PHY1610 - Distributed Parallel Programming with MPI - part 2

Ramses van Zon, Marcelo Ponce

April 1, 2021

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Section 1

MPI Point-to-point communication

MPI: Send Right, Receive Left

```
#include <iostream>
#include <string>
#include <mpi.h>
using namespace std;
int main(int argc, char **argv)
{
    int         rank, size, left, right, tag = 1;
    double      msgsent, msgrcvd;
    MPI_Status  rstatus;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    left = rank - 1;
    if (left < 0) left = MPI_PROC_NULL;
    right = rank + 1;
    if (right >= size) right = MPI_PROC_NULL;
    msgsent = rank*rank;
    msgrcvd = -999.;
    MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    cout << to_string(rank) + ": Sent " + to_string(msgsent)
          + " and got " + to_string(msgrcvd) + "\n";

    MPI_Finalize();
}
```

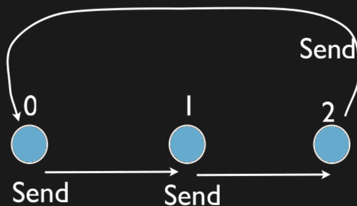
MPI: Send Right, Receive Left

```
$ make secondmessage
$ mpirun -n 3 ./secondmessage
2: Sent 4.000000 and got 1.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
$
```

```
$ mpirun -n 6 ./secondmessage
4: Sent 16.000000 and got 9.000000
5: Sent 25.000000 and got 16.000000
0: Sent 0.000000 and got -999.000000
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
```

MPI: Send Right, Receive Left with Periodic BCs

Periodic Boundary Conditions:



MPI: Send Right, Receive Left with Periodic BCs

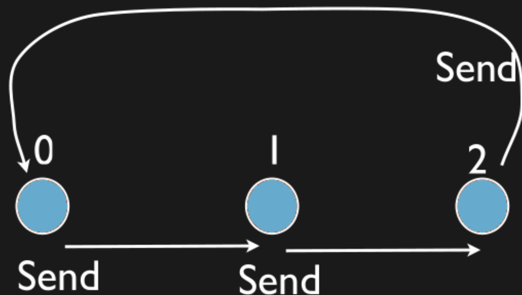
```
...
left = rank - 1;
if (left < 0) left = size-1; // Periodic BC
right = rank + 1;
if (right >= size) right =0; // Periodic BC
msgsent = rank*rank;
msgrcvd = -999.;
...
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Deadlock!

- A classic parallel bug.
- Occurs when a cycle of tasks are waiting for the others to finish.
- Whenever you see a closed cycle, you likely have (or risk) a deadlock.
- Here, all processes are waiting for the send to complete, but no one is receiving.

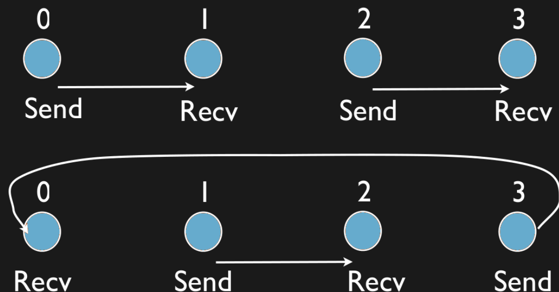
Sends and receives must be paired when sending



How do we fix the deadlock?

Without using new MPI routine, how do we fix the deadlock?

Even-odd solution



- First: evens send, odds receive
- Then: odds send, evens receive
- Will this work with an odd number of processes? How about 2? 1?

MPI: Send Right, Recv Left with Periodic BCs - fixed

```
    ...
    if ((rank % 2) == 0) {
        MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
        MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
    } else {
        MPI_Recv(&msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
        MPI_Ssend(&msgsent, 1, MPI_DOUBLE, right, tag, MPI_COMM_WORLD);
    }
    ...
```

```
$ make fourthmessage
$ mpirun -n 5 ./fourthmessage
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got 16.000000
```

MPI: Sendrecv

```
MPI_Sendrecv(sendptr, count, MPI_TYPE, destination, tag,
             recvptr, count, MPI_TYPE, source, tag, Communicator, MPI_Status)
```

- A blocking send and receive built together.
- Lets them happen simultaneously.
- Can automatically pair send/recvs.
- Why 2 sets of tags/types/counts?

Send Right, Receive Left with Periodic BCs - Sendrecv

Code

