Parallel and Distributed Computing

COS 436

Dr. Tom Nurkkala
tnurkkala@cse.taylor.edu

Taylor University
Computer Science & Engineering

Spring 2018

Overview

MPI—Message Passing Interface

- Standard for message passing computation
- Cross-platform (desktop to supercomputer)
- Language agnostic (C, FORTRAN, C++, ...)

MPICH Implementation http://www.mpich.org/

- Open source (Git)
- 'CH' from *Chameleon* portability system
- MPICH1 (1992), MPICH2 (2001), MPICH3 (2012)
- Collaborators (partial)
  - University of British Columbia, Ohio State
  - Microsoft, IBM, Cray

OpenMPI Implementation https://www.open-mpi.org/

- Open source (Git)
- Collaborators (partial)
  - Auburn, Wisconsin at La Crosse, Michigan
  - Los Alamos National Lab, Oak Ridge National Lab, Sandia National Lab
  - Amazon, AMD, ARM, Broadcom, Cisco, Facebook, Fujitsu, IBM, Intel, *n*VIDIA, Oracle

Communication Domain

- Processes that can communicate with each other
- Stored in communicator
- Communicator type: `MPI_Comm`
- Predefined default: `MPI_COMM_WORLD`

```
1  /* Set up */
2  int MPI_Init(int *argc, char ***argv);
3
4  /* Tear down */
5  int MPI_Finalize();
6
7  /* Total processes */
8  int MPI_Comm_size(MPI_Comm comm, int *size);
9
10 /* Local process index */
11 int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

Typical MPI Initialization

```
1  int
2  main (int argc, char **argv)
3  {
4    int num_procs;
5    int rank;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10   printf("%d: hello (p=%d)\n", rank, num_procs);
11
12   /* Do many things, all at once */
13
14   MPI_Finalize();
15 }
```

Primitive Communication

```
1  /* Send */
2  int MPI_Send(void *buf, int count,
3               MPI_Datatype datatype,
4               int dest, int tag,
5               MPI_Comm comm)
6
7  /* Receive */
8  int MPI_Recv(void *buf, int count,
9               MPI_Datatype datatype,
10              int source, int tag,
11              MPI_Comm comm,
12              MPI_Status *status)
```

MPI Datatypes

```
1 MPI_CHAR
2 MPI_SIGNED_CHAR
3 MPI_UNSIGNED_CHAR
4 MPI_BYTE
5 MPI_WCHAR
6 MPI_SHORT
7 MPI_UNSIGNED_SHORT
8 MPI_INT
9 MPI_UNSIGNED
```

```
1 MPI_LONG
2 MPI_UNSIGNED_LONG
3 MPI_FLOAT
4 MPI_DOUBLE
5 MPI_LONG_DOUBLE
6 MPI_LONG_LONG_INT
7 MPI_UNSIGNED_LONG_LONG
8 MPI_LONG_LONG
```

MPI Status Value

```
1  typedef struct MPI_Status {
2    int MPI_SOURCE;
3    int MPI_TAG;
4    int MPI_ERROR;
5  };
```

code/hello-mpi.c

```
42  int
43  main (int argc, char **argv)
44  {
45    int num_procs;
46    int rank;
47
48    MPI_Init(&argc, &argv);
49    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
50    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
51
52    printf("%d: hello (p=%d)\n", rank, num_procs);
53    round_robin(rank, num_procs);
54    printf("%d: goodbye\n", rank);
55
56    MPI_Finalize();
57  }
```

code/hello-mpi.c

```
7   void
8   round_robin(int rank, int procs)
9   {
10    long int rand_mine, rand_prev;
11    int rank_next = (rank + 1) % procs;
12    int rank_prev = rank == 0 ? procs - 1 : rank - 1;
13    MPI_Status status;
14
15    srandom(time(NULL) + rank);
16    rand_mine = random() / (RAND_MAX / 100);
17    printf("%d: random is %ld\n", rank, rand_mine);
```

code/hello-mpi.c

