Distributed and Parallel Technology

Communication Libraries I
Introduction to MPI

Hans-Wolfgang Loidl
http://www.macs.hw.ac.uk/~hwloidl

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh

No proprietary software has been used in producing these slides
Based on earlier versions by Greg Michaelson and Patrick Maier

Parallel Programming Models

Shared memory
- 1 process
  - *single* address space
  - but many *threads*
- data "transfer": read/write access to shared data
- synchronisation: locking shared data

Message passing
- many processes
  - each with its *own* address space
- data transfer: send/recv messages
- synchronisation: send/recv messages

Programming model $\neq$ parallel architecture
- Can implement message passing on shared memory architecture
- Can implement (virtual) shared memory by message passing

What is MPI?

- A *message passing library specification*
  - advanced message passing programming model (send, receive, broadcast, barriers, ...)
  - *not a specific library implementation*
- Many MPI implementations exist
  - We will use MPICH 3.1, which implements the MPI-3.0 standard
  - `http://www.mcs.anl.gov/research/projects/mpi/`
- Bindings to many programming languages exist
  - We will use C binding
- MPI is based on an SPMD model
  - *Single Program Multiple Data*
  - every processor runs the same program
  - SPMD $\neq$ SIMD: Different processors may (and will in general) execute different instructions at the same time.

History of MPI

- Apr ’92
  - Workshop on *Standards for Message Passing in a Distributed Memory Environment*
- Nov ’92 – Apr ’94
  - Message Passing Interface Forum
  - all big players round one table
- May ’94
  - MPI version 1.0
- Apr ’97
  - MPI-2
- Sep ’09
  - MPI-2.2
- Nov ’14
  - MPI-3.0

Compiling and Running C with MPI

Hosts

- 32-node Beowulf cluster
- hostnames: `bwlf01` ... `bwlf32`
- These machines run the same Linux distribution as the lab machines

```
# cat /etc/redhat-release
CentOS release 6.8 (Final)
```

- Most packages are installed in `/usr/lib64/mpich`

Environment variables

- `$PATH` must include `/usr/lib64/mpich/bin`

Compiling and Running C with MPI

MPI Setup

- Log into any `bwlf??`, eg. `ssh -X bwlf01`
- Check where the system finds the MPI binaries like this

```
# which mpicc
/usr/lib64/mpich/bin/mpicc
```

- Test the MPI configuration by

```
# mpichversion
MPICH Version: 3.1
... a lot of stuff ...
```


Compiling and Running C with MPI

Get the sample sources

Download the sample sources from the web page:
`http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/index.html#sample_cmpi`
or do this on the command line:

```
# wget http://www.macs.hw.ac.uk/ hwloidl/Courses/F21DP/srcs/hello2.c
# wget http://www.macs.hw.ac.uk/ hwloidl/Courses/F21DP/srcs/mpi04
```

You now have a hello world program in file `hello2.c` and a host file that we will use later in `mpi04`.

Compiling and Running C with MPI

Compiling

To compile you need to use an MPI-enabled compiler.

- Compile (with warnings and optimisation)

```
# mpicc -Wall -O -o file file.c
```

E.g.

```
# mpicc -Wall -O -o hello2 hello2.c
```

Compiling and Running C with MPI

Executing

- Run the executable `file` on `p` processors

 # mpirun -n p -hosts hoststring file arg1 arg2 ...

  - Copies `file` to `p` processors
  - Executes each copy with arguments `arg1 arg2 ...`
  - Uses the machine names specified in the string `hoststring` (a comma-separated list of machine names)

- E.g.

 # mpirun -n 4 -hosts "bwlf01,bwlf02,bwlf03,bwlf04" ./hello2
Hello, I am 0 of 4 (hostname is bwlf01)
Hello, I am 2 of 4 (hostname is bwlf03)
Hello, I am 1 of 4 (hostname is bwlf02)
Hello, I am 3 of 4 (hostname is bwlf04)

Compiling and Running C with MPI

Executing

- To avoid typing the string of machine names every time, you can specify a file that contains all machine names and pass it like this:

 # mpirun -n p -f hostfile file arg1 arg2 ...

  - Copies `file` to `p` processors
  - Executes each copy with arguments `arg1 arg2 ...`
  - Uses the machine names specified in `hostfile` (1 name per line)

 # mpirun -n 4 -f mpi4 hello2
Hello, I am 0 of 4 (hostname is bwlf01)
Hello, I am 2 of 4 (hostname is bwlf03)
Hello, I am 1 of 4 (hostname is bwlf02)
Hello, I am 3 of 4 (hostname is bwlf04)

Side remark: In older versions of mpich you needed to start a demon before you could launch a multi-node execution. This is no longer needed and just providing the file with hostnames using the `-f` option should be sufficient. For details see the mpich-3.1 user's guide.

