Distributed Parallel Programming with MPI - part 2

Ramses van Zon

PHY1610 Winter 2026

Communication patterns in MPI

Communication patterns in MPI

1. No communication between processes
   `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Finalize`
2. Point-to-point
   `MPI_Ssend`, `MPI_Recv`, `MPI_Sendrecv`
3. Broadcast
   Send same data from one rank to all others
4. Reduction
   Combine results from all ranks (*e.g.* sum)
5. Scatter
   Send different data from one rank to all others
6. Gather
   Collect data from one rank to all others
7. All-to-all
   Everyone sends something to everyone

3. MPI Broadcast

Broadcast

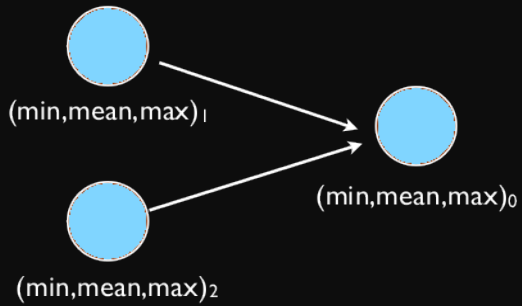
This involves one process sending data to all others.

```cpp
#include <mpi.h>
#include <string>
#include <print>
#include <iostream>
int main() {
    int rank, length, iorank = 0;
    std::string name;
    MPI_Init(nullptr, nullptr);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == iorank) {
       std::print("What is your name? ");
       std::cin  >> name;
       length = name.size();
    }
    MPI_Bcast(&length, 1, MPI_INT,
              iorank, MPI_COMM_WORLD);
    name.resize(length);
    MPI_Bcast(&name[0], length, MPI_CHAR,
              iorank, MPI_COMM_WORLD);
    std::println("Rank {} knows {}", rank, name);
    MPI_Finalize();
}
```

```
$ mpicxx -std=c++23 -o bcastex bcastex.cpp
$ mpirun -n 3 ./bcastex
What is your name? Ramses
Rank 0 knows Ramses
Rank 1 knows Ramses
Rank 2 knows Ramses
```

4. MPI Reductions

Reductions: Min, Mean, Max Example



- Say we want to calculate the min, mean, and max of N random numbers between -1.0 and 1.0
- Should trend to -1/0/+1 for a large N.
- How to parallize this with MPI?
- Partial results computed by each process
- collect all to process 0.

Reductions: Min, Mean, Max Example (1/2)

```
// Computes the min,mean&max of random numbers
#include <mpi.h>
#include <print>
#include <algorithm>
#include <random>
#include <rarray>
int main()
{
  long long N = 200'000'000;
  // find this process place
  int rank, size;
  MPI_Init(nullptr, nullptr);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  // determine its subrange of data
  long long Nper=(N+size-1)/size;
  long long Nstart=Nper*rank;
  long long Nown=(rank<size-1)?Nper
                 :(N-Nper*(size-1));
  rvector<float> dat(Nown);
  std::uniform_real_distribution<float>
     uniform(-1.0,1.0);
```

```
  std::minstd_rand engine(14);
  // each process skip ahead to start
  engine.discard(Nstart);
  // compute local data
  for (long long i=0;i<Nown;i++)
      dat[i] = uniform(engine);
  int MIN=0, SUM=1, MAX=2;
  rvector<float> mmm(3);
  mmm = 1e+19, 0, -1e+19;
  for (long long i=0;i<Nown;i++) {
      mmm[MIN] = std::min(dat[i], mmm[MIN]);
      mmm[MAX] = std::max(dat[i], mmm[MAX]);
      mmm[SUM] += dat[i];
  }
  // send results to a collecting rank
  int collectorrank = 0;
  if (rank != collectorrank)
    MPI_Ssend(mmm.data(), 3,MPI_FLOAT,
              collectorrank, 749,
              MPI_COMM_WORLD);
  else {
    rvector<float> recvmmm(3);
```

