CSE 891 - Section 1: Parallel Computing - Fundamentals and Applications

Fall 2014 - Lectures 7: Programming using the Message Passing Paradigm

H. Metin Aktulga
hma@cse.msu.edu
517-355-1646

Lecture 6- Summary

- Programming Using the Message Passing Paradigm
  - Principles of Message-Passing Programming
  - Building Blocks: Send & Recv operations
    - Semantics of Send & Recv
    - Blocking, Buffered, Non-blocking Sends
- MPI: the Message Passing Interface
    - History of MPI and MPI Implementations
    - The minimal set of MPI routines
    - MPI_Send & MPI_Recv
    - MPI Datatypes
    - Avoiding deadlocks

Asynchronous Send & Recvs

- To overlap communication with computation, MPI provides functions to perform non-blocking send & receives which return before the operations are completed.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
                   int dest, int tag, MPI_Comm comm,
                   MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
                   int source, int tag, MPI_Comm comm,
                   MPI_Request *request)
```

- **From OpenMPI manual:** A nonblocking send call indicates that the system may start copying data out of the send buffer. The sender should not access any part of the send buffer after a nonblocking send operation is called, until the send completes.

Asynchronous Send & Recvs (cont.)

- **Important:** All asynchronous operations are given a request handle that has to be acted on later in one of the following ways:
  - block and wait for the operation to finish with MPI_Wait(…) or MPI_Waitany(…)

    ```
    int MPI_Wait(MPI_Request *request, MPI_Status *status)
    ```

  - test for completion with MPI_Test(…) or MPI_Testany(…) until the test turns out positive

    ```
    int MPI_Test(MPI_Request *request, int *flag,
                 MPI_Status *status)
    ```

  - If neither is done, the asynchronous calls may not be fully completed - messages not sent, memory leaks, etc.

Overlapping Communication and Computation

- Recommended usage is to pair a blocking and non-blocking call
- MPI_Isend on p0 and MPI_Recv on p1:

p0 posts the send and goes back to doing useful computations, p1 does useful computations (followed by MPI_Wait or MPI_Test) and starts waiting on recv only when it can not proceed further

- MPI_Send on p0 and MPI_Irecv on p1:

p1 posts the MPI_Irecv as soon as it knows the kind and (max) length of a message it will receive and continues to do useful computations (followed by MPI_Wait of MPI_Test). When p0 has a message ready to be sent, it immediately find the posted recv.

Overlapping Communication and Computation

- So to achieve the best overlapping, why not post all MPI_Irecv's and/or MPI_Isend's very early on and put MPI_Wait's at the very end?
  - First, we do not know well ahead of time how many and how long most messages will be
  - Also, having too many outstanding asynchronous send & recv's may require too much buffer space - and we may run out of memory!

Avoiding Deadlocks Revisited

- Non-blocking operations resolve most deadlocks. Consider:

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
```

- Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.

An Example

- **Problem:** Compute *n* values and find their sum.
- **Serial solution** is straight-forward

```
sum = 0;
for (i = 0; i< n; ++i) {
  x = Compute_next_value(…);
  sum += x;
}
```

How to do this in parallel?

- We have p cores, where $p \ll n$
- How can we compute the sum in parallel?
- It turns out that there are several ways with different communication requirements
- Let us take a look at them in more detail

Algorithm 1

```
if (my_rank == 0){
 sum = 0;
 for (i = 0; i< n/p; ++i){
   x = Compute_next_value(…);
   sum += x;
 }
 for (i = n/p; i < n; ++i){
   recv(x, src, tag, comm);
   sum += x;
 }
}
else {
 for (i = 0; i< n/p; ++i){
   x = Compute_next_value(…);
   send(x, 0, tag, comm);
 }
}
```

- Is this a good algorithm?
- How many messages are exchanged?
- How many bytes are communicated?
- Is the computation load balanced?
- Can we improve it?

Algorithm 2

```
if (my_rank == 0){
 my_sum = 0;
 for (i = 0; i< n/p; ++i){
  x = Compute_next_value(…);
  my_sum += x;
 }
 for (i = 1; i < p; ++i) {
  recv(other_sum, i,…);
  my_sum += other_sum;
 }
}
else {
 my_sum = 0;
 for (i = 0; i< n/p; ++i) {
  x = Compute_next_value(…);
  my_sum += x;
 }
 send(my_sum, 0,…);
}
```

- Let us make it load balanced
- Is the computation load balanced?
- Now how many messages are exchanged?
- How many bytes are communicated?
- Can we improve it?

