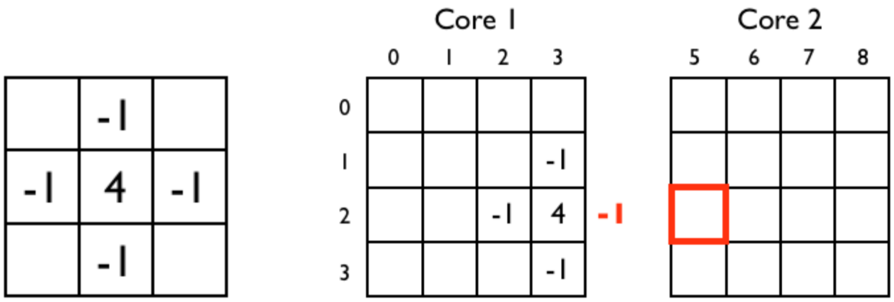
MPCS 51087
Problem Set 2

Summer 2019

1 Distributed Memory Advection

Due Date: Thursday, July 25, 2019 by 11:59 pm

In problem set 1, you were tasked with writing a shared memory code that used threads to allow for parallelization across all CPUs on any given node. However, for larger problems that require the computational resources of many nodes to solve, we often must use a distributed memory model of parallelism. One way of achieving distributed memory parallelism is to use a domain decomposition scheme via MPI that allows for multiple computational nodes to handle separate sub-domains of the problem in parallel. However, it is important to note that when domain decomposing, the subdomains of the problem are not independent of one another. Take for example the arbitrary five point stencil given in Figure 1. Because the stencil requires knowledge of each i, j point's neighbors, and those neighbors may not exist on the local subdomain, communication is required to synchronize data between the various boundary cells on each sub-domain.



(a) 5-Point Stencil

(b) Stencil that needs a cell from its neighbor

Figure 1: Arbitrary 5-point stencil applied to a domain decomposed problem. (Image Source: Kjolstad and Snir, "Ghost Cell Pattern", 2010)

Exchange of border cell data is often called a "ghost-cell" exchange pattern. Typically, as shown in Figure 2, we will bundle up entire rows or columns of data to send together in a single message rather than sending MPI messages for each required cell individually.

Your task for this assignment is to take your advection code from Problem Set 1 and implement a distributed memory 2D domain decomposition scheme using MPI. Because our advection kernel equation from PSet 1 (Equation 1) requires knowledge of each i, j point's neighbors, you will need to use the ghost cell design pattern. This will require implementation of methods for storing and exchanging (via MPI) the ghost cells each iteration.

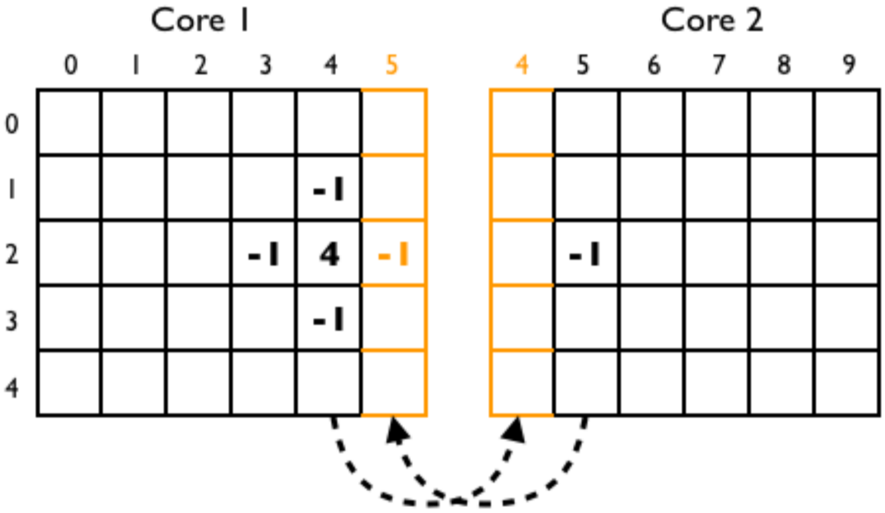


Figure 2: Ghost cell exchange pattern example. (Image Source: Kjolstad and Snir, "Ghost Cell Pattern", 2010)

$$\boxed{C^{n+1}_{i,j} = \frac{1}{4}\left(C^n_{i-1,j} + C^n_{i+1,j} + C^n_{i,j-1} + C^n_{i,j+1}\right) - \frac{\Delta t}{2\Delta x}\left[u\left(C^n_{i+1,j} - C^n_{i-1,j}\right) + v\left(C^n_{i,j+1} - C^n_{i,j-1}\right)\right]} \qquad (1)$$