```
...
MPI_Sendrecv(&msgsent, 1, MPI_DOUBLE, right, tag,
             &msgrcvd, 1, MPI_DOUBLE, left, tag, MPI_COMM_WORLD, &rstatus);
...
```

Execution

```
$ make fifthmessage
$ mpirun -n 5 ./fifthmessage
1: Sent 1.000000 and got 0.000000
2: Sent 4.000000 and got 1.000000
3: Sent 9.000000 and got 4.000000
4: Sent 16.000000 and got 9.000000
0: Sent 0.000000 and got 16.000000
```

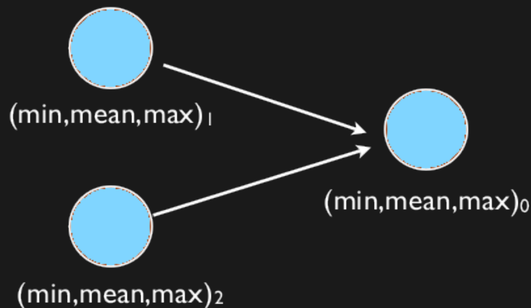
SciNet

Section 2

MPI Reductions

Reductions: Min, Mean, Max Example

- Calculate the min/mean/max of random numbers -1.0 . . . 1.0
- Should trend to -1/0/+1 for a large N.
- How to MPI it?
- Partial results on each node, collect all to node 0.



Reductions: Min, Mean, Max Example

```
#include <mpi.h>
#include <iostream>
#include <algorithm>
#include <random>
#include <rarray>
using namespace std;
int main(int argc, char **argv)
{
  int rank;
  int size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  const long nx = 200000000;
  const long nxper=(nx+size-1)/size;
  const long nxown=(rank<size-1)?nxper
                  :(nx-nxper*(size-1));
  rvector<double> dat(nxown);
  uniform_real_distribution<double>
     uniform(-1.0,1.0);
  minstd_rand engine(14);
  engine.discard(nxper*rank);
  for (long i=0;i<nxown;i++)
      dat[i] = uniform(engine);
```

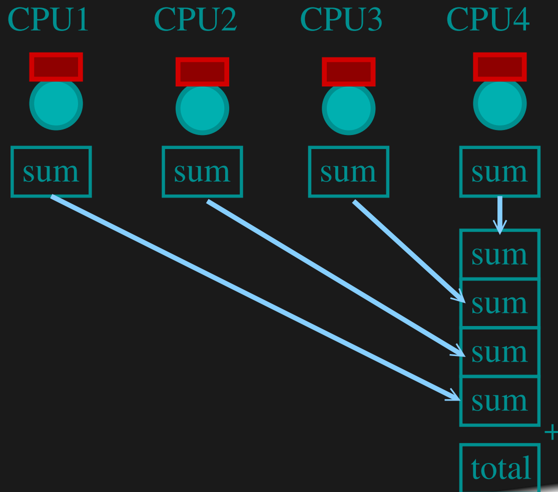
```
  const long MIN=0, SUM=1, MAX=2;
  rvector<double> mmm(3);
  mmm = 1e+19, 0, -1e+19;
  for (long i=0;i<nxown;i++) {
      mmm[MIN] = min(dat[i], mmm[MIN]);
      mmm[MAX] = max(dat[i], mmm[MAX]);
      mmm[SUM] += dat[i];
  }
  const long tag = 13;
  const long collectorrank = 0;
  if (rank != collectorrank)
    MPI_Ssend(mmm.data(), 3,MPI_DOUBLE,
              collectorrank, tag,
              MPI_COMM_WORLD);
  else {
    rvector<double> recvmmm(3);
    for (long i = 1; i < size; i++) {
       MPI_Recv(recvmmm.data(), 3,
                MPI_DOUBLE,
                MPI_ANY_SOURCE, tag,
                MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
```