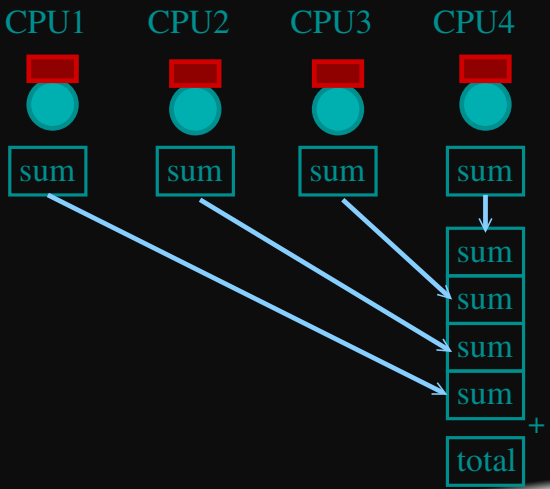
Reductions: Min, Mean, Max Example (1/2)

```
    for (long long i=1; i<size; i++) {
       MPI_Recv(recvmmm.data(), 3,
                MPI_FLOAT,
                MPI_ANY_SOURCE, 749,
                MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
       mmm[MIN] = std::min(recvmmm[MIN],
                           mmm[MIN]);
       mmm[MAX] = std::max(recvmmm[MAX],
                           mmm[MAX]);
       mmm[SUM] += recvmmm[SUM];
    }
    // output
    std::println("Global Min/mean/max {} {} {}",
                 mmm[MIN],
                 mmm[SUM]/N,
                 mmm[MAX]);
  }
  MPI_Finalize();
}
```

Efficiency?

- This requires $(P-1)$ messages.
- or $2(P-1)$ if everyone then needs to get the answer.

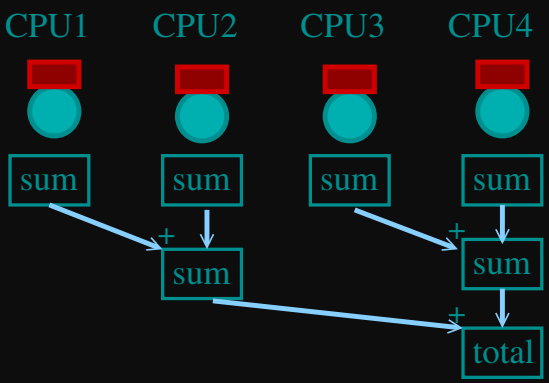
$$T_{comm} = PC_{comm}$$



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_Reduce(sendbuf, recvbuf, count, MPI_TYPE, MPI_Op, root, Communicator);

MPI_Allreduce(sendptr, rcvptr, count, MPI_TYPE, MPI_Op, 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<float> globalmmm(3);
MPI_Reduce(&mmm[MIN],&globalmmm[MIN],1,MPI_FLOAT, MPI_MIN, collectorrank, MPI_COMM_WORLD);
MPI_Reduce(&mmm[MAX],&globalmmm[MAX],1,MPI_FLOAT, MPI_MAX, collectorrank, MPI_COMM_WORLD);
MPI_Reduce(&mmm[SUM],&globalmmm[SUM],1,MPI_FLOAT, MPI_SUM, collectorrank, MPI_COMM_WORLD);
if (rank==collectorrank)
  std::println("Global Min/mean/max {} {} {}",
               mmm[MIN],
               mmm[SUM]/nx,
               mmm[MAX]);
```

Reduction example with MPI

