PHY1610 - Distributed Parallel Programming with MPI - part 3

Ramses van Zon

April 2, 2024

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**.

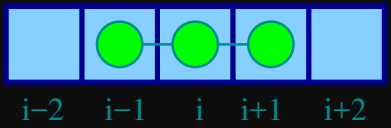
Discretizing Derivatives

- Partial Differential Equations like the diffusion equation

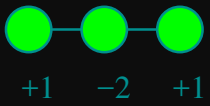
$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

are usually numerically solved by finite differencing the discretized values.

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



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



- Larger "stencils" → More accuracy.

Diffusion equation in higher dimensions

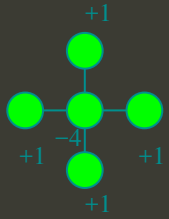
Spatial grid separation: Δx. Time step Δt.
Grid indices: i, j. Time step index: (n)

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



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

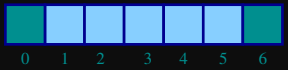
$$\left.\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)\right|_{i,j} \approx \frac{T_{i-1,j}^{(n)} + T_{i,j-1}^{(n)} - 4T_{i,j}^{(n)} + T_{i+1,j}^{(n)} + T_{i,j+1}^{(n)}}{\Delta x^2}$$

Stencils and Boundaries

How do you deal with boundaries?

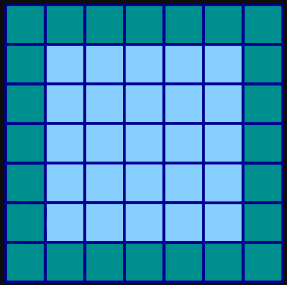
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution: Guard cells
  - Pad domain with these guard cells so that stencil works even for the first point in domain.
  - Fill guard cells with values such that the required boundary conditions are met.

1D



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

2D



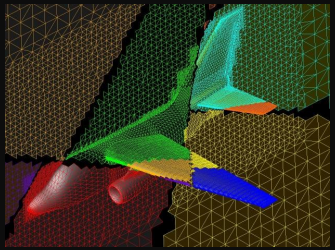
What does this have to do with MPI?

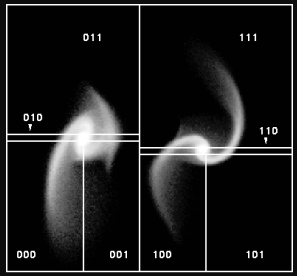
Guard cells will come in very very handy when parallelizing aplications whose domains are too large to fit in memory or who need more cores than are available on one node.

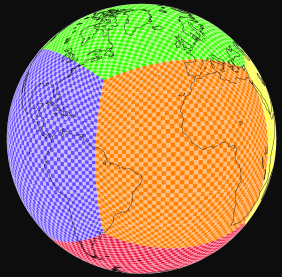
For such applications, one often uses Domain decomposition as a strategy to MPI parallelize the computation.

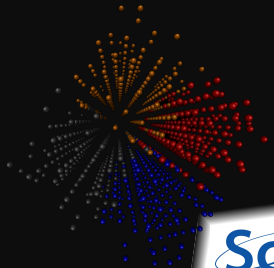
Domain decomposition

- 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.



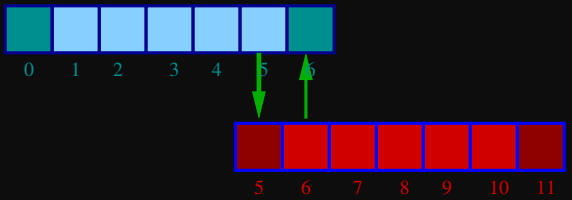






Guard cell exchange

- 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 `dgbmv` for a submatrix.
- the for-loop over `i` could also easily be parallelized with OpenMP

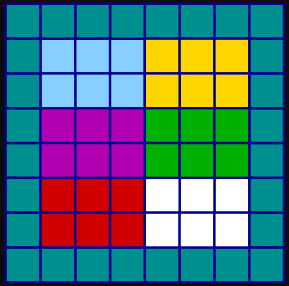
  ($\Rightarrow$ hybrid MPI-OpenMP code).

