

# PHY1610 - Distributed Parallel Programming with MPI - part 3

Ramses van Zon, Marcelo Ponce

April 6, 2021

## Section 1

# 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

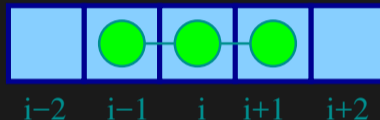
- Partial Differential Equations like the diffusion equation

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

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' → 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:  $\Delta x$ .      Time step  $\Delta t$ .

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

1D

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

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



# Diffusion equation in higher dimensions

Spatial grid separation:  $\Delta x$ . Time step  $\Delta t$ .

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

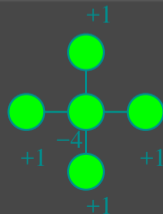
## 1D

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

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



## 2D



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

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

# Stencils and Boundaries

How do you deal with boundaries?

- The stencil juts out, you need info on cells beyond those you're updating.

# 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.



# 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  $i = n_g \dots N - 2n_g$ .

# Stencils and Boundaries

How do you deal with boundaries?

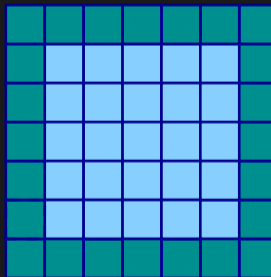
- The stencil juts out, you need info on cells beyond those you're updating.
- Common solution: **Guard cells**
  - ▶ Pad domain with these guard cells so that stencil works even for the first point in domain.
  - ▶ Fill guard cells with values such that the required boundary conditions are met.

1D



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

2D



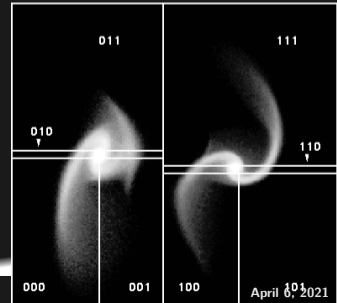
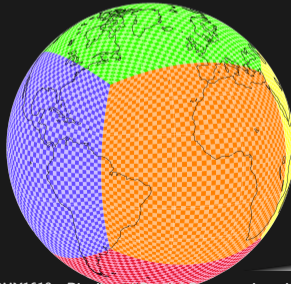
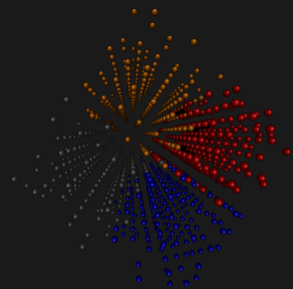
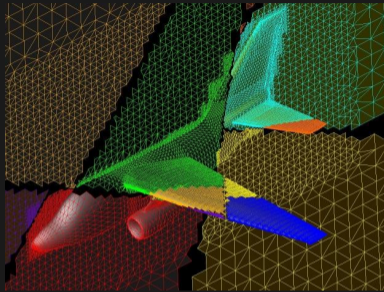
## What does this have to do with MPI?

Guard cells will come in very very handy when parallelizing applications 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.

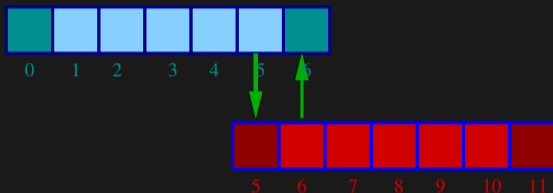
# Domain decomposition

- 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];
}
```

# 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];
}
```

## 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();
```

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

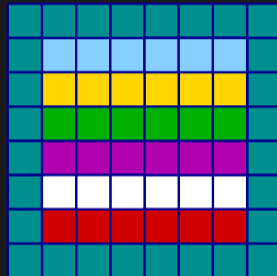
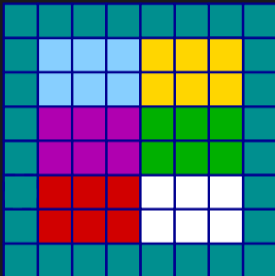
## After MPI