```cpp
#include <mpi.h>
#include <print>
#include <algorithm>
#include <random>
#include <rarray>
int main()
{
  long long N = 200'000'000;
  // find this process place
  int rank, size;
  MPI_Init(nullptr, nullptr);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  // determine its subrange of data
  long long Nper=(N+size-1)/size;
  long long Nstart=Nper*rank;
  long long Nown=(rank<size-1)?Nper
                 :(N-Nper*(size-1));
  rvector<float> dat(Nown);
  std::uniform_real_distribution<float>
     uniform(-1.0,1.0);
  std::minstd_rand engine(14);
  // each process skip ahead to start
  engine.discard(Nstart);
  // compute local data
  for (long long i=0;i<Nown;i++)
      dat[i] = uniform(engine);
  int MIN=0, SUM=1, MAX=2;
  rvector<float> mmm(3);
  mmm = 1e+19, 0, -1e+19;
  for (long long i=0;i<Nown;i++) {
      mmm[MIN] = std::min(dat[i], mmm[MIN]);
      mmm[MAX] = std::max(dat[i], mmm[MAX]);
      mmm[SUM] += dat[i];
  }
  // send results to a collecting rank
  int collectorrank = 0;
  rvector<float> globalmmm(3);
  MPI_Reduce(&mmm[MIN],&globalmmm[MIN],1,MPI_FLOAT,M
  MPI_Reduce(&mmm[MAX],&globalmmm[MAX],1,MPI_FLOAT,M
  MPI_Reduce(&mmm[SUM],&globalmmm[SUM],1,MPI_FLOAT,M
  if (rank==0)
      std::println("Global Min/mean/max {} {} {}",
                   mmm[MIN],
                   mmm[SUM]/N,
                   mmm[MAX]);
  MPI_Finalize();
}
```

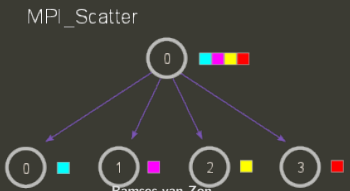
More Collective Operations

Collective

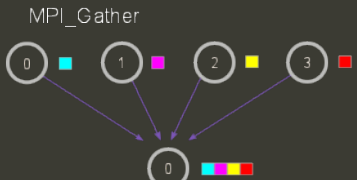
- Reductions are an example of a ***collective*** operation.
- As opposed to the pairwise messages we've seen before
- All processes in the communicator must participate.
- Cannot proceed until all have participated.
- Don't necessarity know what's "under the hood".

Other MPI Collectives

5. Scatter



6. Gather

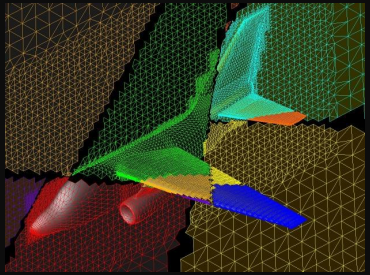


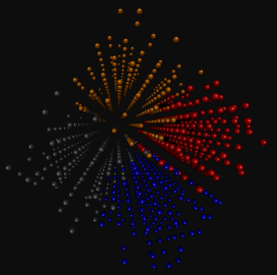
7. Even more:
   - All-to-all ...
   - File I/O
   - Barriers (avoid!)

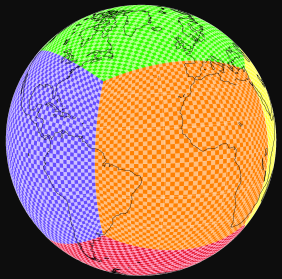
MPI Domain decomposition

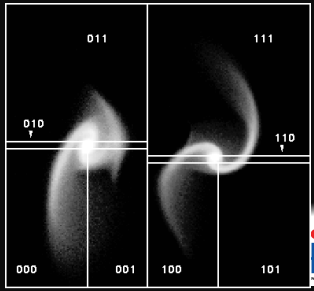
Domain decomposition

- 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.









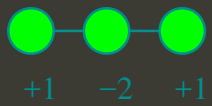
Diffusion equation

Consider a diffusion equation (lecture 13!) with an explicit **finite-difference**, **time-marching** method.

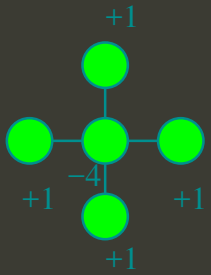
Imagine the problem is too large to fit in the memory of one node, so we need to do **domain decomposition**, and use **MPI**.

Stencils

We can think of the algorithm as applying *stencils* to each point:



1D

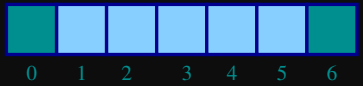


2D

Boundaries

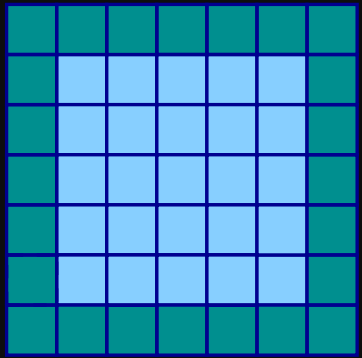
- The stencil juts out, you need info on cells beyond those you're updating.
- Before, we just changed the code for those points.
- Another common solution is to use ***Guard cells***:
  - Pad domain with these guard celss so that stencil works even for the first point in domain.
  - Fill guard cells with values such that the required boundary conditions are met.

1D



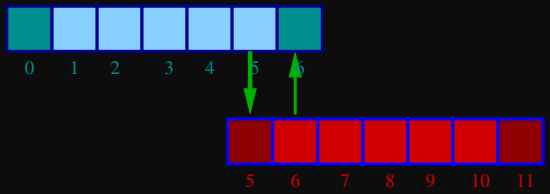
- Number of guard cells $n_g = 1$
- Loop from $i = n_g..N - 2n_g$.

2D



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]);
 std::swap(T, newT);
}
```