After MPI

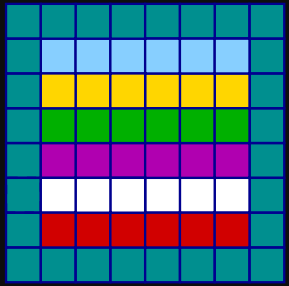
```
MPI_Init(nullptr,nullptr);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
left  = rank-1; if(left<0)left=MPI_PROC_NULL;
right = rank+1; if(right>=size)right=MPI_PROC_NULL;
localn = n/size;
a = 0.25*dt/pow(dx,2);
guardleft = 0;
guardright = localn+1;
for (int t=0;t<maxt;t++) {
 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];
}
```

2D diffusion with MPI

How to divide the work in 2d?



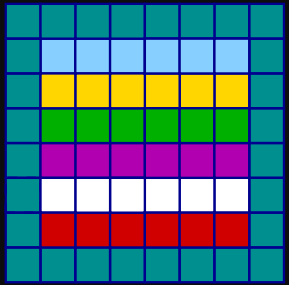
- Less communication (18 edges).
- Harder to program, non-contiguous data to send, left, right, up and down.



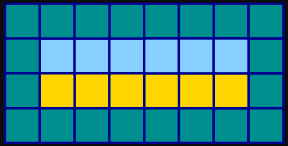
- Easier to code, similar to 1d, but with contiguous guard cells to send up and down.
- More communication (30 edges).

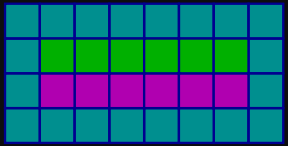
Let's look at the easiest domain decomposition.

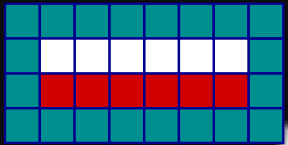
Serial:



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: 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.

One has 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 16-core nodes, you can say

```
#SBATCH --nodes=3
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=8

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 8 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

- 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 process per node, and threads for the rest.

But:

- The memory of the MPI processes tends to be closer to the cores.
- No cache coherency slowdowns, like in OpenMP.
- Less chance of race conditions.

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.
- In OpenMP, there are always serial portions, 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 8-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.

Hybrid: I/O considerations

If the file system is a network file system, it has the same limitations.

If the file system is a single disk in the nodes, having several processes write at once can slow things down.

But network file systems are often parallel, when using multple nodes.

MPI can help here.

You can use MPI to do IO in parallel

- I/O is often the slowest part of a computing system.
- Large HPC installations have parallel file systems to help
- These have many disks on the back-end, enabling parallel reading and writing
- As with many parallel technique, parallelization is not automatic

Solutions:

- Could use a separate file for each process.
  - But now output depends on #processes.
  - Can lead to directory locking.
- MPI-IO: Sub-library that enables binary parallel file I/O to single files from all processes.
- HDF5 and NetCDF also allow parallel I/O if those libraries were built to support it.

MPI-IO is similar to ordinary files

```
MPI_Offset offset = (msgsize*rank);

MPI_File  file;
MPI_Status stat;

MPI_File_open(MPI_COMM_WORLD, "helloworld.txt",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &file);

MPI_File_seek(file, offset, MPI_SEEK_SET);
MPI_File_write(file, msg, msgsize, MPI_CHAR, &stat);
MPI_File_close(&file);
```

You have to control the data layout and what process gets to write where in the file!

One usually creates a so-called 'File view' to help with that.

Another Example

```
MPI_Offset offset = (msgsize*rank);

MPI_File file;
MPI_Status stat;

MPI_File_open(MPI_COMM_WORLD, "helloworld.txt",
                  MPI_MODE_CREATE | MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &file);

// Collective Coordinated Write
MPI_File_write_at_all(file, offset, msg, msgsize, MPI_CHAR, &stat);

MPI_File_close(&file);
```

Here, MPI-IO is similar to MPI collectives.