High Performance Computing

MPCS 51087

Problem Set 2

Distributed Memory Advection

Spencer Zepelin

July 25, 2019

Compiling and Running the Code

The program was written in C with the c99 standard. All testing and data collection was performed using MPICH rather than Open-MPI. Source code can be found in the file "advection_modes.c". A makefile has been provided to assist in compiling. Running the command "make" will make the executable named "mpirun" by running the following command:

mpicc -std=c99 -o mpirun advection_modes.c -fopenmp -lm

Calling "make try" will ensure the executable has been compiled and then run it with MPI enabled with the following default arguments:

- nprocs = 36 (Number of MPI Processes)
- N = 5040 (Matrix Dimension)
- NT = 300 (Number of timesteps)
- L = 1.0 (Physical Cartesian Domain Length)
- T = 1.0e3 (Total Physical Timespan)
- u = 5.0e-7 (X velocity Scalar)
- v = 2.85e-7 (Y velocity Scalar)
- nt = 4 (Number of Threads)
- mode = mpi_blocking

Calling "make run" will ensure the executable has been compiled and then run it serially with MPI disabled with the following default arguments:

- N = 5040 (Matrix Dimension)
- NT = 300 (Number of timesteps)
- L = 1.0 (Physical Cartesian Domain Length)
- T = 1.0e3 (Total Physical Timespan)
- u = 5.0e-7 (X velocity Scalar)
- v = 2.85e-7 (Y velocity Scalar)
- nt = 1 (Number of Threads)
- mode = serial

To run the executable with other arguments in either serial or OMP threaded mode, simply run:

./mpirun N NT L T u v nt mode

To run the executable with other MPI-enabled modes, run:

mpiexec -n nprocs ./mpirun N NT L T u v nt mode

In both cases, arguments represent the values described in the default case. As outlined in the prompt, the mode argument has five valid options: serial, threads, mpi_blocking, mpi_non_blocking, and hybrid.

Data from an execution will be output to three files: f0.bin, f1.bin, and f2.bin. They represent the data after initialization, after half of the advection timesteps, and after the final timestep. All batch scripts used on Midway are included in the directory “batch_scripts”.

Code Design

As outlined in the prompt, a single executable is capable of running all five modes. After taking and processing the necessary arguments and validating that the Courant Stability Condition is met, the program moves into either the MPI-disabled or MPI-enabled branch. From here, the program liberally makes use of 13 helper functions to dictate the manner in which functional steps are performed, namely, whether or not threading is used and whether message passing is performed in a blocking or non-blocking manner. These helper functions should make the primary code branches more legible. Additionally, a number of preprocessor definitions were utilized to make operation modes and ghost arrays used for message passing more intuitive. For MPI modes, I enforced processor positioning rather than using the cartesian decomposition and wrote to file using a coordinator rank rather than collective IO. Modifying either design decision could potentially improve the program’s performance.

MPI Data Plotting

The following commands were used to run the program serially and with over 28 MPI ranks (36, in total) using blocking message passing after compiling the program. Only minor modifications were made to the program such that the second execution did not overwrite the data files of the first.

