Distributed Parallel Programming with MPI - part 2

Ramses van Zon

PHY1610 Winter 2023

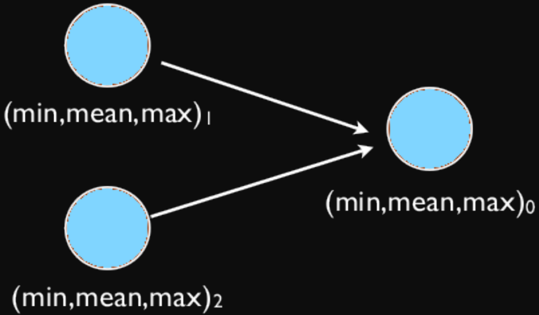
Quick MPI Recap so far

- In MPI, `P` processes are started and running all the time:
  `mpirun -n P application appargumentss`
- Single Program/Multiple Data model: each process runs the same application code (with same arguments)
- All processes are part of a communicator:
  `MPI_Init(&argc, &argv)` to be called in `int main(int argc, char** argv)`
- All processes should finalize their role in the MPI applications:
  `MPI_Finalize()` to be called at the end of `int main(int argc, char** argv)`
- `rank` and `size` are the only distinguishing factor:
  `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`
  `MPI_Comm_size(MPI_COMM_WORLD, &size)`
- Each process must determine what its role/data is based on `rank` and `size`.
- Communication of data among processes requires explicit sending and receiving of messages.
  `MPI_Ssend(sendptr,count,type,tag,MPI_COMM_WORLD)`
  `MPI_Recv(recvptr,maxcount,type,tag,MPI_COMM_WORLD,status)`
- MPI is a C and Fortran Library, so pointers abound. It knows C types (MPI_DOUBLE, MPI_INT, . . .), and can send linear arrays.

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

```
// Computes the min,mean&max of random nu
#include <mpi.h>
#include <iostream>
#include <algorithm>
#include <random>
#include <rarray>
using namespace std;
int main(int argc, char **argv)
{
  const long nx = 200'000'000;
  // find this process place
  int rank;
  int size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  // determine its subrange of data
  const long nxper=(nx+size-1)/size;
  const long nxstart=nxper*rank;
  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);
```

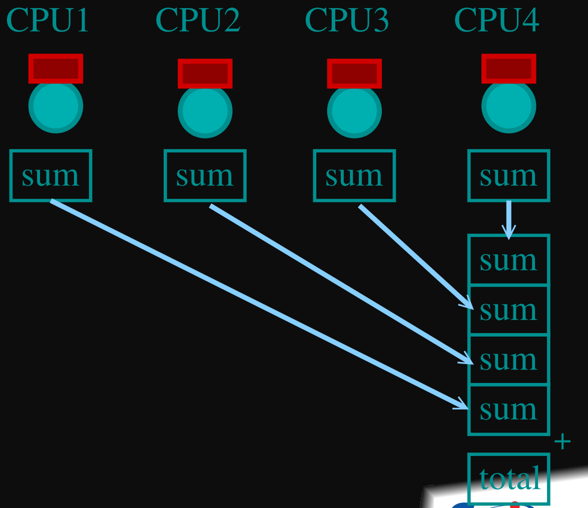
```
  // each process skip ahead to start
  engine.discard(nxstart);
  // compute local data
  for (long i=0;i<nxown;i++)
      dat[i] = uniform(engine);
  const long MIN=0, SUM=1, MAX=2;
  rvector<double> mmm(3);
  mmm = 1e+19, 0, -1e+19;
  for (long i=0;i<nxown;i++) {
      mmm[MIN] = min(dat[i], mmm[MIN]);
      mmm[MAX] = max(dat[i], mmm[MAX]);
      mmm[SUM] += dat[i];
  }
  // send results to a collecting rank
  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];
    }
    // output
    std::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.

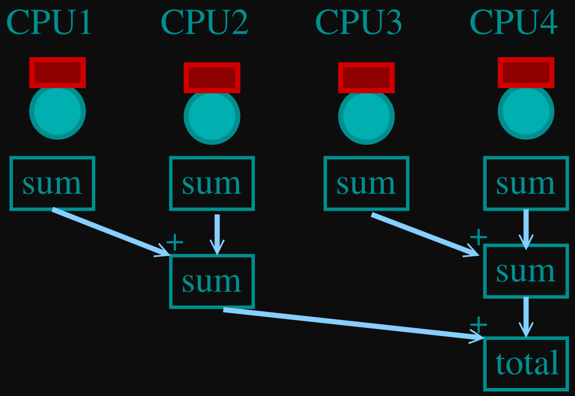
$$T_{comm} = P C_{comm}$$



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.
- The “`All`” variant sends 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;
```