Algorithm 2 (cont.)

my_sum after local n/p local summations

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

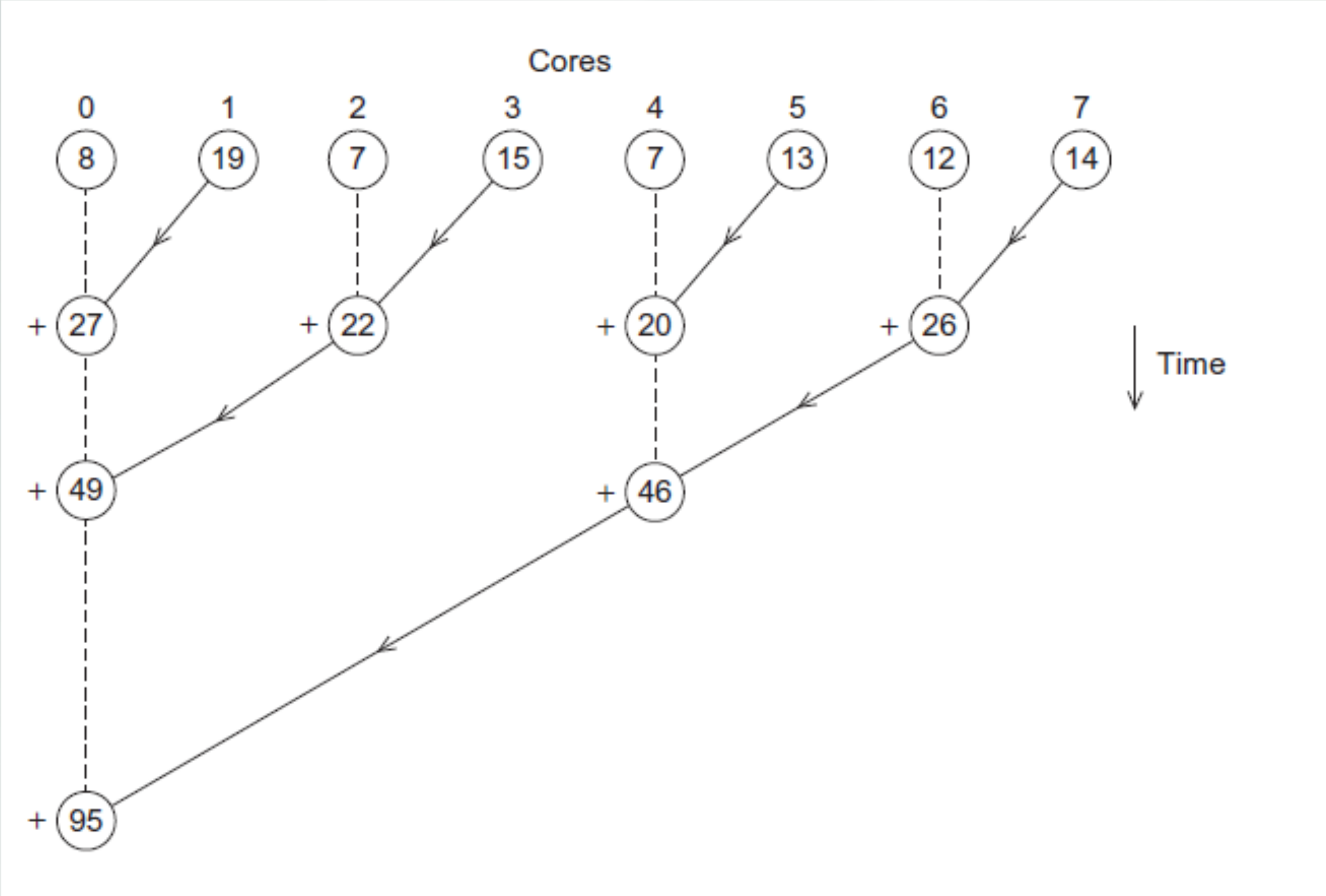
Global sum

8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

my_sum after comm & summations at process 0

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

How about a tree algorithm?



Algorithm 3

```
my_sum = 0;
for (i = 0; i < n/p; ++i){
 x = Compute_next_value(…);
 my_sum += x;
}

m = log(p);
for (i = 0; i < m; ++i) {
 determine my partner
 if (I am sender)
  send(my_sum, partner,…);
 else { // I am receiver
  recv(other_sum, partner,…);
  my_sum += other_sum;
}

// At this point, p0's my_sum
// is the global sum
```

- Let us make it load balanced
- Is the computation load balanced?
- Now how many messages are exchanged?
- How many bytes are communicated?

Algorithm 3 (cont.)

