PHY1610 - Distributed Parallel Programming with MPI - part 2

Ramses van Zon

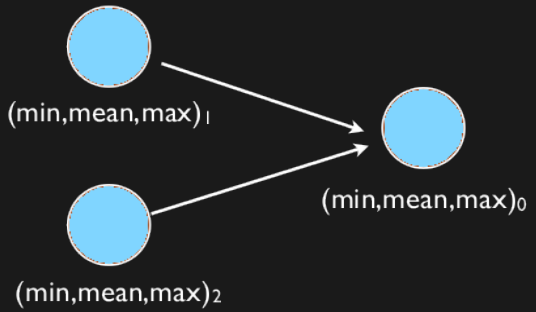
March 31, 2022

1

MPI Reductions

Reductions: Min, Mean, Max Example

- Calculate the min/mean/max of random numbers -1.0 ... 1.0
- Should tend 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);
```

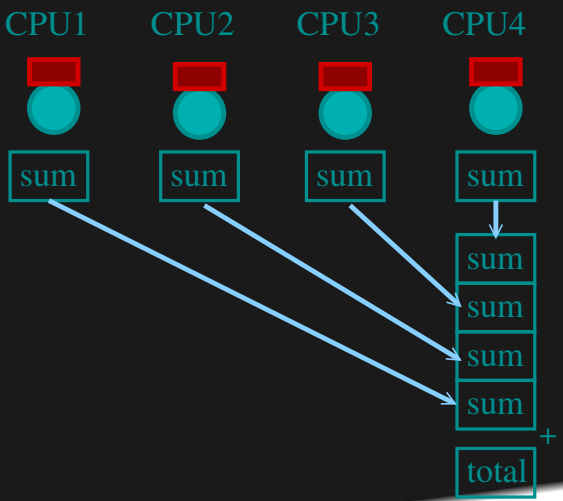
```
  const long MIN=0, SUM=1, MAX=2;
  rvector<double> mmm(3);
  mmm = 1e+19, 0, -1e+19;
  for (long i=0;i<nxown;i++) {
      mmm[MIN] = min(dat[i], mmm[MIN]);
      mmm[MAX] = max(dat[i], mmm[MAX]);
      mmm[SUM] += dat[i];
  }
  const long tag = 13;
  const long collectorrank = 0;
  if (rank != collectorrank)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

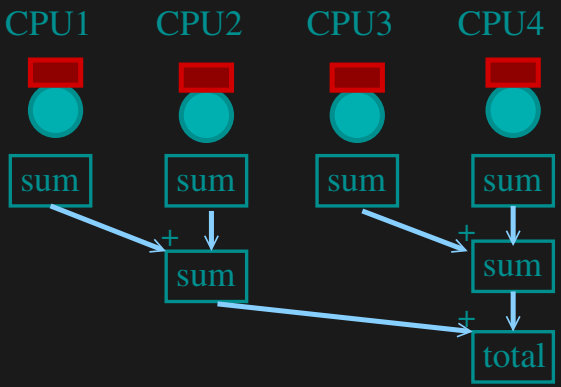
$$T_{comm} = PC_{comm}$$



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`.

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;
```

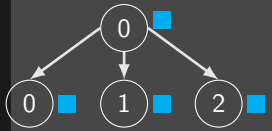
More Collective Operations

Collective

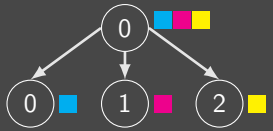
- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's "under the hood".

Other MPI Collectives

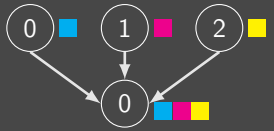
Broadcast



Scatter

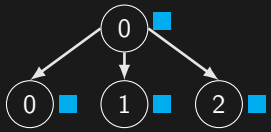


Gather



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

Broadcast

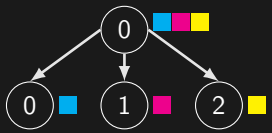



```
int MPI_Bcast(void *buffer, int count,
              MPI_Datatype datatype,
              int root, MPI_Comm comm);
```