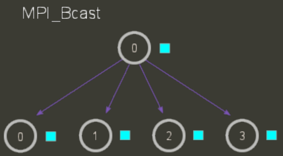
More Collective Operati

Collective

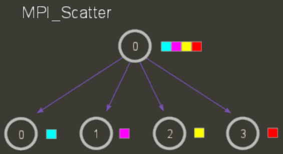
- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's 'under the hood'.

Other MPI Collectives

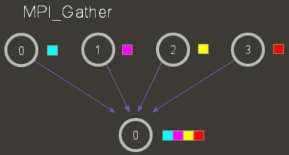
Broadcast



Scatter



Gather



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

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**.

Discretizing Derivatives

- Partial Differential Equations like the diffusion equation

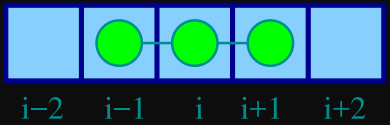
$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

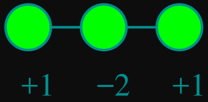
are usually numerically solved by finite differencing the discretized values.

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

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

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





Diffusion equation in higher dimensions

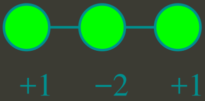
Spatial grid separation: Δx. Time step Δt.

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

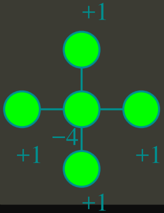
1D

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

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



2D



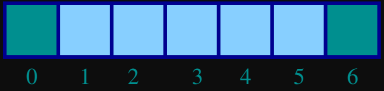
$$\left.\frac{\partial T}{\partial t}\right|_{i,j} \approx \frac{T_{i,j}^{(n)} - T_{i,j}^{(n-1)}}{\Delta t}$$

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

Stencils and Boundaries

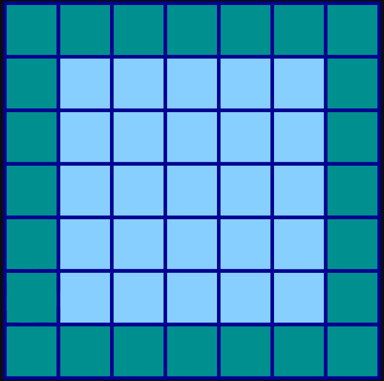
- How do you deal with boundaries?
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution: ***Guard cells***:
  - Pad domain with these guard celss so that stencil works even for the first point in domain.
  - Fill guard cells with values such that the required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

1D



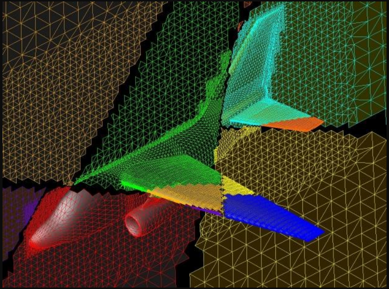
- Number of guard cells $n_g = 1$
- Loop from $i = n_g \ldots N - 2n_g$.

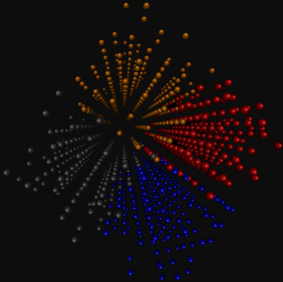
2D

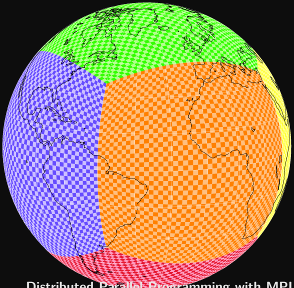


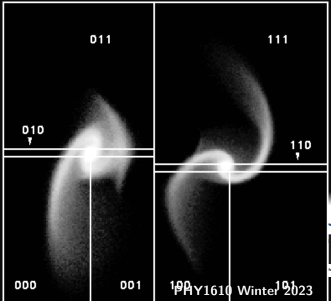
Domain decomposition

- 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.



