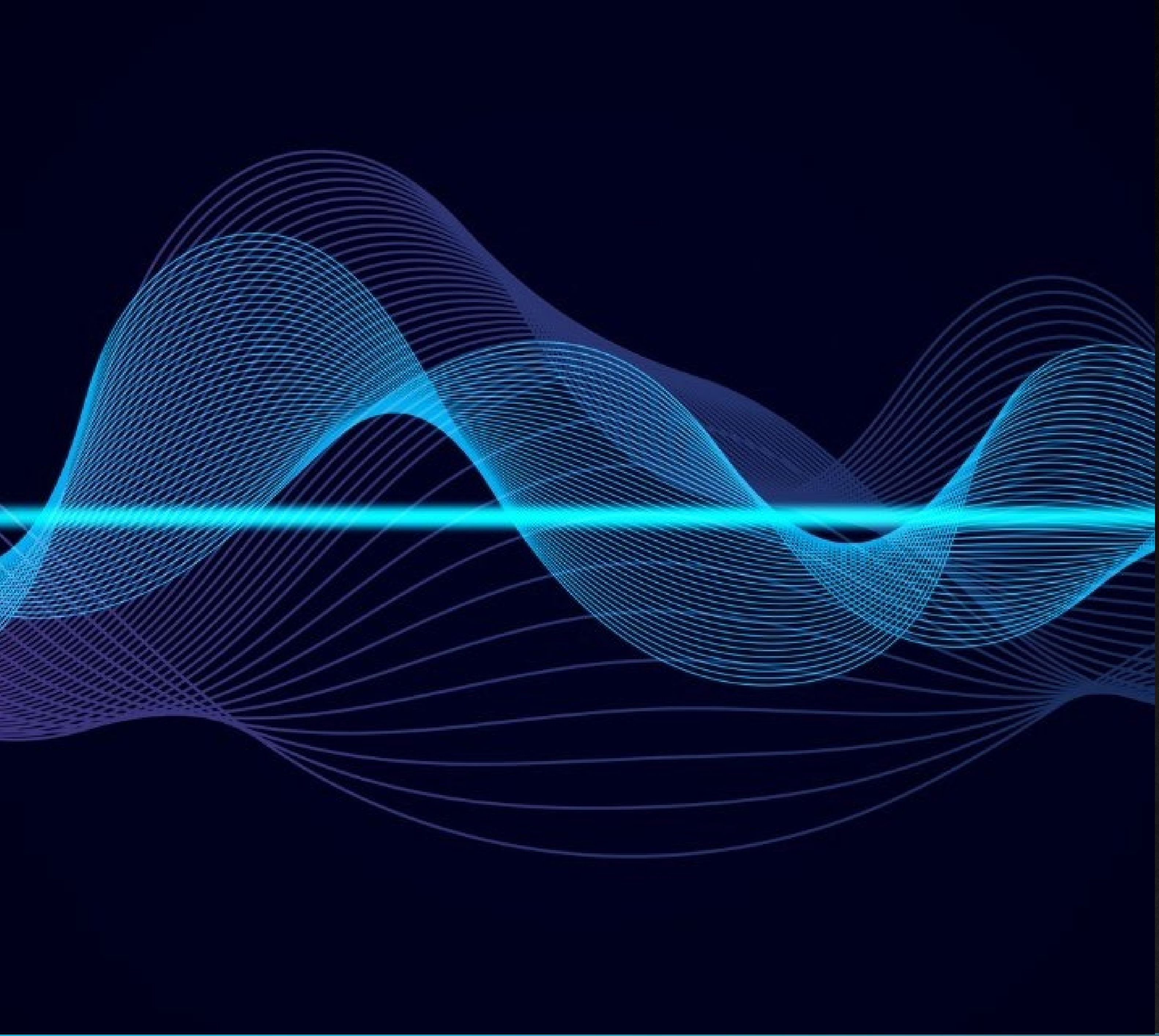


# Computational Physics

PHYS 6260

## Parallel Programming OpenMP / MPI

Announcements:

- HW6: Due Friday 3/1
- Project proposal, Due Friday 3/7

## Lecture Outline

We will cover these topics

- Finishing up OpenMP
- MPI

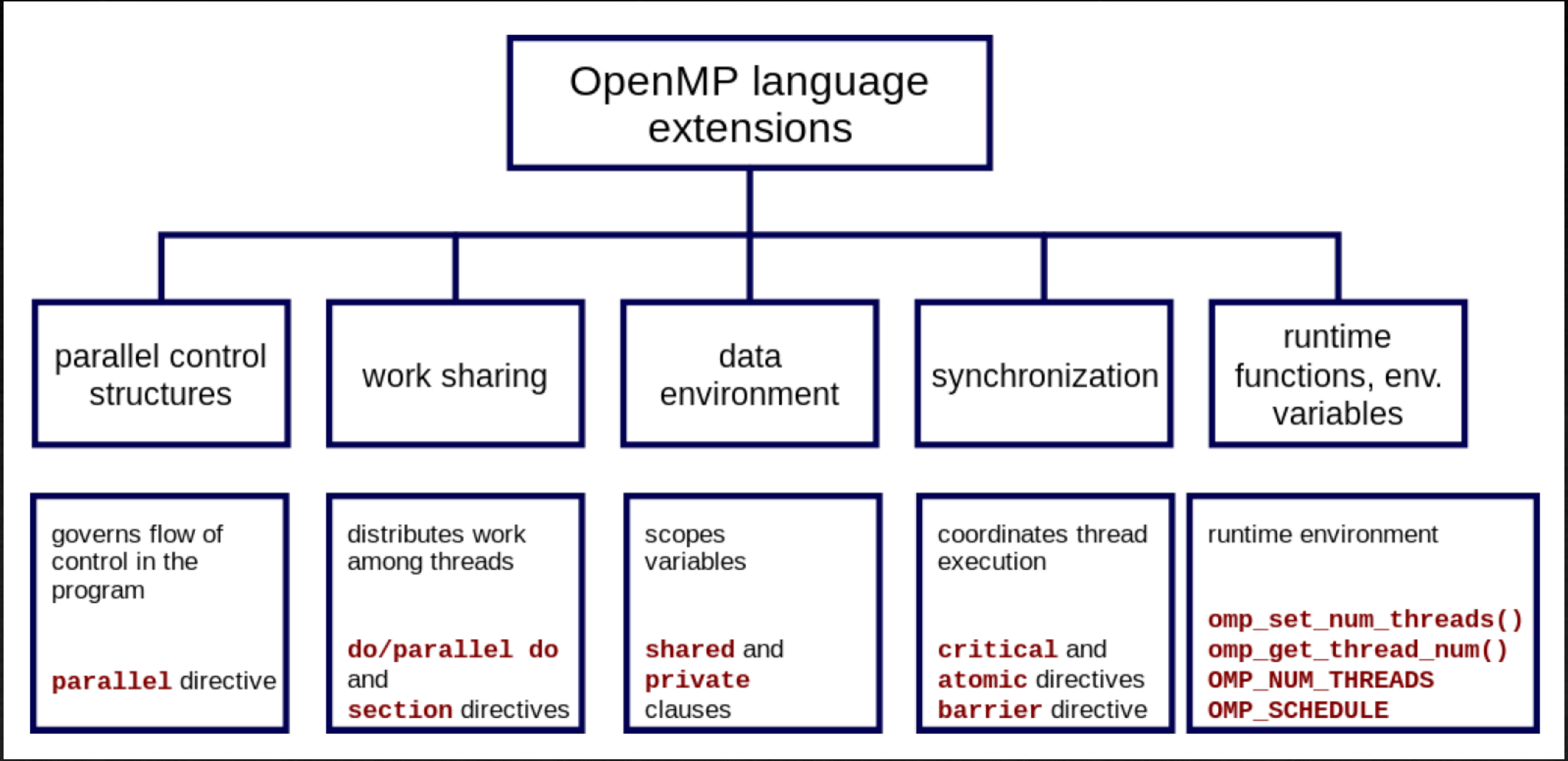
# OpenMP: Critical sections

- Within a parallel region, sometimes you need to ensure that only one thread at a time can write to a variable
- Consider the following

```
if ( a[i][j] > maxa ) {
   maxa = a[i][j]
   imax = i
   jmax = j
}
```