- Parallel sum example shows how send & recv can be used as building blocks for complex communication patterns
- The tree structure used is called a binomial tree - similar to, but not to be confused with a binary tree
- As we will see, commonly used complex operations are already implemented in MPI
- Parallel sum is called a reduction as implemented by MPI_Reduce

Collective Communication Operations

- MPI provides an extensive set of functions for performing common collective communication operations.
- Each of these operations is defined over a group corresponding to the communicator.
- All processors in a communicator must call these operations.
- Up until MPI-3, all collective calls used to be blocking. Now there exists their non-blocking counterparts, too!
  - At least they are in the specification
  - When will they be implemented? and how efficient will the implementations be???

Collective Communication Operations

- The barrier synchronization operation in MPI:

```
int MPI_Barrier(MPI_Comm comm)
```

- The one-to-all broadcast operation is:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
       int source, MPI_Comm comm)
```

- The all-to-one reduction operation is:

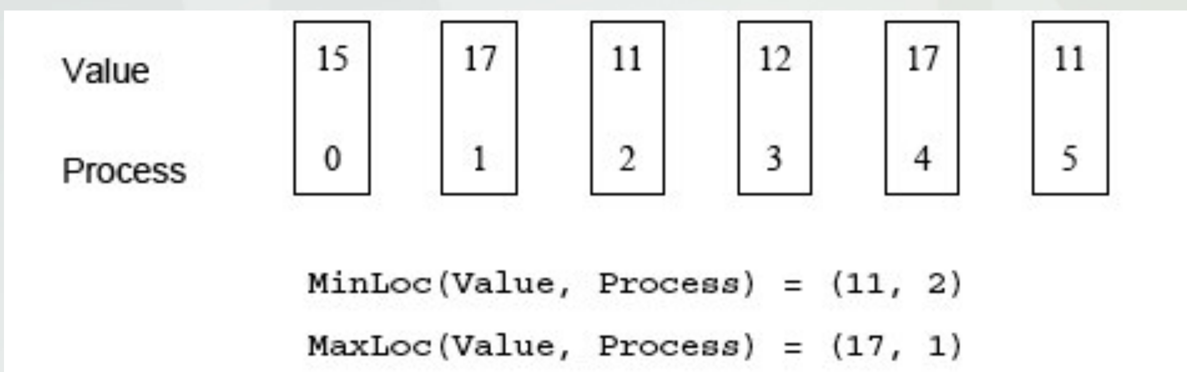
```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
       MPI_Datatype datatype, MPI_Op op, int target,
       MPI_Comm comm)
```

Pre-defined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

Collective Communication Operations

- The operation `MPI_MAXLOC` combines pairs of values (v_i, l_i) and returns the pair (v, l) such that v is the maximum among all v_i 's and l is the corresponding l_i (if there are more than one, it is the smallest among all these l_i ’s).
- `MPI_MINLOC` does the same, except for minimum value of v_i.



Collective Communication Operations

MPI datatypes for data-pairs used with the `MPI_MAXLOC` and `MPI_MINLOC` reduction operations

MPI Datatype	C Datatype
`MPI_2INT`	pair of ints
`MPI_SHORT_INT`	short and int
`MPI_LONG_INT`	long and int
`MPI_LONG_DOUBLE_INT`	long double and int
`MPI_FLOAT_INT`	float and int
`MPI_DOUBLE_INT`	double and int

Collective Communication Operations

- If the result of the reduction operation is needed by all processes, MPI provides:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
             int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm)
```

- To compute prefix-sums, MPI provides:

```
int MPI_Scan(void *sendbuf, void *recvbuf,
      int count, MPI_Datatype datatype, MPI_Op op,
      MPI_Comm comm)
```

Collective Communication Operations

- The gather operation is performed in MPI using:

```
int MPI_Gather(void *sendbuf, int sendcount,
               MPI_Datatype senddatatype, void *recvbuf,
               int recvcount, MPI_Datatype recvdatatype,
               int target, MPI_Comm comm)
```

- MPI_Allgather function gathers the data at all processes:

```
int MPI_Allgather(void *sendbuf, int sendcount,
               MPI_Datatype senddatatype, void *recvbuf,
               int recvcount, MPI_Datatype recvdatatype,
               MPI_Comm comm)
```

- The corresponding scatter operation is:

```
int MPI_Scatter(void *sendbuf, int sendcount,
               MPI_Datatype senddatatype, void *recvbuf,
               int recvcount, MPI_Datatype recvdatatype,
               int source, MPI_Comm comm)
```

Collective Communication Operations

- The all-to-all personalized communication operation is performed by:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, void *recvbuf,
        int recvcount, MPI_Datatype recvdatatype,
        MPI_Comm comm)
```

- Using this core set of collective operations, parallel programs can be greatly simplified.