1.1 MPI Parallelization with Blocking Communicators (20 pts)

Write a parallel advection solver using a two-dimensional Cartesian MPI decomposition and blocking communication. A few notes regarding your implementation:

- You should use the MPI Cartesian communicator functions, which are helpful for organizing 2D communication, such as `MPI_Cart_create()` and `MPI_Cart_shift()`, etc.
- Use blocking MPI communication routines for this section of the assignment.
- You still need to have periodic "wrap-around" boundary conditions. This means that all nodes (even exterior nodes) must be able to exchange ghost cell data with their 4 neighbors.
- You do not need to be able to decompose to any arbitrary number of MPI ranks, it is fine to enforce that only powers of two are acceptable so that 2D mapping is easy. Additionally, it is fine to enforce that the dimension of the problem N is evenly divisible by the square root of the number of MPI ranks. However, be sure to document and assert this condition in your code, so that the code does not seg fault if run with invalid parameters.
- Watch out for deadlocks!

Later in this problem set, you will be coding up other methods of parallelism (non-blocking and hybrid MPI-OpenMP), so your code will need to have a way of easily selecting which parallelism type to use. This should be done through the command line interface, and should extend the same interface from Problem Set 1. We will label our various methods of parallelism as follows:

- Serial = "serial"
- Shared memory threading via OpenMP = "threads"
- Distributed memory MPI (blocking) = "mpi_blocking"

- Distributed memory MPI (non-blocking) = “mpi_non_blocking”
- Hybrid MPI + OpenMP = ”hybrid”

You will also need to have a command line argument to specify the number of threads per MPI rank to use. So, for example, if we wanted to run with the following parameters:

- N = 3000 (Matrix Dimension)
- NT = 20000 (Number of timesteps)
- L = 1.0 (Physical Cartesian Domain Length)
- T = 1.0e6 (Total Physical Timespan)
- u = 5.0e-7 (X velocity Scalar)
- v = 2.85e-7 (Y velocity Scalar)
- t = 1 (OpenMP threads per process)

these would be input to your program when running in MPI blocking mode on 16 ranks as:

```
mpirun -n 16 ./my_advection_program 3000 20000 1.0 1.0e6 5.0e-7 2.85e-7 1 mpi_blocking
```

Or, if we wanted to run in hybrid mode with 16 threads per process, and 4 MPI ranks, this would be run as:

```
mpirun -n 4 ./my_advection_program 3000 20000 1.0 1.0e6 5.0e-7 2.85e-7 16 hybrid
```

1.2 MPI Plotting (15 pts)

When running a domain decomposed code, we are interested not only in finding a way of distributing the computational work out between nodes, but also in reducing the required memory footprint per node. With domain decomposition, we can use the combined memory resources of many nodes to hold a problem in memory that is much larger than what might be possible using only a single node. For instance, current leadership class computers have on the order of petabytes of total system memory available.

This means that if we wish to store data from our simulation for plotting or analysis purposes, traditional serialization schemes do not always work. We cannot simply output the global domain to disk using normal file i/o, as it may be possible when running large problems that no one process can fit all of the domain in memory at once. Rather, we are forced to do one of three things:

1. Write MPI code to manually synchronize all processors, such that each processor in turn opens a file, writes it’s share to it, then closes it before signaling that it’s done so the next process can open the file.
2. Write bespoke MPI code to have all ranks take turns sending their data to a single designated rank, and have this centralized i/o rank handle and coordinate everything that’s written to the file.
3. Use the MPI library file i/o commands (e.g., `MPI_File_open()`, `MPI_File_write_at_all()`, etc) to handle all of this a lot more easily.

For this assignment, we will require you to use option (3) above to implement MPI File i/o capabilities that let you generate output data matrix files which you can then plot using your plotting scripts from the last homework. Your outputted files and plots should be identical to those generated using your serial code or MPI code given the same inputs.

As part of your submission, run a problem in serial and with MPI on at least 28 ranks and have both generate an output data matrix. Write a short script or use a system command to confirm the outputted data files are identical, and submit plots of both in your PDF writeup. This will confirm that there were no errors in your MPI file i/o routines and may help you to fix any bugs as you develop your code.

1.3 MPI Parallelization with Non-Blocking Communicators (10 pts)

Create a new version of your parallel code that uses non-blocking MPI communicators, such that for each rank, information can be transmitted to and from all four Cartesian neighbors at the same time. It is not enough to just replace function calls with their non-blocking versions, you must ensure that communication between all neighbors can take place at the same time.

You should implement this in the form of new function(s) that exist within the same code base as your blocking version. You must make it easy to switch between the various methods of parallelism via command line, using the interface described in subsection 1.1.