Compiling and Running C with MPI

Host file

- The user can supply their own host file `hosts`

```
% mpirun -f hosts -n p file arg1 arg2 ...
```

- Use `bping` or this bash script to filter live hosts

```
wget http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/mpiAll
echo -n > bwlfLive
for host in `cat mpiAll`; do
    ping -q -c 1 -w 1 $host && echo $host >> bwlfLive
done
```

- To run a shell command on all machines, use the command `brsh` and the environment variable `BEONODES`

```
# wget http://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/mpi04
# which brsh
/home/hwloidl/bin/beotools/brsh
# export BEONODES="`cat mpi04`"
# echo $BEONODES
bwlf01 bwlf02 bwlf03 bwlf04
# brsh hostname
bwlf01 bwlf01
bwlf02 bwlf02
bwlf03 bwlf03
bwlf04 bwlf04
```

[0] The ` symbol is a backtick symbol (unicode: \x60)

Hello World

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

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

  MPI_Init(&argc, &argv);                  /* start "virtual machine" */
  MPI_Comm_size(MPI_COMM_WORLD, &p); /* get size of VM */
  MPI_Comm_rank(MPI_COMM_WORLD, &id);/* get own rank in VM */

  printf("I am %d of %d\n", id, p);      /* payload */

  MPI_Finalize();                          /* shut down VM */
  return 0;
}
```

Red tape is the same in all MPI programs — only payload varies.

Running Hello World

```
% mpicc -Wall -O -o hello hello.c
% mpirun -n 5 hello
I am 0 of 5
I am 2 of 5
I am 3 of 5
I am 4 of 5
I am 1 of 5
% mpirun -n 5 hello
I am 0 of 5
I am 3 of 5
I am 2 of 5
I am 1 of 5
I am 4 of 5
```

```
% mpirun -n 5 hello
I am 0 of 5
I am 2 of 5
I am 1 of 5
I am 4 of 5
I am 3 of 5
% mpirun -n 5 hello
I am 0 of 5
I am 1 of 5
I am 4 of 5
I am 2 of 5
I am 3 of 5
```

Order of output is random even if all processors reside on the same machine.

MPI Red Tape Explained

- `MPI_Init(&argc, &argv);`
  - initializes MPI (must be called before any other MPI functions)
- `MPI_Finalize();`
  - shuts down MPI (and frees any resources allocated by MPI)
- `MPI_Comm_size(MPI_COMM_WORLD, &p);`
  - `p` = #processors (as given by option `-np`) MPI is running on
  - processors may be virtual (eg. with option `-all-local`)
- `MPI_Comm_rank(MPI_COMM_WORLD, &id);`
  - `id` = rank of this processor
  - `0 <= id < p`

Aside: MPI organizes processors into groups called *communicators*. `MPI_COMM_WORLD` is the top level communicator, consisting of all processors allocated by `mpirun` on startup.

Basic Point to Point Communication in MPI

MPI offers two basic point to point communication functions:

- `MPI_Send(message, count, datatype, dest, tag, comm)`
  - Blocks until `count` items of type `datatype` are sent from the `message` buffer to processor `dest` in communicator `comm`.
    - ★ `message` buffer may be reused on return, but message may still be in transit!
- `MPI_Recv(message, count, datatype, source, tag, comm, status)`
  - Blocks until receiving a `tag`-labelled message from processor `source` in communicator `comm`.
  - Places the message in `message` buffer.
    - ★ `datatype` must match datatype used by sender!
    - ★ Receiving fewer than `count` items is OK, but receiving more is an error!

Aside: Many parallel programs can be written using just the basic `MPI_Send` and `MPI_Recv` (and the red tape we saw in Hello World).

Send and Receive in more Detail

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

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

- `message` pointer to send/receive buffer
- `count` number of data items to be sent/received
- `datatype` type of data items
- `comm` communicator of destination/source processor
  - For now, use default communicator `MPI_COMM_WORLD`
- `dest/source` rank (in `comm`) of destination/source processor
  - Pass `MPI_ANY_SOURCE` to `MPI_Recv()` if source is irrelevant
- `tag` user defined message label
  - Pass `MPI_ANY_TAG` to `MPI_Recv()` if tag is irrelevant
- `status` pointer to struct with info about transmission
  - Info about source, tag and #items in message received

MPI Datatypes

Datatypes serve as descriptors of the data to be sent/received

- Tell the system how to pack/unpack/convert the data.
- Simple semantics for basic builtin datatypes; not so simple for complex user defined types.

Basic builtin datatypes correspond to simple C types:

MPI_Datatype constant	C type
MPI_CHAR	char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Example: Number Guessing Game

- 2-player game: *thinker* and *guesser*
- Thinker thinks of a number between 1 and 100.
- Guesser guesses.
- Thinker replies whether the guess is high, low or correct.
- If not correct, guesser guesses again, and so on...

Implement this game as MPI program running on two processors.

- Thinker on processor 0
  - Receives integer guesses
  - Sends replies as characters h, l or c
- Guesser on processor 1
  - Sends integer guesses
  - Receives characters

Aside: This is a distributed, non-parallel program, because the game is turn-based and thus inherently sequential.

Number Guessing Game — main()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mpi.h>
#include <time.h>

int main(int argc, char ** argv)
{
  int p, id;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if (id == 0)
    thinker();   /* Thinker on processor 0 */
  else
    guesser();   /* Guesser on processor 1 */

  MPI_Finalize();
  return 0;
```