```
// bcast.cpp
#include <mpi.h>
#include <iostream>
int main(int argc,char* argv[])
{
    int rank, size;
    double value = -999;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank==0)
        value = 3.14159;
    MPI_Bcast(&value, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    std::cout << "rank " + std::to_string(rank)
        + " got value = " + std::to_string(value) + "\n";
    MPI_Finalize();
}
```

```
$ module load gcc/9 openmpi
$ mpicxx -O2 -g -std=c++17 bcastex.cpp -o bcastex
$ mpirun -n 4 ./bcastex
rank 3 got value = 3.141590
rank 0 got value = 3.141590
rank 2 got value = 3.141590
rank 1 got value = 3.141590
```

Scatter

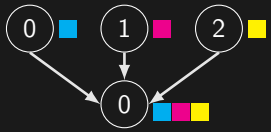



```
int MPI_Scatter(const void *sendbuf, int sendcount,
                MPI_Datatype sendtype, void *recvbuf,
                int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
```

```
// scatterex.cpp
#include <mpi.h>
#include <iostream>
#include <rarray>
int main(int argc,char* argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    rvector<double> allvalues;
    if (rank==0)
        allvalues = linspace(0.0,(double)size,size,false);
    double value;
    MPI_Scatter(allvalues, 1, MPI_DOUBLE,
                &value, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    std::cout << "rank " + std::to_string(rank)
        + " got value = " + std::to_string(value) + "\n";
    MPI_Finalize();
}
```

```
$ module load gcc/9 openmpi rarray
$ mpicxx -O2 -g -std=c++17 scatterex.cpp -o scatterex
$ mpirun -n 4 ./scatterex
rank 1 got value = 1.000000
rank 2 got value = 2.000000
rank 3 got value = 3.000000
rank 0 got value = 0.000000
```

Gather




```
int MPI_Gather(const void *sendbuf, int sendcount,
               MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm);
```