Serial:
./mpirun 300 20000 1.0 1.0e6 5.0e-7 2.85e-7 1 serial

MPI:
mpiexec -n 36 ./mpirun 300 20000 1.0 1.0e6 5.0e-7 2.85e-7 1 mpi_blocking

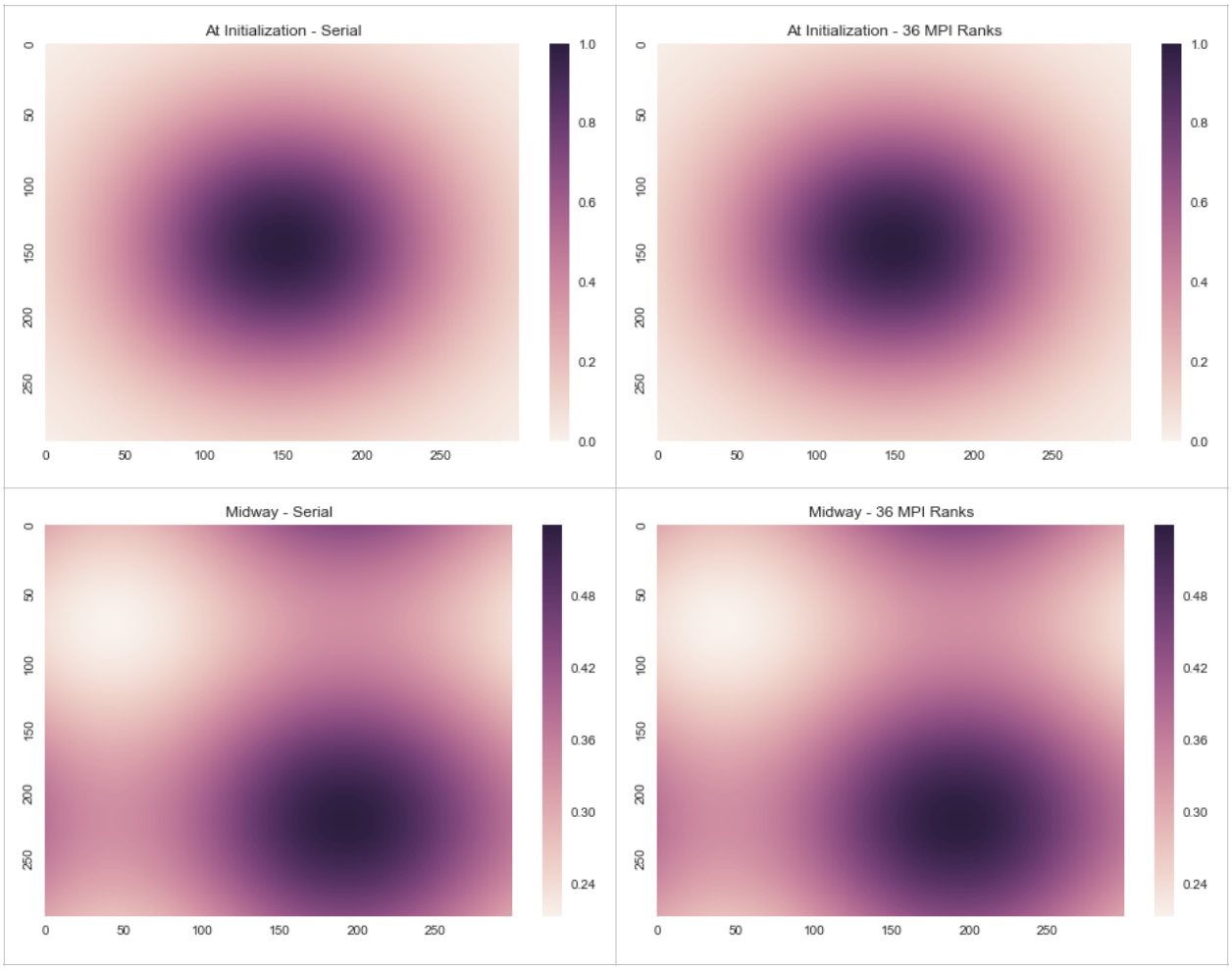
In addition to plotting the data, I ran the following commands to ensure the binary files were identical:

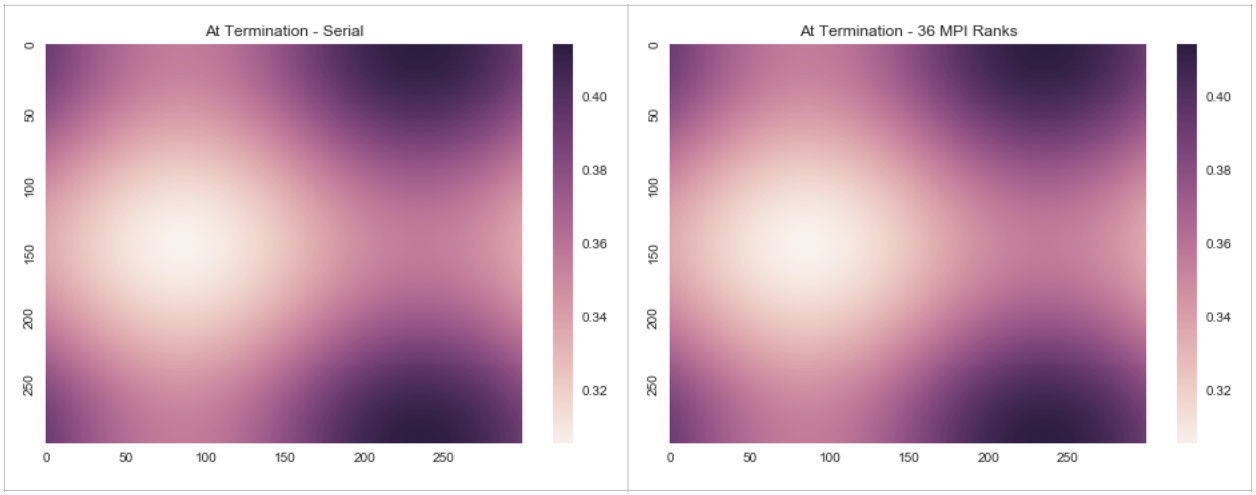
- diff f0.bin fb0.bin
- diff f1.bin fb2.bin
- diff f2.bin fb2.bin