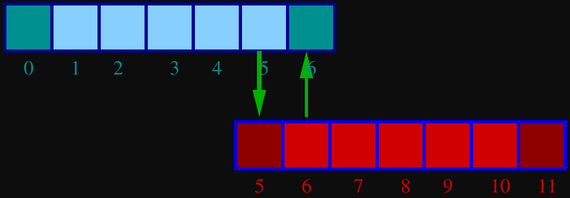






Guard cell exchange

- 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.

1D diffusion with MPI

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];
}
```

Note:

- the for-loop over `i` could also have been a call to dgemv for a submatrix.
- the for-loop over `i` could also easily be parallelized with OpenMP

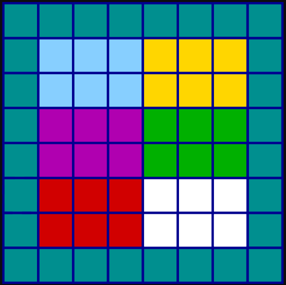
  (⇒ hybrid MPI-OpenMP code).

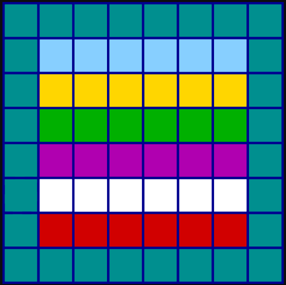
After MPI

```
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();
```

2D diffusion with MPI

How to divide the work in 2d?



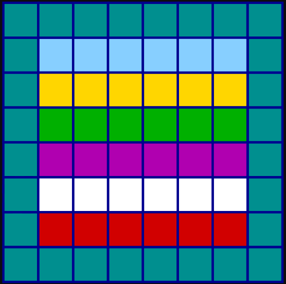


- Less communication (18 edges).
- Harder to program, non-contiguous data to send, left, right, up and down.

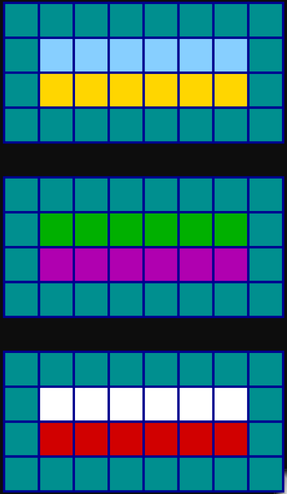
- Easier to code, similar to 1d, but with contiguous guard cells to send up and down.
- More communication (30 edges).

Let's look at the easiest domain decomposition.

Serial:



Parallel ***($P = 3$)*:**



Communication pattern:

- Copy upper stripe to upper neighbour bottom guard cell.
- Copy lower stripe to lower neighbout top guard cell.
- Contiguous cells: can use `count` in `MPI_Sendrecv`.
- Similar to 1d diffusion.

Hybrid MPI+OpenMP

Hybrid MPI+OpenMP

You can mix MPI and OpenMP.

This can be beneficial: pure MPI requires more communications and more memory overhead.

Coding

As far as coding is involved, that's easy: use MPI calls and OpenMP directives.

Usually, the MPI part is the trickiest: do that first.

Running

As far as running hybrid code, that can be tricky too. You want to avoid running one mpi process per core, and then overloading that core with multiple threads.

The scheduler can help in this respect. E.g. with SLURM, with 16-core nodes, you can say

```
#SBATCH --nodes=3
#SBATCH --ntaskspernode=2
#SBATCH --cpuspertask=8
module load gcc openmpi
export OMP_NUM_THREADS=8
mpirun ./hybridcode  # can use srun instead of mpirun too.
....
```

to get 6 mpi processes spread over 3 nodes, each running 8 threads.