```
       mmm[MIN] = min(recvmmm[MIN],
                      mmm[MIN]);
       mmm[MAX] = max(recvmmm[MAX],
                      mmm[MAX]);
       mmm[SUM] += recvmmm[SUM];
    }
    cout << "Global Min/mean/max "
         << mmm[MIN] << " "
         << mmm[SUM]/nx <<" "
         << mmm[MAX] <<endl;
  }
  MPI_Finalize();
}
```

Efficiency?

- Requires (P-1) messages
- 2(P-1) if everyone then needs to get the answer.

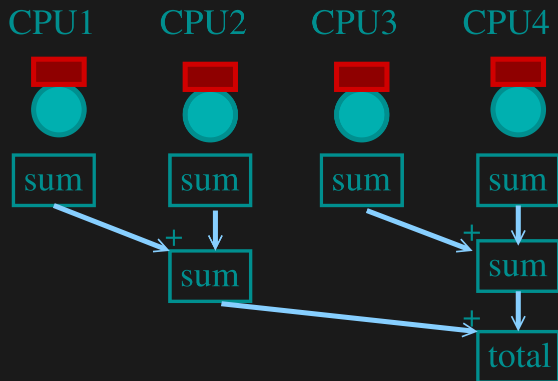
$$T_{comm} = PC_{comm}$$



Better Summing

- Pairs of processors; send partial sums
- Max messages received $\log_2(P)$
- Can repeat to send total back.

$$T_{comm} = 2\log_2(P)C_{comm}$$



Reduction: Works for a variety of operations (+,*,min,max)

MPI Collectives

```
MPI_Allreduce(sendptr, rcvptr, count, MPI_TYPE, MPI_Op, Communicator);

MPI_Reduce(sendbuf, recvbuf, count, MPI_TYPE, MPI_Op, root, Communicator);
```

- sendptr/rcvptr: pointers to buffers
- count: number of elements in ptrs
- MPI_TYPE: one of MPI_DOUBLE, MPI_FLOAT, MPI_INT, MPI_CHAR, etc.
- MPI_Op: one of MPI_SUM, MPI_PROD, MPI_MIN, MPI_MAX.
- Communicator: MPI_COMM_WORLD or user created.
- All variant send result back to all processes; non-All sends to process root.

SciNet

Reductions: Min, Mean, Max with MPI Collectives

```
rvector<double> globalmmm(3);
MPI_Allreduce(&mmm[MIN], &globalmmm[MIN], 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
MPI_Allreduce(&mmm[MAX], &globalmmm[MAX], 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
MPI_Allreduce(&mmm[SUM], &globalmmm[SUM], 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
if (rank==0)
   cout << "Global Min/mean/max "
        << mmm[MIN] << " "
        << mmm[SUM]/nx << " "
        << mmm[MAX] << endl;
```

SciNet

More Collective Operati...

Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

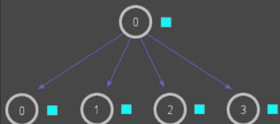
More Collective Operati

Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

Broadcast



More Collective Operati

Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

Broadcast

Scatter

More Collective Operat

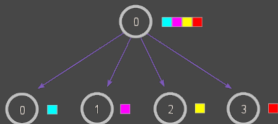
Collective

- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

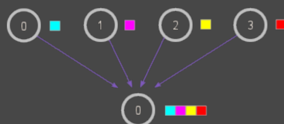
Other MPI Collectives

Broadcast

Scatter



Gather



More Collective Operati

Collective

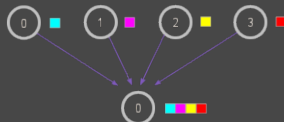
- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

Broadcast

Scatter

Gather



- File I/O
- Barriers (avoid!)
- All-to-all . . .

Section 3

MPI Domain decomposition

Solving the diffusion equation with MPI

Consider a diffusion equation with an explicit **finite-difference**, **time-marching** method.

Imagine the problem is too large to fit in the memory of one node, so we need to do **domain decomposition**, and use **MPI**.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Discretizing Derivatives

- Partial Differential Equations like the diffusion equation

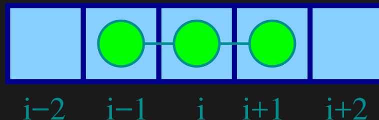
$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

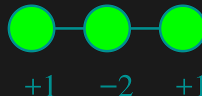
are usually numerically solved by finite differencing the discretized values.

- Implicitly or explicitly involves interpolating data and taking the derivative of the interpolant.

- Larger 'stencils' $\rightarrow$ More accuracy.

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}$$





Domain decomposition

- A very common approach to parallelizing on distributed memory computers.
- Subdivide the domain into contiguous subdomains.
- Give each subdomain to a different MPI process.
- No process contains the full data!
- Maintains locality.
- Need mostly local data, ie., only data at the boundary of each subdomain will need to be sent between processes.