```
// gatherex.cpp
#include <mpi.h>
#include <iostream>
#include <rarray>
int main(int argc,char* argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    rvector<double> allvalues(size);
    double value = rank;
    MPI_Gather(&value, 1, MPI_DOUBLE,
               allvalues, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if (rank == 0)
        std::cout << "rank 0 got " << allvalues <<  "\n";
    MPI_Finalize();
}
```

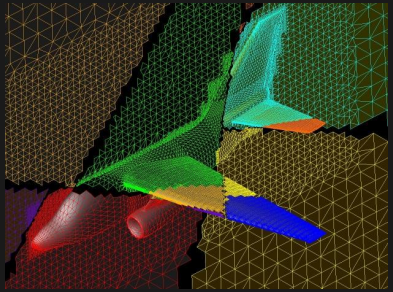
```
$ module load gcc/9 openmpi rarray
$ mpicxx -O2 -g -std=c++17 scatterex.cpp -o scatterex
$ mpirun -n 4 ./scatterex
rank 0 got {0,1,2,3}
```

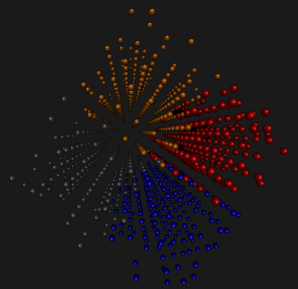
2

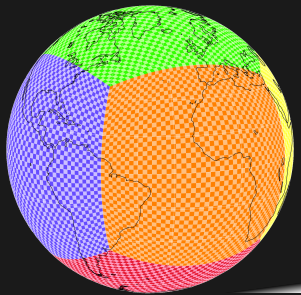
MPI Domain decomposition

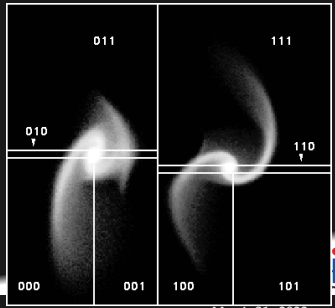
Domain decomposition

- 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.









Solving a PDE with MPI

Consider e.g. 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**.

Discretizing Derivatives

- Partial Differential Equations like the diffusion equation

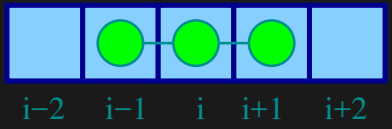
$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

  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.

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



Discretize in higher dimensions

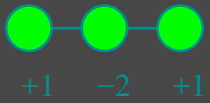
Spatial grid separation: Δx. Time step Δt.

Grid indices: i, j. Time step index: (n)

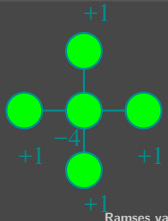
1D

$$\left.\frac{\partial T}{\partial t}\right|_i \approx \frac{T_i^{(n)} - T_i^{(n-1)}}{\Delta t}$$

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_i \approx \frac{T_{i-1}^{(n)} - 2T_i^{(n)} + T_{i+1}^{(n)}}{\Delta x^2}$$



2D



$$\left.\frac{\partial T}{\partial t}\right|_{i,j} \approx \frac{T_{i,j}^{(n)} - T_{i,j}^{(n-1)}}{\Delta t}$$

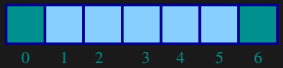
$$\left.\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)\right|_{i,j} \approx \frac{T_{i-1,j}^{(n)} + T_{i,j-1}^{(n)} - 4T_{i,j}^{(n)} + T_{i+1,j}^{(n)} + T_{i,j+1}^{(n)}}{\Delta x^2}$$

Stencils and Boundaries

How do you deal with boundaries?

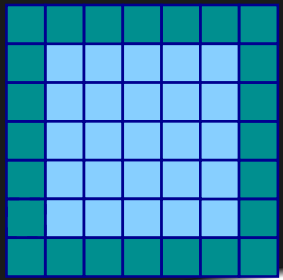
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution: Guard cells
  - Pad domain with these guard cells so that stencil works even for the first point in domain.
  - Fill guard cells with values such that the required boundary conditions are met.

1D



- Number of guard cells $n_g = 1$
- Loop from $i = n_g \ldots N - 2n_g$.

2D



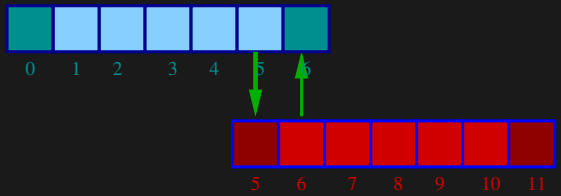
What does this have to do with MPI?

Guard cells will come in very very handy when parallelizing aplications whose domains are too large to fit in memory or who need more cores than are available on one node.

For such applications, one often uses Domain decomposition as a strategy to MPI parallelize the computation.

Guard cell exchange

- In the domain decomposition, the stencils will jut out into a neighbouring subdomain.
- Much like the boundary condition.
- One uses guard cells for domain decomposition too.
- If we managed to fill the guard cell with values from neighbouring domains, we can treat each coupled subdomain as an isolated domain with changing boundary conditions.



- Could use even/odd trick, or sendrecv.

Example: 1D diffusion with MPI (snippet)

Before MPI

```
a = 0.25*dt/pow(dx,2);
guardleft = 0;
guardright = n+1;
for (int t=0;t<maxt;t++) {
 T[guardleft] = 0.0;
 T[guardright] = 0.0;
 for (int i=1; i<=n; i++)
   newT[i] = T[i] + a*(T[i+1]+T[i-1]-2*T[i]);
 for (int i=1; i<=n; i++)
   T[i] = newT[i];
}
```

After MPI

```
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;
localn = n/size;
a = 0.25*dt/pow(dx,2);
guardleft = 0;
guardright = localn+1;
for (int t=0;t<maxt;t++) {
 MPI_Sendrecv(&T[1],         1,MPI_DOUBLE,left, 11,
              &T[guardright],1,MPI_DOUBLE,right,11,
              MPI_COMM_WORLD,MPI_STATUS_IGNORE);
 MPI_Sendrecv(&T[nlocal],    1,MPI_DOUBLE,right,11,
              &T[guardleft], 1,MPI_DOUBLE,left, 11,
              MPI_COMM_WORLD,MPI_STATUS_IGNORE);
 if (rank==0) T[guardleft] = 0.0;
 if (rank==size-1) T[guardright] = 0.0;
 for (int i=1; i<=localn; i++)
   newT[i] = T[i] + a*(T[i+1]+T[i-1]-2*T[i]);
 for (int i=1; i<=n; i++)
   T[i] = newT[i];
}
MPI_Finalize();
```