Datatype datatype,
  int dest, **int** tag, MPI_Comm comm)
- **int** MPI_Recv(**void** *buf, **int** count, MPI_Datatype datatype,
  int source, **int** tag, MPI_Comm comm, MPI_Status *status)

Point-to-point communication

- **int** MPI_Send(**void** *buf, **int** count, MPI_Datatype datatype, **int** dest, **int** tag, MPI_Comm comm)
  - Send message of length `count` ~~bytes~~
    and datatype `datatype`
    contained in `buf`
    with tag `tag`
    to process number `dest`
    in communicator `comm`
  - MPI_Send(&x, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD)

Point-to-point communication

- **int** MPI_Recv(**void** *buf, **int** count, MPI_Datatype datatype,
  int source, **int** tag, MPI_Comm comm, MPI_Status *status)
  - Receive message of length `count` bytes
    and datatype `datatype`
    with tag `tag`
    in buffer `buf`
    from process number `source`
    in communicator `comm`
    and record status `status`
  - MPI_Recv(&x, 1, MPI_DOUBLE, source, tag,
    MPI_COMM_WORLD, &status)
  - MPI_ANY_TAG, MPI_ANY_SOURCE
- Getting information

```
MPI_Status status;
MPI_Recv(..., &status);
recvd_tag = status.MPI_TAG;
recvd_source = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recvd_count);
```

Simple program

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv)
{
  int i, rank, size, dest;
  int to, src, from, count, tag;
  int st_count, st_source, st_tag;
  double data[10];
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Process %d of %d is alive\n", rank, size);
  dest = size - 1;
  src = 0;
```

Simple program

```
if (rank == src) {
  to = dest;
  count = 10;
  tag = 2010;
  for (i = 0; i < 10; i++)
  data[i] = i;
  MPI_Send(data, count, MPI_DOUBLE, to, tag,
           MPI_COMM_WORLD);
}
```

Simple program

```
else if (rank == dest) {
  tag = MPI_ANY_TAG;
  count = 10;
  from = MPI_ANY_SOURCE;
  MPI_Recv(data, count, MPI_DOUBLE, from, tag,
           MPI_COMM_WORLD, &status);
  MPI_Get_count(&status, MPI_DOUBLE, &st_count);
  st_source= status.MPI_SOURCE;
  st_tag= status.MPI_TAG;
  printf("Status␣info:␣source␣=␣%d,␣tag␣=␣%d,␣count␣
  st_source, st_tag, st_count);
  printf("␣%d␣received:␣", rank);
  for (i = 0; i < st_count; i++)
  printf("%2.1f␣", data[i]);
  printf("\n");
}
MPI_Finalize();
return 0;
}
```

Simple program

```
Process 0 of 4 is alive
Process 1 of 4 is alive
Process 2 of 4 is alive
Process 3 of 4 is alive
Status info: source = 0, tag = 2010, count = 10
 3 received: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.
```

MPI Datatypes

MPI datatype	C datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	
`MPI_PACKED`	user-defined structure

Also user-defined:

MPI_TYPE_VECTOR(count, blocklen, stride, oldtype, newtype)

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)

Deadlocks

- Blocking communication
  - MPI_SEND does not complete until buffer is available for reuse.
  - MPI_RECV does not complete until buffer is available for use.
- Example showing a communication pattern that leads to a deadlock

```
Process 0      Process 1
Recv(1)        Recv(0)
Send(1)        Send(0)
```

  - In this example, the RECV call is blocking.
- Example that is unsafe to use is

```
Process 0      Process 1
Send(1)        Send(0)
Recv(1)        Recv(0)
```

  - This situation is correct logically, but it depends on how the blocking is implemented for the SEND and RECV.

Avoiding deadlock

- The following scenario is always safe because each SEND is matched by a corresponding RECV before the other SEND starts.

```
Process 0      Process 1
Send(1)        Recv(0)
Recv(1)        Send(0)
```

- Some Solutions
  - Reorder the communciations as shown in the example above
  - Use the MPI_Sendrecv, which supplies the receive buffer at the same time as the send buffer.

