High Performance Computing

MPCS 51087

Problem Set 1

Serial and Threaded Advection

Spencer Zepelin

July 15, 2019

Compiling and Running the Code

Programs were written in C. Source code can be found in the directories “src_serial” and “src_parallel”. Makefiles are provided in each directory. Running the command “make” will make the respective executable: *advection* for the serial version and *advection_parallel* for the parallel. Calling “make run” will ensure the executable has been compiled and then run it with the following default arguments:

- N = 400 (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)
- nt = 4 (Number of Threads; *parallel version only*)

To run an executable with other arguments, simply run:

./advection N NT L T u v

Or,

./advection_parallel N NT L T u v nt

In both cases, arguments represent the values described in the default case.

Code Design

The serial and parallel versions of the code are nearly identical. Necessary arguments are taken off the command line. The serial version uses <time.h> to record the total runtime, while the parallel uses the function built into omp. Arguments are parsed and echoed back to the stdout and a total memory estimate

is generated using the matrix size. We note that while the actual memory size will not be precisely the size of the matrices, their memory space will dominate. Memory is allocated for two matrices: one for the current step and one for the subsequent step. In my implementation, I treat the matrices as 1d arrays and handle the indexing logic myself.

The initial data is calculated with the assistance of a helper function to initialize a 2d gaussian distribution. The parallel version parallelizes this steps across its threads.

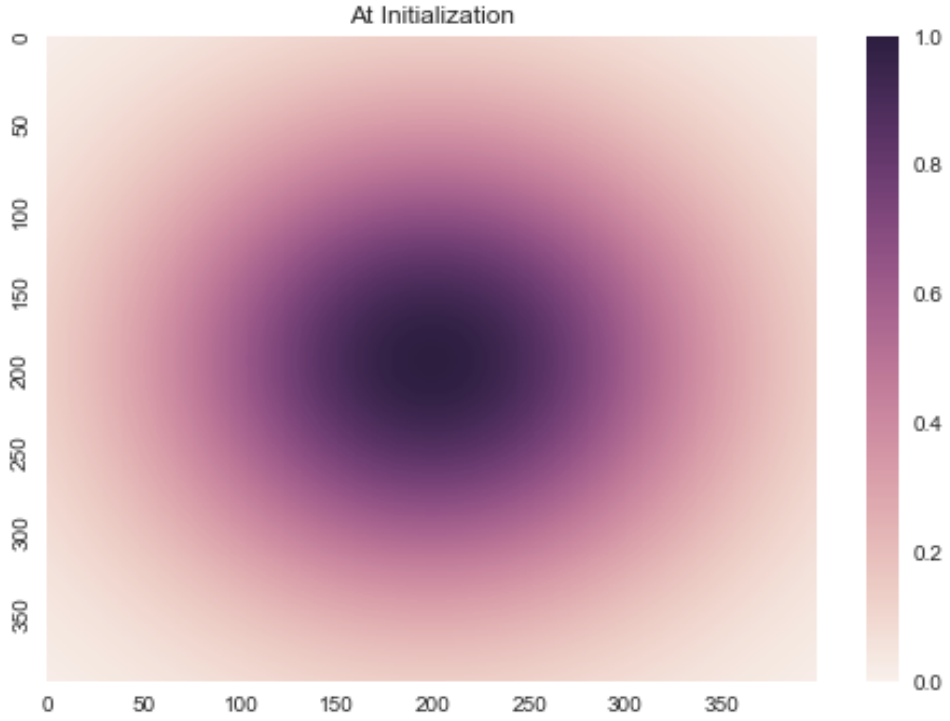
Next, the program loops over timesteps and then over every location in the matrix calculating the next step based on the current. The parallel version again parallelizes the calculations internal to each timestep. Finally, the pointers for the current step and the next step are swapped in preparation for the subsequent timestep.