- If this is in the middle of a loop, what happens if 2 different threads meet this criteria?
- Marking this section as critical will ensure only one thread changes variables at a time
- Warning: critical sections can be very slow

# OpenMP: available modes


OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

## OpenMP: Porting to OpenMP

- You can parallelize your code piece-by-piece
- Since OpenMP directives look like comments to the compiler, your old version is still there
- Generally, you are not changing any of the original code – just adding directives

## OpenMP: Advanced options

- “if” clause tells OpenMP only to parallelize a region if certain conditions are met (e.g. a test of the size of an array)

- “firstprivate” is like “private” except each copy is initialized to the value from the original value

- “schedule” affects the balance of the work distributed to the threads. Options: static, dynamic, guided, chunk sizes, number of threads

## OpenMP: OpenMP in Python

- Python enforces a “global interpreter lock” that means only one thread can talk to the interpreter at any one time
  - OpenMP within pure python is not possible

- However, C extensions or Cython code called from python can use shared-memory parallelism
  - That is, the underlying code can have OpenMP directives

## MPI Parallelism

## MPI: Distributed parallelism

- The Message Passing Library (MPI) is the standard library for distributed parallel computing
- Now each core cannot directly see each other’s memory
- You need to manage how the work is divided and explicitly send and receive messages between MPI processes

## MPI: Hello world

- No longer do we simply use comments. We need to call routines in the MPI library.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

hostname[MAX_LENGTH-1] = '\0';
gethostname(hostname, MAX_LENGTH-1);
printf("P%04d/%04d: Hello world from %s\n",
       rank, size, hostname);

MPI_Finalize();
```

## MPI: Hello world

- MPI jobs are run with a command line executable, usually called mpirun, where you specify the number of cores