```
19    if (rank % 2 == 0) {
20      printf("%d: sending %ld to %d\n",
21             rank, rand_mine, rank_next);
22      MPI_Send((void *)&rand_mine, 1, MPI_LONG, rank_next,
23               1, MPI_COMM_WORLD);
24          printf("%d: sent\n", rank);
25      MPI_Recv((void *)&rand_prev, 1, MPI_LONG, rank_prev,
26               1, MPI_COMM_WORLD, &status);
27          printf("%d: received\n", rank);
28    } else {
29      MPI_Recv((void *)&rand_prev, 1, MPI_LONG, rank_prev,
30               1, MPI_COMM_WORLD, &status);
31      printf("%d: sending %ld to %d\n",
32             rank, rand_mine, rank_next);
33      MPI_Send((void *)&rand_mine, 1, MPI_LONG, rank_next,
34               1, MPI_COMM_WORLD);
35    }
```

code/hello-mpi.c

```
37    printf("%d: I had %ld, %d had %ld\n",
38           rank, rand_mine,
39           rank_prev, rand_prev);
40  }
```

Output (as run)

```
1: hello (p=4)
1: random is 29
2: hello (p=4)
2: random is 65
2: sending 65 to 3
3: hello (p=4)
3: random is 51
3: sending 51 to 0
3: I had 51, 2 had 65
3: goodbye
```

```
0: hello (p=4)
0: random is 93
0: sending 93 to 1
0: I had 93, 3 had 51
0: goodbye
1: sending 29 to 2
1: I had 29, 0 had 93
1: goodbye
2: I had 65, 1 had 29
2: goodbye
```

Output (*stable* sorted)

```
sort --numeric-sort --stable output.txt
```

```
0: hello (p=4)
0: random is 93
0: sending 93 to 1
0: I had 93, 3 had 51
0: goodbye
1: hello (p=4)
1: random is 29
1: sending 29 to 2
1: I had 29, 0 had 93
1: goodbye
2: hello (p=4)
2: random is 65
2: sending 65 to 3
2: I had 65, 1 had 29
2: goodbye
3: hello (p=4)
3: random is 51
3: sending 51 to 0
3: I had 51, 2 had 65
3: goodbye
```

Overview

Simultaneous Send and Receive

```
1  int MPI_Sendrecv(void *sendbuf, int sendcount,
2                   MPI_Datatype senddatatype,
3                   int dest, int sendtag,
4
5                   void *recvbuf, int recvcount,
6                   MPI_Datatype recvdatatype,
7                   int source, int recvtag,
8
9                   MPI_Comm comm,
10                  MPI_Status *status);
```

code/round-robin-sr.c

```
19  MPI_Sendrecv((void *)&rand_mine,
20                1, MPI_LONG, rank_next, 1,
21
22                (void *)&rand_prev,
23                1, MPI_LONG, rank_prev, 1,
24
25                MPI_COMM_WORLD,
26                &status);
27
28  printf("%d: I had %ld, %d had %ld\n",
29         rank, rand_mine,
30         rank_prev, rand_prev);
```

Barrier and Broadcast

```
1  /* Return after every process calls */
2  int MPI_Barrier(MPI_Comm comm);
3
4  /* One-to-all broadcast */
5  int MPI_Bcast(void *buf, int count,
6                MPI_Datatype datatype,
7                int source,
8                MPI_Comm comm);
```

Broadcast

code/broad-barrier.c

```
25    long int random_value;
26    int broadcaster_rank = procs - 1;
27
28    if (rank == broadcaster_rank) {
29      srandom(time(NULL) + rank);
30      random_value = random() / (RAND_MAX / 10);
31      printf("%d: broadcasting %ld\n", rank, random_value);
32    }
33
34    MPI_Bcast((void *)&random_value,
35              1, MPI_LONG,
36              broadcaster_rank,
37              MPI_COMM_WORLD);
38
39    if (rank != broadcaster_rank) {
40      printf("%d: received %ld\n", rank, random_value);
41    }
```

Barrier

code/broad-barrier.c