Number Guessing Game — thinker()

```
void thinker()
{
  int number, guess;
  char reply;
  MPI_Status status;

  srand((unsigned int)time(NULL));

  reply = 'x';
  number = rand() % 100 + 1;
  printf("0: (I'm thinking of %d)\n",number);
  while (reply != 'c') {
    MPI_Recv(&guess, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
    if (guess == number)
      reply = 'c';
    else if (guess > number)
      reply = 'h';
    else
      reply = 'l';
    printf("0: 1 guessed %2d; I'm responding %c\n", guess, reply);
    MPI_Send(&reply, 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
  }
}
```

Number Guessing Game — guesser()

```
void guesser()
{
  int guess, high, low;
  char reply;
  MPI_Status status;

  sleep(1); srand((unsigned int)time(NULL));

  low = 1;
  high = 100;
  guess = rand() % 100 + 1;
  printf("1: I'm guessing %2d\n", guess);
  while (1) {
    MPI_Send(&guess, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&reply, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
    switch (reply) {
      case 'c': printf("1: 0 replied %c\n", reply); return;
      case 'h': high = guess; break;
      case 'l': low = guess; break;
    }
    guess = (high + low) / 2;
    printf("1: 0 replied %c; I'm guessing %2d\n", reply, guess);
  }
}
```


Running the Game

```
% mpirun -n 2 guess
0: (I'm thinking of 77)
1: I'm guessing 33
0: 1 guessed 33; I'm responding l
1: 0 replied l; I'm guessing 66
0: 1 guessed 66; I'm responding l
1: 0 replied l; I'm guessing 83
0: 1 guessed 83; I'm responding h
1: 0 replied h; I'm guessing 74
0: 1 guessed 74; I'm responding l
1: 0 replied l; I'm guessing 78
0: 1 guessed 78; I'm responding h
0: 1 guessed 76; I'm responding l
1: 0 replied h; I'm guessing 76
0: 1 guessed 77; I'm responding c
1: 0 replied l; I'm guessing 77
1: 0 replied c
```

```
% mpirun -n 2 guess
0: (I'm thinking of 2)
1: I'm guessing 16
0: 1 guessed 16; I'm responding h
0: 1 guessed  8; I'm responding h
1: 0 replied h; I'm guessing  8
1: 0 replied h; I'm guessing  4
0: 1 guessed  4; I'm responding h
1: 0 replied h; I'm guessing  2
0: 1 guessed  2; I'm responding c
1: 0 replied c
```

Output from processors is merged at random

- Causal or chronological ordering lost (but may be reconstructible)

Naive Parallel Matrix Multiplication

Matrix multiplication M3 = M1 * M2

- M1 m*n matrix, M2 n*m matrix, M3 m*m matrix
- M3 defined via dot product: M3[i][j] = (row i of M1) * (column j of M2)

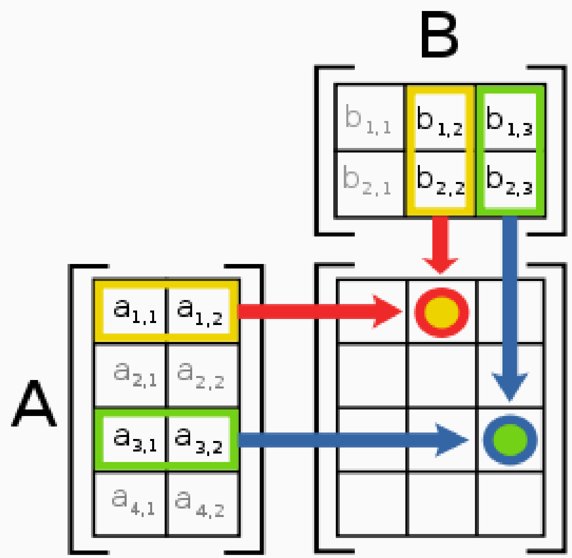
Naive parallelisation:

```
Send m and n to every processor
for i from 1 to m
  for j from 1 to m
    Send row i of M1 and column j of M2 to processor j+1
  for j from 1 to m
    Receive M3[i][j] from processor j+1
```

Problem: C stores matrices row-wise. How to transmit columns of M2?