  ```
  mpirun –n 4 ./hello_mpi
  ```

- You need to install the MPI libraries on your machine
  - OpenMPI is the most popular on local machines
    e.g. “apt install openmpi-bin libopenmpi-dev” on Debian systems
  - MVAPICH2 is the most common on HPC platforms

## MPI: Concepts

- A separate instance of your program is run on each core – these are MPI processes
- Thread safety is not an issue here because each instance of the program is isolated from the others
- You need to tell the library the datatype of the variable you are communicating and its size (i.e. the buffer)
- Processors can be grouped together
  - Communicators label different groups
  - MPI_COMM_WORLD is the default communicator (all processes)

## MPI: Concepts

- Many types of operations: send/receive, collective (e.g. broadcast, gather, scatter)
- There are 100+ functions, but one can do any message passing with only 6
  - MPI_Init
  - MPI_Comm_Size
  - MPI_Comm_Rank
  - MPI_Send
  - MPI_Recv
  - MPI_Finalize
- More efficient communication can be done with the more advanced functions
- System vendors will usually provide their own MPI implementation that is well matched with the hardware

## MPI Example: Computing pi

- Let’s calculate pi by computing the integral (as compared to the trivially parallel Monte Carlo method)

$$\int_0^1 \frac{1}{1+x^2}\,dx = \arctan(x)\Big|_0^1 = \frac{\pi}{4}$$

- We will divide the [0,1] interval into $N_{\text{proc}}$ pieces so each process only sees its own interval
- Each core computes the sum for its interval
- The “root” process will add all the integrals at the end
- Let’s look at the code

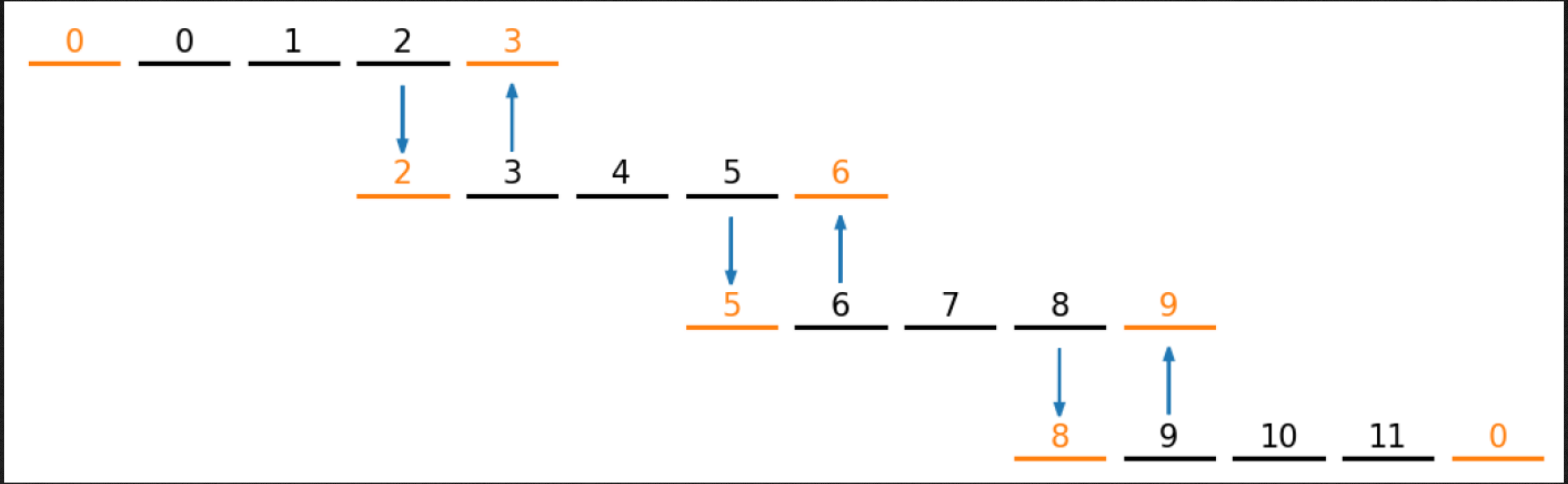
## MPI Example: Send / receive

- The main idea in MPI is sending messages between processes
- MPI_Send() and MPI_Recv() pairs provide this functionality
- This is a "blocking" send / receive
  - For sending code, the program resumes when it is safe to reuse the buffer
  - For receiving code, the program resumes when the message is received
- May cause network contention if the destination process is busy doing its own communication

- There are non-blocking sends, where you explicitly attach a buffer and the program continues after sending the data

# MPI Example: Send / receive

- Simple example (mimics ghost zone filling)
- One each process, allocate an integer array of 5 elements
- Fill the middle 3 with a sequence (e.g. proc 0: [0, 1, 2], proc 1: [3, 4, 5], …)
- Send messages to fill the left and right element with the corresponding element from the neighboring processes



## MPI Example: Send / receive

- Good communication performance often requires staggering the communication