The commands indicated the data files given were identical in all three cases.

To plot the data, I used the REPL environment of Jupyter Notebook with Python and the numpy, matplotlib, and seaborn libraries. The notebook file is saved as “ps2_plotting.ipynb”. For convenience, I have copied the code therein to “plotting.py”, but the code has not been modified to save to file and, as such, should simply be used as a reference. Row zero, column zero is in the top left corner of the plots and the highest row and column in the bottom left. As such, simulations with positive X and Y scalar values should see the blob drift down and to the right as it disperses.

These plots represent, respectively, the test simulation after initialization, after half the timesteps, and at termination.





Strong Scaling Study

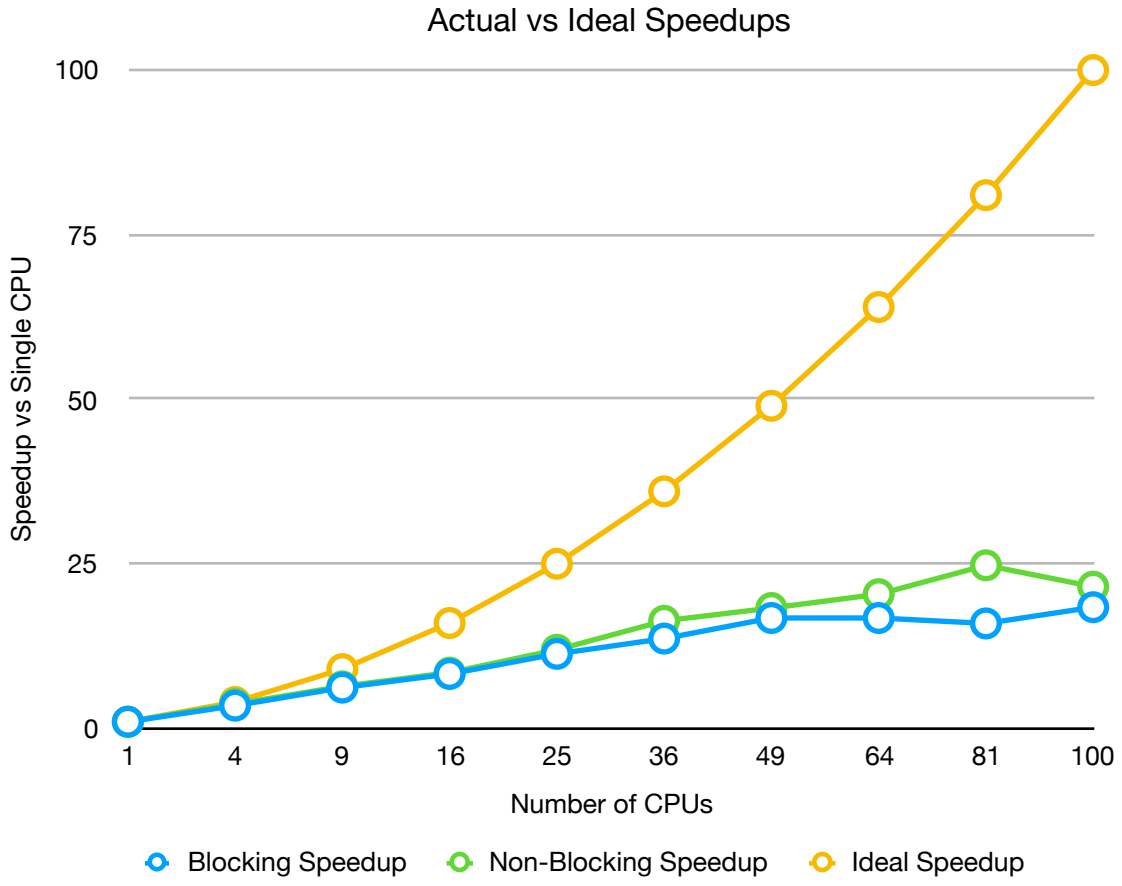
Per the prompt, the following arguments were used in the study:

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

Both modes were tested with 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100 processes. Data from the tests, as well as plots of the measured speedup versus ideal against the number of CPUs, are below.

Strong Scaling Study (Table 1)

CPUs	Nodes Necessary	Tasks per node	Blocking Runtime	Non-Blocking Runtime	Ideal Runtime	Blocking Speedup	Non-Blocking Speedup	Ideal Speedup
1	1	1	22.63	23.03	22.63	1.00	1.00	1
4	1	4	6.57	6.23	5.66	3.44	3.70	4
9	1	9	3.66	3.61	2.51	6.18	6.38	9
16	1	16	2.74	2.72	1.41	8.26	8.47	16
25	1	25	2.00	1.93	0.91	11.32	11.93	25
36	2	18	1.66	1.41	0.63	13.63	16.33	36
49	2	25	1.35	1.26	0.46	16.76	18.28	49
64	3	22	1.35	1.13	0.35	16.76	20.38	64
81	3	27	1.42	0.93	0.28	15.94	24.76	81
100	4	25	1.23	1.07	0.23	18.40	21.52	100



The performance improvement, while significant, falls far short of the ideal improvement. Both blocking and non-blocking tests saw almost uniform improvement in performance with an increases in CPUs. The non-blocking topped out at nearly a 25x speedup while blocking reached nearly 20x.

If only 5% of the code cannot be parallelized, a 20x speedup is the optimal result. As such, even though the speedup falls far below the ideal, this result is well within the realm of possibility and indicates strong performance.

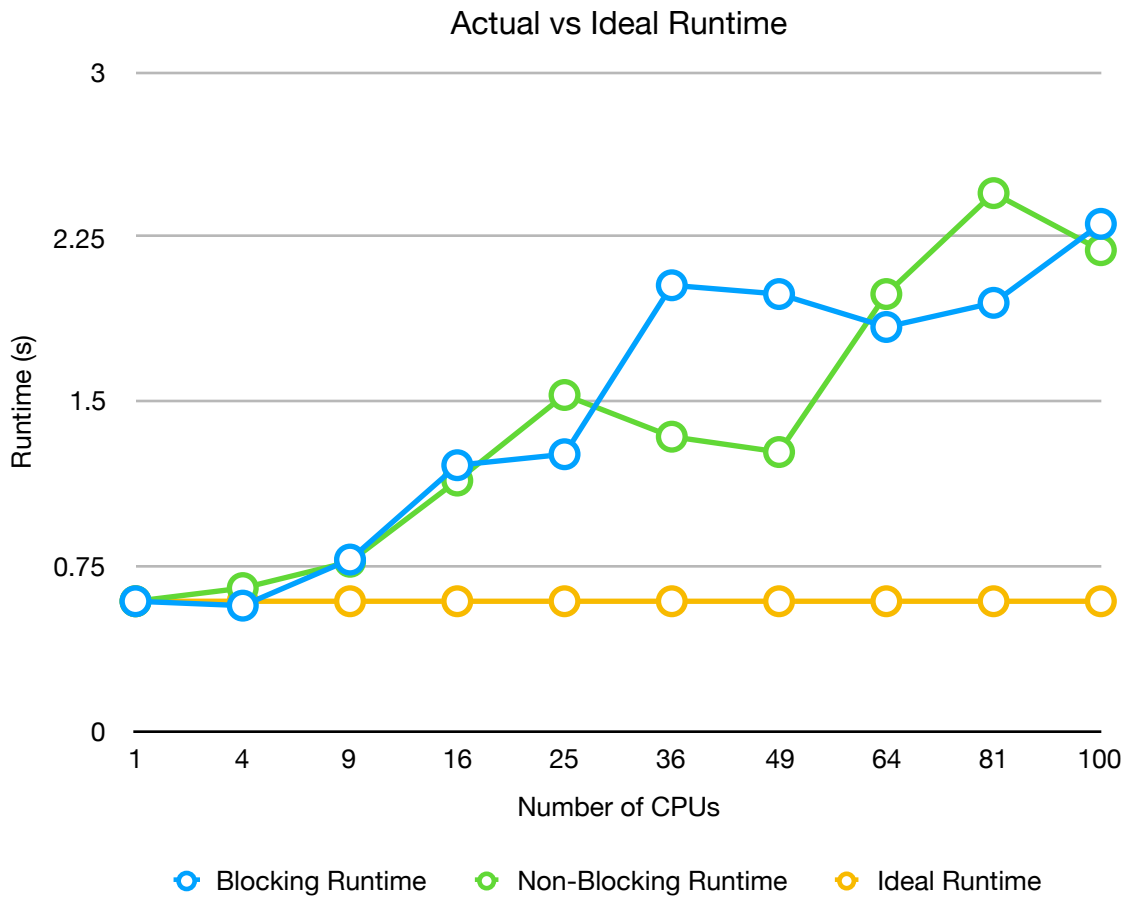
MPI-disabled OMP parallelism with 25 threads over 25 cores ran in 1.86 seconds vs blocking's 2.00 seconds and non-blocking's 1.93 seconds. Under these testing conditions, I speculate that —as the data suggests—simple threading may be the most effective. As the problem size continues to grow larger, the overhead associated with message passing is amortized, and it could likely outperform simple threading.