```
Process 0      Process 1
Sendrecv(1)    Sendrecv(0)
```

Avoiding deadlock

- Use non-blocking ISend or IRecv.
  - Non-blocking send:
    MPI_Isend(..., request); doing some other work;
    MPI_Wait(request, status);
  - Non-blocking receive:
    MPI_Irecv (..., request); doing some other work;
    MPI_Wait(request, status);
- MPI_Test(request, flag, status)
- Other variations of MPI_Test, MPI_Wait and MPI_Probe can be used to check on multiple operations.

Avoiding deadlock

- e.g.: The Send/Recv does not block in this case, but you must check for compeletion of communications before using the buffers:

```
Process 0       Process 1
ISend(1)        ISend(0)
IRecv(1)        IRecv(0)
Waitall         Waitall
```

or

```
Process 0       Process 1
IRecv(1)        IRecv(0)
Send(1)         Send(0)
Wait            Wait
```

Collective Communication

- Two simple collective operations:
  - MPI_BCAST(buffer, count, datatype, root, comm)
  - MPI_REDUCE(sendbuf, recvbuf, count, datatype, operation, root, comm)
- The routine MPI_BCAST sends data from one process to all others.
- The routine MPI_REDUCE combines data from all processes returning the result to a single process.
- MPI_SCATTER, MPI_GATHER and many more: MPI_ALLGATHER MPI_ALLGATHERV MPI_ALLREDUCE MPI_ALLTOALL MPI_ALLTOALLV MPI_BCAST MPI_GATHER MPI_GATHERV MPI_REDUCE MPI_REDUCESCATTER MPI_SCAN MPI_SCATTER MPI_SCATTERV
- A collective operation must be called by all processes in the communicator.
- Use these whenever possible !

Built-in Collective Computation Operations

MPI Name	Operation
`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_PROD`	Product
`MPI_SUM`	Sum
`MPI_LAND`	Logical and
`MPI_LOR`	Logical or
`MPI_LXOR`	Logical exclusive or (xor)
`MPI_BAND`	Bitwise and
`MPI_BOR`	Bitwise or
`MPI_BXOR`	Bitwise xor
`MPI_MAXLOC`	Maximum value and location
`MPI_MINLOC`	Minimum value and location

Also user-defined operations possible.

Communicators

- Pre-defined communicators
  - MPI_COMM_WORLD: Contains all processes available at the time the program was started.
  - MPI_COMM_SELF: Contains only the local process. Not used to communicate.
- User-defined communicators
  - **int** MPI_Comm_split(MPI_Comm comm,
    int color, **int** key, MPI_Comm *newcomm)
    - Each subgroup (`newcomm`) contains all processes having the same `color`. Within each subgroup, processes are ranked in the order defined by the value of the argument `key`.
- Inter communicators
  - An inter-communication is a point-to-point communication between processes in intra-communicators.

    int MPI_Intercomm_create(MPI_Comm local_comm, **int** local_leader,
    MPI_Comm peer_comm, **int** remote_leader,
    int tag, MPI_Comm *newintercomm)

Simple Program 2

```c
#include <stdio.h>
#include <mpi.h>
main(int argc, char **argv) {
  MPI_Comm new_comm, inter_comm;
  MPI_Status status;
  int myrank, size, even, value, localrank, interrank;
  int sendi, recvi;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  even = ((myrank % 2) == 0); // Odd, even ?
  /* Split comm into two comms */
  MPI_Comm_split(MPI_COMM_WORLD, even, myrank,
    &new_comm);
  value = myrank;
  MPI_Bcast(&value, 1, MPI_INT, 0, new_comm);
  MPI_Comm_rank(new_comm, &localrank);
  printf("Rank %d LocalRank %d: Got broadcast value of %d\n"
  myrank, localrank, value);
```

Simple Program 2

```
  MPI_Intercomm_create(new_comm, 0,
    MPI_COMM_WORLD, (even ? 1 : 0),
    2010, &inter_comm);
  MPI_Comm_rank(new_comm, &interrank);
  sendi = recvi = myrank;
  if (myrank < (size/2)*2) {
    MPI_Sendrecv( &sendi, 1, MPI_INT, interrank, 2020,
      &recvi, 1, MPI_INT, interrank, 2020,
      inter_comm, &status );
    printf("Rank %d LocalRank %d: SendRecv between %d %d\n",
    myrank, localrank, sendi, recvi);
  }
  MPI_Finalize();
  return 0;
}
```

Simple Program 2

```
mpirun -np 5 simple2
Rank 0 LocalRank 0: Got broadcast value of 0
Rank 1 LocalRank 0: Got broadcast value of 1
Rank 2 LocalRank 1: Got broadcast value of 0
Rank 3 LocalRank 1: Got broadcast value of 1
Rank 4 LocalRank 2: Got broadcast value of 0
Rank 0 LocalRank 0: SendRecv between 0 1
Rank 1 LocalRank 0: SendRecv between 1 0
Rank 2 LocalRank 1: SendRecv between 2 3
Rank 3 LocalRank 1: SendRecv between 3 2
```

Outlook

- Persistent communication
- Groups
- Topologies
- Debugging, profiling: e.g.
  - ddt, -dgb=gdb, valgrind (wrapper),
  - -mpilog, -mpitrace, -mpianim
- MPI-2
  - spwan new processes
  - one-sided communication
  - parallel I/O
  - more debugging

Thanks !