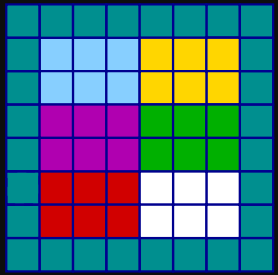
Note: the for-loop over `i` could also have been a call to dgemv for a submatrix.

After MPI

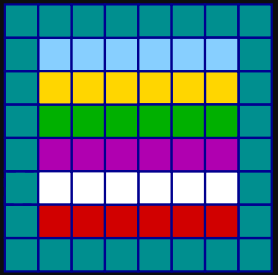
```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
left =rank-1; if(left<0)left=MPI_PROC_NULL;
right=rank+1; if(right>=size)right=MPI_PROC_NULL;
localn = n/size;
a = 0.25*dt/pow(dx,2);
guardleft = 0;
guardright = localn+1;
for (int t=0;t<maxt;t++) {
 MPI_Sendrecv(&T[1],         1,MPI_FLOAT,left, 7,
              &T[guardright],1,MPI_FLOAT,right,7,
              MPI_COMM_WORLD,MPI_STATUS_IGNORE);
 MPI_Sendrecv(&T[nlocal],    1,MPI_FLOAT,right,7,
              &T[guardleft], 1,MPI_FLOAT,left, 7,
              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]);
 std::swap(T, newT);
}
MPI_Finalize();
```

2D diffusion with MPI

How to divide the work in 2d?



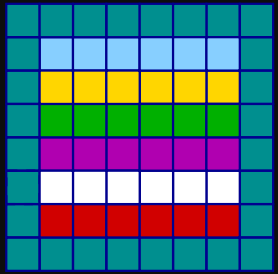
- Less communication (18 edges).
- Harder to program, non-contiguous data to send, left, right, up and down.



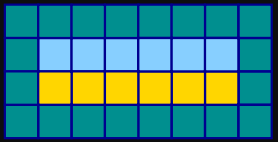
- Easier to code, similar to 1d, but with contiguous guard cells to send up and down.
- More communication (30 edges).

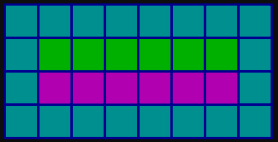
Let's look at the easiest domain decomposition.

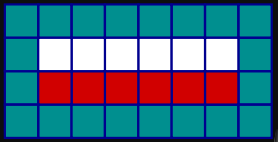
Serial:



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.

MPI + OpenMP

MPI is scalable and portable.

Why would we want make things more complicated with + `OpenMP`?

1. Communication cost:
   The more processes, the more communication:

   Want less processes, but still use all cores:

   - Reduce the number of processes
   - Increasing the thread count per process

   That requires MPI + Multicore
   Usually: MPI + OpenMP

2. Memory overhead
   The more processes, the more communication.

3. Unused computing hardware:
   You may have compute accelerators that you weren't using.

   Usually, GPUs.

   This needs specialize programming:
   MPI + CUDA
   or
   MPI + OpenMP GPU offloading

Hybrid MPI+OpenMP: Coding

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

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.