- Simple solution: Transmit rows of *transposed* matrix.

Matrix Multiplication



[0] Picture from Wikipedia

Naive Parallel Matrix Mult — Auxiliary Functions

```
/* transpose an m*n matrix into a n*m matrix */
int ** transpose(int ** M, int m, int n)
{
  int ** MT;
  int i, j;
  MT = allocMatrix(n, m);
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      MT[j][i] = M[i][j];
  return MT;
}

/* compute the dot product of two vectors of length n */
int dotProd(int * V1, int * V2, int n)
{
  int dp = 0;
  int i;
  for (i = 0; i < n; i++)
    dp = dp + V1[i] * V2[i];
  return dp;
}
```


Naive Parallel Matrix Mult — main()

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

int main(int argc, char ** argv)
{
  int p, id, m, n;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if (id == 0) {
    FILE * fin = fopen(argv[1], "r");
    fscanf(fin, "%d %d", &m, &n);
    int ** M1 = allocMatrix(m,n);
    int ** M2 = allocMatrix(n,m);
    readMatrix(fin, M1, m, n);
    readMatrix(fin, M2, n, m);
    fclose(fin);
```

(continues on next slide)


```
    int i;
    for(i = 1; i < p; i++) {
      MPI_Send(&m, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
      MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    }

    if (p == m+1) {
      int ** M3 = matrixProdMaster(M1, M2, m, n);
      writeMatrix(stdout, M3, m, m);
    }
    else
      printf("Must have %d processors\n", m+1);
  }
  else {
    MPI_Status status;

    MPI_Recv(&m, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

    if (p == m+1)
      matrixProdWorker(m, n);
  }

  MPI_Finalize();
  return 0;
}
```


Naive Parallel Matrix Mult — matrixProdMaster()

```
/* return m*m matrix as the product of m*n matrix M1 and n*m matrix M2;
   employ m workers, each computing 1 row of the product matrix */
int ** matrixProdMaster(int ** M1, int ** M2, int m, int n)
{
  int i, j;
  MPI_Status status;

  int ** M2T = transpose(M2, n, m);
  int ** M3 = allocMatrix(m,m);
  for (i = 0; i < m; i++) {
    for (j = 0; j < m; j++) {
      MPI_Send(M1[i], n, MPI_INT, j+1, 0, MPI_COMM_WORLD);
      MPI_Send(M2T[j], n, MPI_INT, j+1, 0, MPI_COMM_WORLD);
    }

    for(j = 0; j < m; j++)
      MPI_Recv(&(M3[i][j]), 1, MPI_INT, j+1, 0, MPI_COMM_WORLD, &status
  }
  return M3;
}
```

Naive Parallel Matrix Mult — matrixProdWorker()

```
/* compute 1 row of an m*m product matrix */
void matrixProdWorker(int m, int n)
{
  MPI_Status status;
  int dp, i;
  int * R = allocVector(n);
  int * C = allocVector(n);

  for (i = 0; i < m; i++)
  {
    MPI_Recv(C, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(R, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    dp = dotProd(C, R, n);
    MPI_Send(&dp, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }
}
```


Naive Parallel Matrix Mult

Master explained:

- `MPI_Send(M1[i], n, MPI_INT, j+1, 0, MPI_COMM_WORLD);`
  - Send row `i` of `M1` as vector of n integers to processor `j+1`
- `MPI_Send(M2T[j], n, MPI_INT, j+1, 0, MPI_COMM_WORLD);`
  - Send row `j` of `M2T` (= column `j` of `M2`) as vector of n integers to processor `j+1`
- `MPI_Recv(&(M3[i][j]),1,MPI_INT,j+1,0,MPI_COMM_WORLD,&status);`
  - Receive 1 integer from processor `j+1` and store it in `M3[i][j]`
- Note: All tags are 0.

Worker is much simpler:

- Receive `m` pairs of vectors of length `n`
- Compute and send their dot product

Question: Why is this a poor parallelisation?