- This is accomplished through overlapping communication and computation
- For example, you would send the data, compute something, and then receive any data necessary

- A combined MPI_Sendrecv() call can help avoid deadlocking
- Let’s look at the same code with such a call

## MPI Example: Relaxation

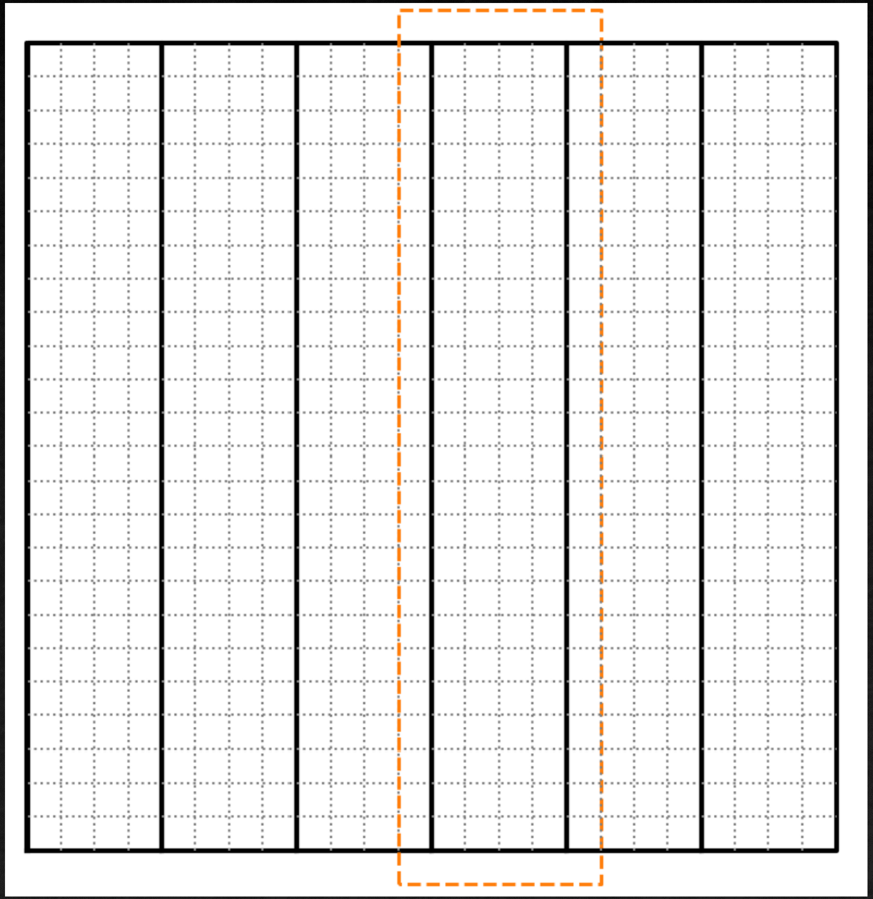
- Let’s do the same relaxation problem bu tnow using MPI instead of OpenMP

- In the OpenMP version, we allocated a single array covering the entire domain
- All cores had access to the whole array

- In the MPI version, each processor will allocate a smaller array, covering only a portion of the entire domain
- Each process can only access their own portion
- Necessary when data exceeds the memory limits of a single node / computer

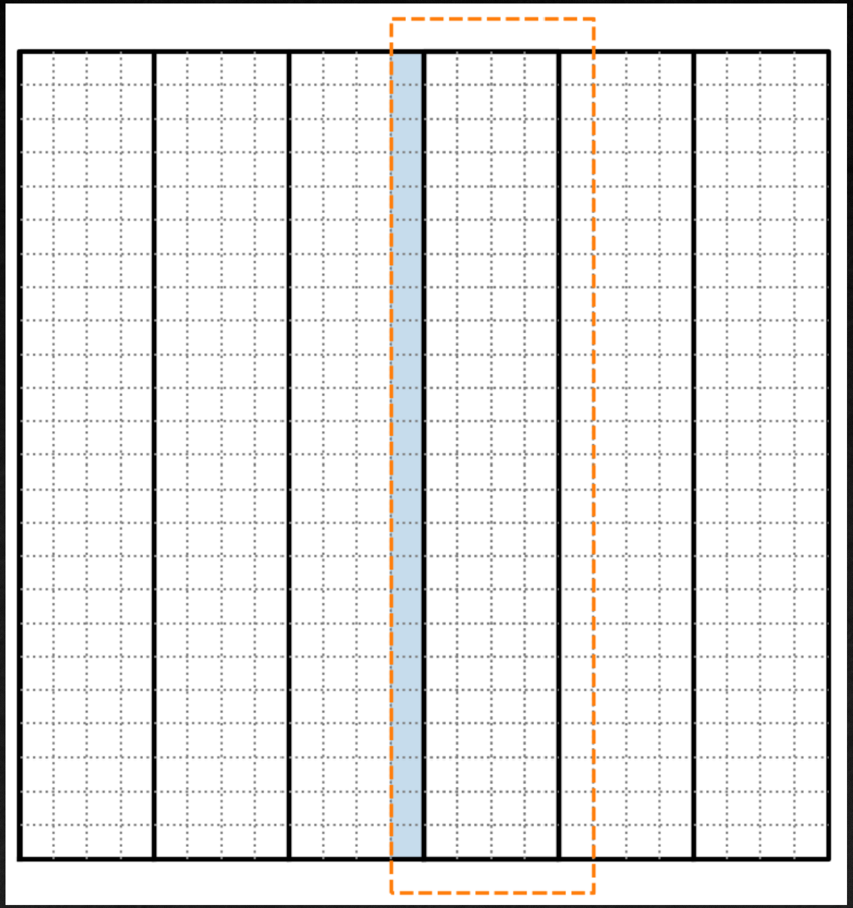
# MPI Example: Relaxation

- We will decompose the domain in 1D
- Each process allocates a slab that covers the full y-extent of the domain
- Width in x-direction = nx/nprocs
  - If not evenly divisible, some slabs will have differing widths
- Perimeter of 1 ghost zone surrounding each subdomain
- We will refer to a global index space [0:nx-1] x [0:ny-1]
- Memory needs to spread across all cores



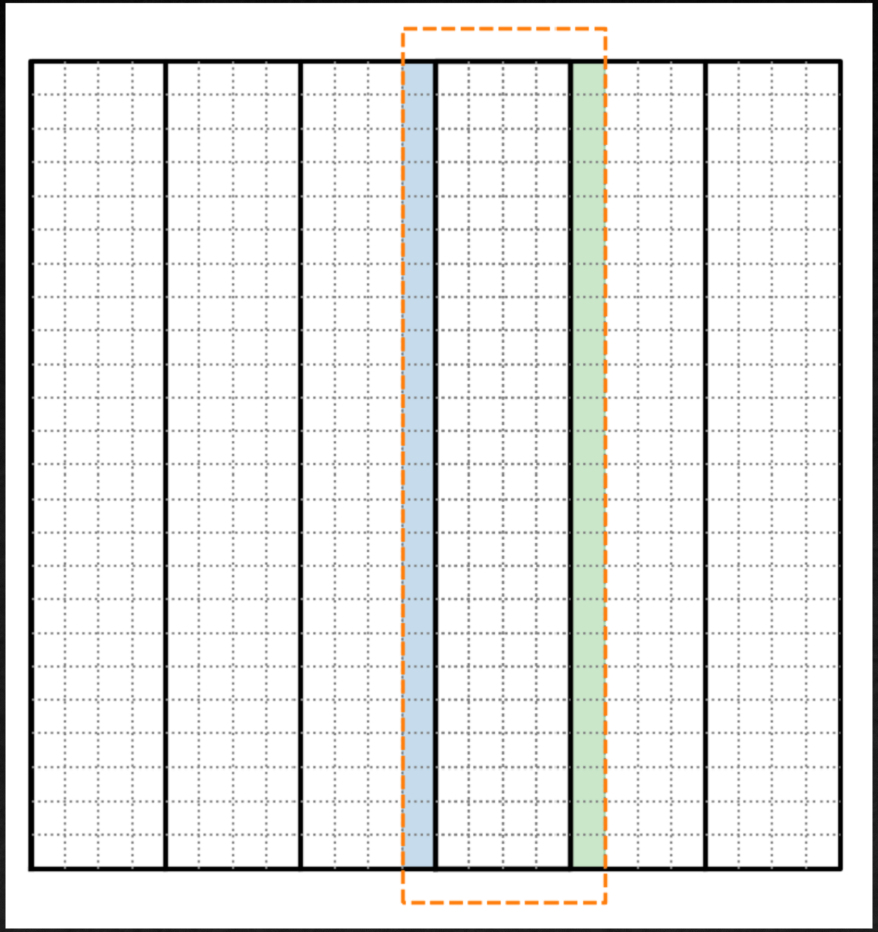