But: have to initialize MPI differently, instead of `MPI_Init`, use `MPI_Init_thread`:

```
int required = SOMETHING;
int provided;

MPI_Init_thread(&argc,&argv,required,&provided);

if (provided < required) exit(1);
```

Here, SOMETHING can be:

- `MPI_THREAD_SINGLE`
  Only one thread will execute.
- `MPI_THREAD_FUNNELED`
  Only the thread that called MPI_Init_thread will make MPI calls.
- `MPI_THREAD_SERIALIZED`
  Only one thread will make MPI library calls at one time.
- `MPI_THREAD_MULTIPLE`
  Multiple threads may call MPI at once with no restrictions.

Hybrid MPI+OpenMP: Running

You must be specific about the numbers to avoid overloading cores.

In scheduled jobs

The scheduler can help in this respect. E.g. with SLURM, with 40-core nodes, you can say

```
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=20
#SBATCH --time=1:00:00
module load gcc openmpi
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
mpirun ./hybridcode  # can use srun instead of mpirun too.
```

This gets 6 mpi processes spread over 3 nodes, each running 20 threads.

On login nodes or your own machine

E.g. to get 4 mpi processes on the node each running 3 threads, you'd do

```
$ module load gcc openmpi
$ export OMP_NUM_THREADS=3
$ mpirun -n 4 ./hybridcode
```

Hybrid: Which mix of MPI/OpenMP is best?

There are many, many aspects which factor into this decision.

While it's true that many applications get their best performance from running in hybrid mode, the precise balance of threads and processes is not always the same.

Hybrid: Memory consideration

MPI takes more memory than OpenMP, because:

- Every process has its own memory.
- Even if data is distributed, each process has to have the executable loaded.
- and there will be additional ghost cells
- MPI will use internal buffers.

This would suggest using one MPI process per node, and threads to use the cores.

But:

- The memory of the MPI processes tends to be closer to the cores.
- No cache coherency slowdowns in MPI, unlike in OpenMP.

Hybrid: CPU considerations

- Both processes and threads will be assigned to different cores on the CPU by the OS.
- That is, as long as there are enough cores.
- If you underutilize cores, the OS may move your process.
  But it does not move the memory with it!
- If your MPI computation keeps the cores busy, i.e., it's pure MPI, the OS won't see a reason to move them.
- In OpenMP, there are always serial portions, and the chance of "thread migration" is real.
- But in MPI, processes may be waiting for communication or syncronization.

Nodes may have several CPUs in different sockets, e.g, the Teach cluster has 2 sockets with each an 20-core CPU. These have their own caches and different parts of main memory that are closer to each socket.

Binding

When running in hybrid mode, consider **pinning** a.k.a. **binding** processes and threads to specific cores.

OpenMP

There's a few environment variables that control the binding of OpenMP Threads:

- `OMP_PROC_BIND=true` tells openmp to perform binding of threads.
- `OMP_PLACES=X` where X can be `cores`, `threads` or `sockets`, or a list of core numbers.

MPI

Binding is done with options to mpirun, but these differ per MPI implementation. For openmpi:

- `--map-by X`, where X is hwthread, core, L1cache, L2cache, L3cache, socket, numa, or, board.
- `--report-bindings` option to allows check the bindings

https://docs.open-mpi.org/en/v5.0.x/man-openmpi/man1/mpirun.1.html

Slurm

Some MPI implementations are integrated enough with the scheduler that it will "do the right thing" by default, if you set `ntasks_per_node`. Still, be explicit if you can.

Hybrid: Communication considerations

Network

Often, nodes have one connection to the network.

If you have multiple MPI processes on a node, they share this connection.

Less processes may mean less communication, and less communication buffers, which can help.

Course Conclusions

Conclusions

Thanks for sticking to the end of this course.

- We have covered a lot.
- From software development in C++ all the way to parallel programming.
- All of which from the angle of scientific computing.
- And with many exercise to practice the skills.

But there is always more to learn in this field.

Check out https://explora.alliancecan.ca for many more training opportunities in this field.

Course evaluation

If you haven't yet, take some minutes to complete the **course evaluation**!