1.4 Strong Scaling Study (15 pts)

Perform strong scaling studies out to at least 100 MPI ranks that are mapped to their own physical CPUs. On Midway, using the "broadwl" partition, running the largest configuration with 100 ranks will require 4 nodes. You do not need to test your code on all processor counts, just doing 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100 is fine. You should test and compare both your blocking and non-blocking MPI implementations on the same plot. Follow the same instructions from PSet 1 for performing a strong scaling study, using the following input parameters:

- N = 5040
- NT = 300
- L = 1.0
- T = 1.0e3
- u = 5.0e-7
- v = 2.85e-7

A few notes to remember when running your tests on Midway:

- Make sure your plotting routines are disabled when doing the performance studies.
- Each run should be performed on an appropriate allocation configuration. Some wasteage is unavoidable (e.g., testing 36 ranks will require running on a 56 CPU allocation), but you should not run your serial test case on a 4 node, 112 CPU allocation.
- All sbatch submissions should be put in with a very low wall time of 5 minutes. The longest test run (the serial case) should execute in about 2 minutes, with higher core counts only requiring a few seconds. Runtimes longer than 5 minutes should not be needed for any configuration.
- If your code does not complete within the allotted 5 minutes when running in parallel, do not extend the walltime and try again. You code has likely deadlocked and needs to be debugged!

Next, re-run your OpenMP code from PSet 1 with the above parameters using 25 cores of a single node, and compare the runtime results to your MPI blocking and non-blocking codes on 25 cores. You do not need to repeat a full scaling study in OpenMP, just do the one run on 25 cores. Which model of parallelism is fastest in this case, and why might that be?

1.5 Weak Scaling Study (15 pts)

Perform a weak scaling study out to at least 100 MPI ranks that are mapped to their own physical CPUs (again, it's fine to just do tests on 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100 CPUs). You should test and compare both your blocking and non-blocking implementations on the same plot. Follow the same instructions from Pset 1 for performing a weak scaling study for the problem size below:

- N = 800 (increase as you add MPI ranks)

- NT = 300
- L = 1.0
- T = 1.0e3
- u = 5.0e-7
- v = 2.85e-7

1.6 Hybrid Parallelism: MPI + OpenMP (15 pts)

You final task for this assignment is to implement a hybrid parallel scheme that allows for both MPI (either blocking or non-blocking, your choice) and threading. This can potentially be useful in cases where there are high communication costs between MPI ranks, but multiple nodes must be used to have enough computational resources to run a problem. For instance, consider a 4 nodes allocation with each node having 28 CPUs. While an MPI only scheme would require 112 MPI ranks, a hybrid parallel scheme can allow your code to be run using for instance only 4 MPI ranks (one for each computational node), while 28 threads per MPI rank can be used for on-node parallelism. In the context of our domain decomposed advection codes, this hybrid scheme reduces the overall amount of ghost cell exchange information required. However, it's worth noting that hybrid parallelism is not guaranteed to be faster or slower than pure MPI, as threads have their own overhead, and the smaller subdomains that accompany higher MPI rank counts can actually provide improved data locality in some cases.

With your implementation, you should do a set of test runs on 4 nodes of Midway using 16 cores on each node. We will uses fewer than the 28 cores per node that are available so as to ensure that the $\sqrt{\text{MPI Ranks}}$ is an integer number for all the configurations we are testing. Run three tests with the following configurations:

1. Hybrid mode with 1 MPI rank per node and 16 threads per rank.
2. Hybrid mode with 4 MPI ranks per node and 4 threads per rank.
3. MPI only mode (either blocking or non-blocking, your choice), with 16 MPI ranks per node and 1 thread per rank.

In your writeup, give the timing results for all three runs, and discuss what you think is causing any performance differences. Input parameters for the runs should be:

- N = 10000
- NT = 200
- L = 1.0
- T = 1.0e3
- u = 5.0e-7
- v = 2.85e-7

NOTE: Just a reminder to disable your output routines, and do not output a data file for this problem scale, as the file for these parameters is quite large and we don't have a huge disk quota in Midway.

As was the case for your non-blocking routines, this parallelization scheme should be implemented in the form of a new set of function(s) that exist within the same code base and can be selected from via the command line interface described in subsection 1.1.

1.7 Compiling, Code Cleanliness, and Documentation (10 pts)

We plan on compiling and running your code as part of grading it. You must include a makefile capable of compiling your code into a runnable executable. Also, while we are not grading to any rigid coding standard, your code should be human readable and well commented so we can understand what is going on.

1.8 What to Submit

You submission should be in the form of a zip or tar file containing the following items:

- Your source code for all three parallelization schemes (Blocking MPI, Non-Blocking MPI, and MPI+OpenMP). As part of grading, we may want to compile and run your different parallelizations, so you must make it easy for us to switch between them and document how this is done.
- A makefile
- Your plotting script(s)
- A single writeup (in PDF form) containing all plots and discussions asked for in the assignment