# MPI Example: Relaxation

- Left set of ghost zones are filled by receiving a message from the core (slab) to the left



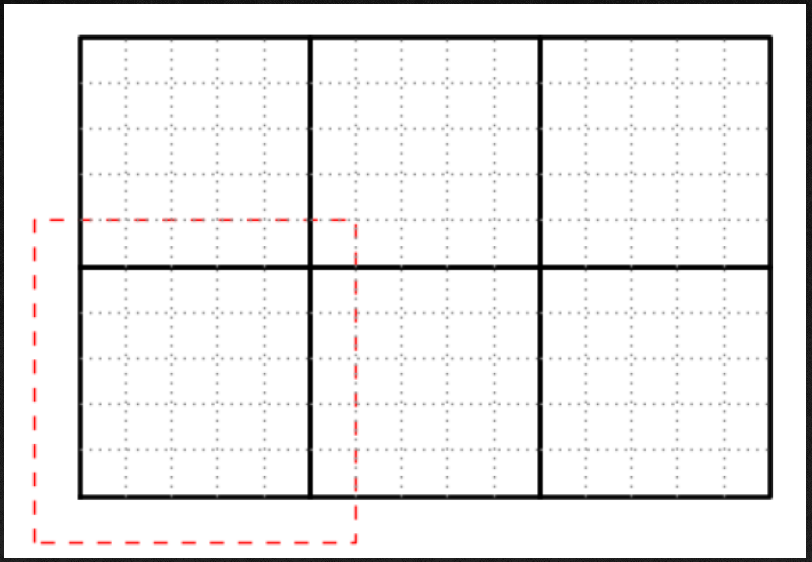
# MPI Example: Relaxation

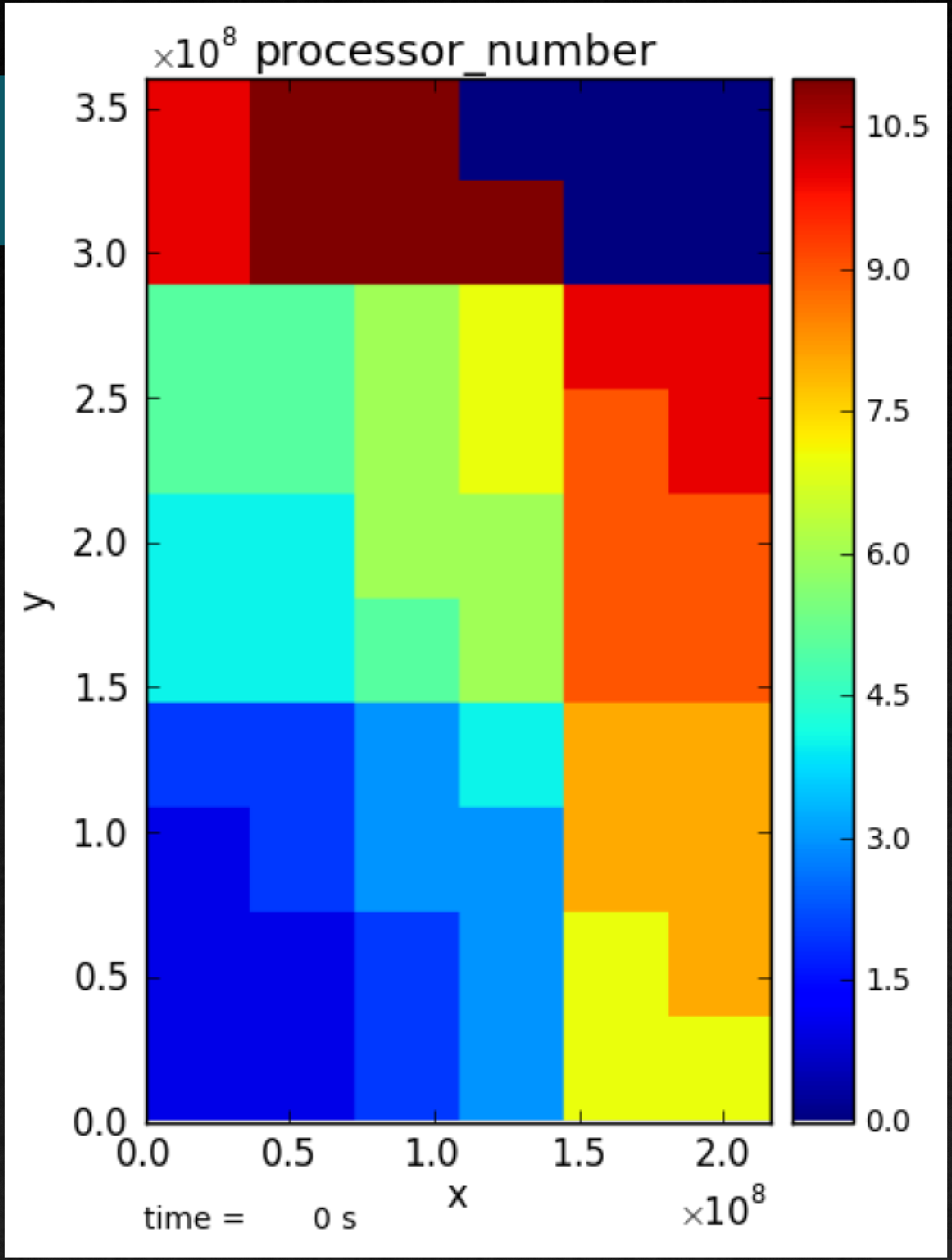
- Left set of ghost zones are filled by receiving a message from the core (slab) to the left
- Right set of ghost zones are filled by receiving a message from the process (slab) to the right
- Top and bottom ghost zones are physical boundaries



# MPI Example: Relaxation

- Generally speaking, one wants to minimize the surface-to-volume ratio
- Reduces communication





## MPI Example: Relaxation

- Most of the parallelism comes in the ghost zone filling
  - Fill left GZs by receiving data from the process to the left
  - Fill right GZs by receiving data from the process to the right
  - Send / receive pairs: we want to try to avoid contention (this can be very tricky, and people spend a lot of time worrying about this!)
- On the physical boundaries, we simply fill as usual
- The way we have written the code, our relaxation routine doesn't need to do any parallelism itself.
  - It just operates on the domain it is given
- For computing a norm, we will need to reduce the local sums across processes
- Let's look at the code

## Weak versus Strong Scaling

- In assessing the parallel performance of code, there are two common methods

- Strong scaling: keep the problem size fixed and increase the number of cores
  - Eventually you will become work-starved
  - Scaling will stop when communication and overhead dominate

- Weak scaling: increase the amount of work in proportion to the number of cores
  - Perfect scaling will result in the same “wall clock” time for all core counts

## Parallel debugging

- There are parallel debuggers, but they are usually expensive
- It’s possible to spawn multiple gdb (GNU debugger) sessions, but this gets out of hand quickly – “mpirun -n 2 xterm -e gdb ./a.out”
- Print is still your friend
  - Run a small of a problem as possible on as few cores as necessary
- Some round off differences are to be expected from sums (different order of operation … numeric addition is not associative)

## Hybrid parallelism

- To get good performance on current HPC platforms (>1k cores), you need to use hybrid parallelism

- OpenMP within a node / socket, MPI across nodes

- For example, in our MPI relaxation code, we could split the loops over each subdomain over multiple cores on a node using OpenMP
- Then we have MPI to communicate across nodes and OpenMP within the nodes
- The hybrid approach is often needed to get the best performance on big machines