```
43  int nap_time = random_value + (2 * rank);
44  printf("%d @ %02d: sleeping %ds\n",
45         rank, get_timer(), nap_time);
46  sleep(nap_time);
47
48  printf("%d @ %02d: enter b-a-r-r-i-e-r\n",
49         rank, get_timer());
50  MPI_Barrier(MPI_COMM_WORLD);
51  printf("%d @ %02d: leave barrier\n",
52         rank, get_timer());
```

Simple Elapsed Timer

code/broad-barrier.c

```
8   time_t start_time;
9
10  void
11  start_timer(void) {
12    time(&start_time);
13  }
14
15  int
16  get_timer(void) {
17    time_t now;
18    time(&now);
19    return now - start_time;
20  }
```

Computation proceeds concurrently with send and receive

1. Non-blocking send
   - Start send operation
   - Return *before* data copied out of buffer
2. Non-blocking receive
   - Start receive operation
   - Return *before* data received and copied in to buffer

Non-Blocking Send and Receive

```
1  /* Non-blocking send */
2  int MPI_Isend(void *buf, int count,
3                MPI_Datatype datatype,
4                int dest, int tag,
5                MPI_Comm comm,
6                MPI_Request *request);
7
8  /* Non-blocking receive */
9  int MPI_Irecv(void *buf, int count,
10               MPI_Datatype datatype,
11               int source, int tag,
12               MPI_Comm comm,
13               MPI_Request *request);
```

Make sure operation completes

1. Non-blocking send
   - Want to overwrite buffer
   - Must wait until data sent
2. Non-blocking receive
   - Want to use received data
   - Must wait until data received

Await Non-Blocking Completion

```
1  /* Test whether request complete */
2  int MPI_Test(MPI_Request *request,
3               int *flag,
4               MPI_Status *status);
5
6  /* Wait for request to complete */
7  int MPI_Wait(MPI_Request *request,
8               MPI_Status *status);
```

code/round-robin-non-block.c

```
20    printf("%d: sending %ld to %d\n",
21           rank, rand_mine, rank_next);
22    MPI_Isend((void *)&rand_mine, 1, MPI_LONG,
23              rank_next, 1, MPI_COMM_WORLD,
24              &send_request);
25    MPI_Recv((void *)&rand_prev, 1, MPI_LONG,
26             rank_prev, 1, MPI_COMM_WORLD,
27             &status);
28
29    printf("%d: I had %ld, %d had %ld\n",
30           rank, rand_mine,
31           rank_prev, rand_prev);
```

code/round-robin-non-block.c

```
33    int wait_count = 0;
34    int flag = 0;
35    do {
36      wait_count++;
37      MPI_Test(&send_request, &flag, &status);
38    } while(!flag);
39    printf("%d: MPI_Test calls: %d\n", rank, wait_count);
```

MPI Topologies

- Linear (default)
- Cartesian
- Graph

Cartesian Topology

```
1  /* Create Cartesian topology */
2  int MPI_Cart_create(MPI_Comm comm_old,
3                      int ndims, int *dims,
4                      int *periods, int reorder,
5                      MPI_Comm *comm_cart);
```

- `comm_old`—existing communicator
- `ndims`–number of dimensions
- `dims`–size of each dimension
- `periods`–does each dimension "wrap around"?
- `reorder`–renumber `ranks`?
- `comm_cart`—new communicator

Coordinates and Ranks

```
1  /* Coordinates to rank */
2  int MPI_Cart_rank(MPI_Comm comm_cart,
3                    int *coords,
4                    int *rank);
5
6  /* Rank to coordinates */
7  int MPI_Cart_coords(MPI_Comm comm_cart,
8                      int rank, int maxdims,
9                      int *coords);
```

Shift Along a Dimension

```
1  int MPI_Cart_shift(MPI_Comm comm_cart,
2                     int dir, int s_step,
3                     int *rank_source,
4                     int *rank_dest)
```

- `comm_cart`—communicator
- `dir`—direction (dimension of topology)
- `s_step`—step size
- `rank_source`—source rank
- `rank_dest`—destination rank