Weak Scaling Study

Per the prompt, the following arguments were used in the study:

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

Both modes were tested with 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100 processes with N increasing with the number of cores to ensure each study had an equivalent problem size for each core. Data from the tests, as well as a plot of the ideal time versus actual against the number of cores, are below.



Weak Scaling Study (Table 2)

CPUs	Nodes Necessary	Tasks per node	N	N^2	Blocking Runtime	Non-Blocking Runtime	Ideal Runtime
1	1	1	800	640000	0.59	0.59	0.59
4	1	4	1600	2560000	0.57	0.65	0.59
9	1	9	2400	5760000	0.78	0.77	0.59
16	1	16	3200	10240000	1.21	1.14	0.59
25	1	25	4000	16000000	1.26	1.53	0.59
36	2	18	4800	23040000	2.03	1.34	0.59
49	2	25	5600	31360000	1.99	1.27	0.59
64	3	22	6400	40960000	1.84	1.99	0.59
81	3	27	7200	51840000	1.95	2.45	0.59
100	4	25	8000	64000000	2.31	2.19	0.59

The weak scaling study shows what is likely a logarithmic relationship (though appearing more linear due to the polynomial scale of the x-axis) between the number of cores and the slowdown compared to ideal runtime. I speculate the added cost is a consequence of greater volumes of data being passed between processes. That said, the overhead of establishing and coordinating message passing likely remains roughly constant as we scale up the number of CPUs. Even still, as the number of cores grows, so do the number of processes which need to communicate with a process on a separate node, likely costing some amount of performance.

Hybrid Parallelism

Per the prompt, the following arguments were used in the study:

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

Data from the three trials indicating the number of nodes, ranks per node, and threads per rank are included below.

Hybrid Parallelism (Table 3)

Nodes	MPI Ranks/Node	Threads/Rank	Mode	Runtime
4	1	16	hybrid	2.24
4	4	4	hybrid	1.94
4	16	1	mpi_non_blocking	2.44

While this is an admittedly small study, the results indicate that—as we might have expected—a hybrid approach balancing overhead between MPI ranks and OMP threads is the optimal approach. Holding the number of processes*threads constant at 64, we see the best performance when we minimize the number of both processes and ranks. In this way we reduce the amount of message passing that has to take place while giving each thread a meaningful chunk of work to perform. I speculate that the thread-heavy model is missing out on the performance to be gained by message passing while creating more thread overhead while the message-passing-heavy model is adding additional overhead for message passing and missing out on thread optimization.