```
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();
```



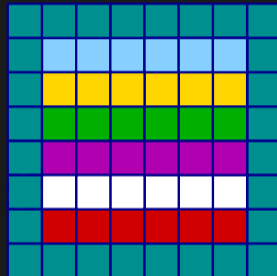
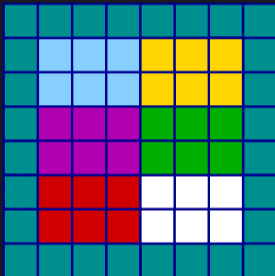
# 2D diffusion with MPI

How to divide the work in 2d?



## 2D diffusion with MPI

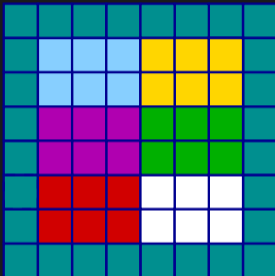
How to divide the work in 2d?



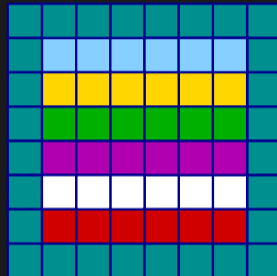
- Less communication (18 edges).
- Harder to program, non-contiguous data to send, left, right, up and down.

## 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.

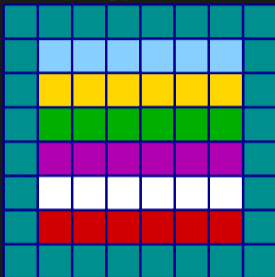


- Easier to code, similar to 1d, but with contiguous guard cells to send up and down.
- More communication (30 edges).

Let's look at the easiest domain decomposition.

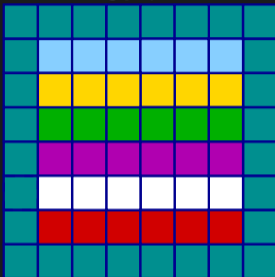
Let's look at the easiest domain decomposition.

*Serial:*

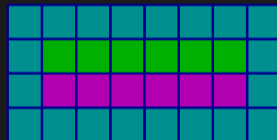
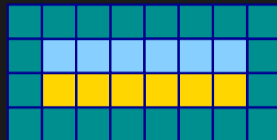


# Let's look at the easiest domain decomposition.

*Serial:*

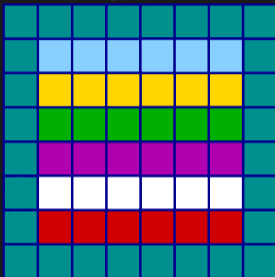


*Parallel ( $P = 3$ ):*

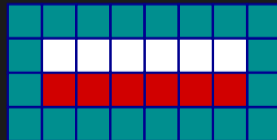
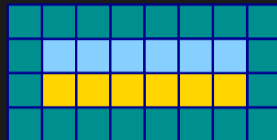


## Let's look at the easiest domain decomposition.

*Serial:*



*Parallel ( $P = 3$ ):*



### *Communication pattern:*

- Copy upper stripe to upper neighbour bottom guard cell.
- Copy lower stripe to lower neighbour top guard cell.
- Contiguous cells: can use count in MPI\_Sendrecv.
- Similar to 1d diffusion.

## Section 2

# 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.

# 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`  
If the process is multithreaded, only the thread that called `MPI_Init_thread` will make MPI calls.
- `MPI_THREAD_SERIALIZED`  
If the process is multithreaded, only one thread will make MPI library calls at one time.
- `MPI_THREAD_MULTIPLE`  
If the process is multithreaded, 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.
....
```

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

## 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.
....
```

to get 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
```

## Section 3

# MPI-IO

# 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.

## MPI-IO is similar to MPI\_Collectives

```
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);
```



## Useful MPI-IO functions:

- MPI\_File\_open
- MPI\_File\_set\_view
- MPI\_File\_write
- MPI\_File\_write\_at
- MPI\_File\_write\_all
- MPI\_File\_write\_at\_all
- MPI\_File\_read
- MPI\_File\_read\_at
- MPI\_File\_read\_all
- MPI\_File\_read\_at\_all
- MPI\_File\_close

Note: after module load gcc openmpi you can access the man pages of these functions.

E.g.

```
$ module load gcc openmpi  
$ man MPI_File_open
```