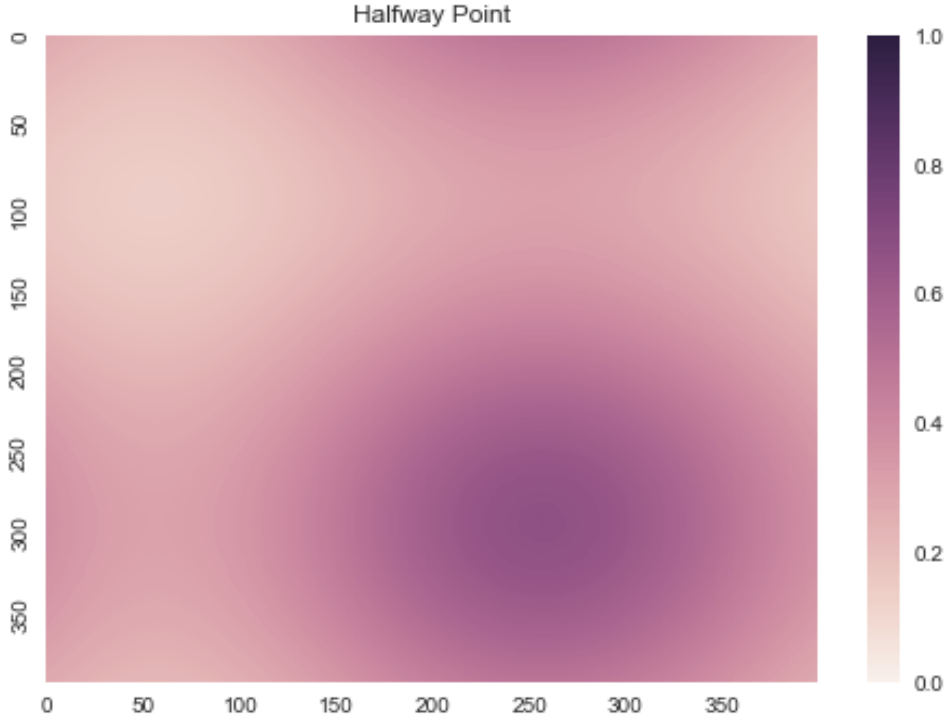
Data is written to file after the data is initialized, after the middle timestep, and after the final timestep. The total runtime is printed to stdout at the conclusion of the program.

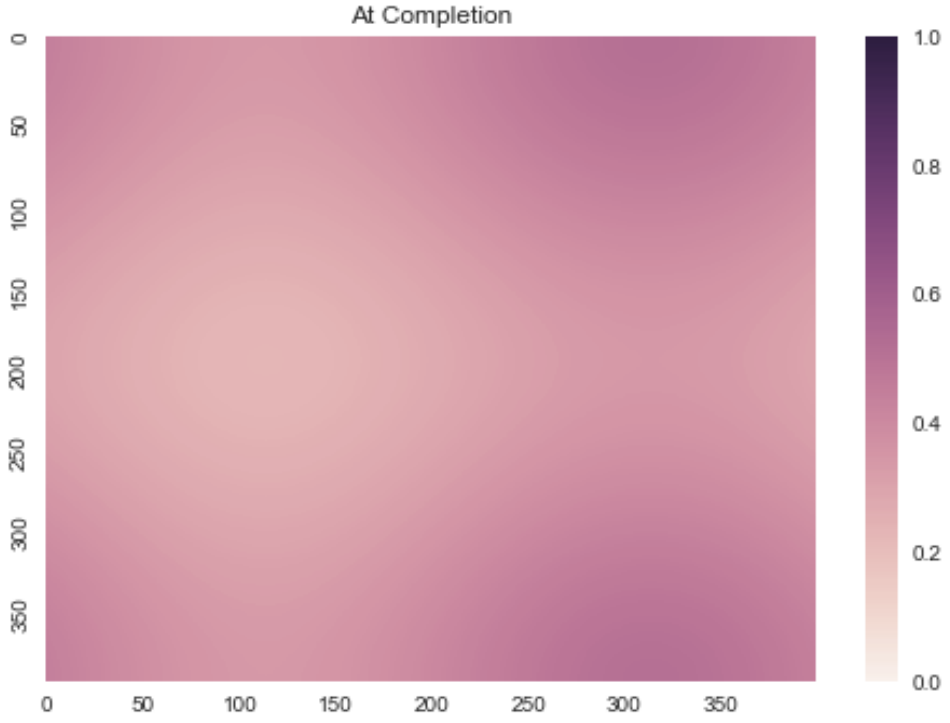
Data Plotting

I used the REPL environment of Jupyter Notebook with Python and the numpy, matplotlib, and seaboard libraries to plot the data. The notebook file, “ps1_viz_serial.ipynb”, resides in the src_serial folder. For convenience, I have copied the code therein to “plotting.py” in the same directory, 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.




Halfway Point
0
50
100
150
200
250
300
350
1.0
0.8
0.6
0.4
0.2
0.0


At Completion
0
50
100
150
200
250
300
350
1.0
0.8
0.6
0.4
0.2
0.0

Scheduler Performance Testing

Test were run with the following command:

./advection_parallel N 20000 1.0 1.0e6 5.0e-7 2.85e-7 28

Schedule Testing Data (Table 1)

Schedule	Chunk Size	Runtime N=2000 (s)	Runtime N=200 (s)
Static	Default (n/28)	61.41	0.98
Static	n/100	68.52	1.22
Static	n/200	69.09	1.23
Dynamic	Default (1)	79.78	1.13
Dynamic	n/28	70.17	1.08
Dynamic	n/100	68.73	1.15
Guided	Default	62.44	1.10
Guided	n/100	68.79	1.09
Guided	n/200	64.86	1.08

where *N* was represented the dimensions of the matrix. Data is presented in the table below.

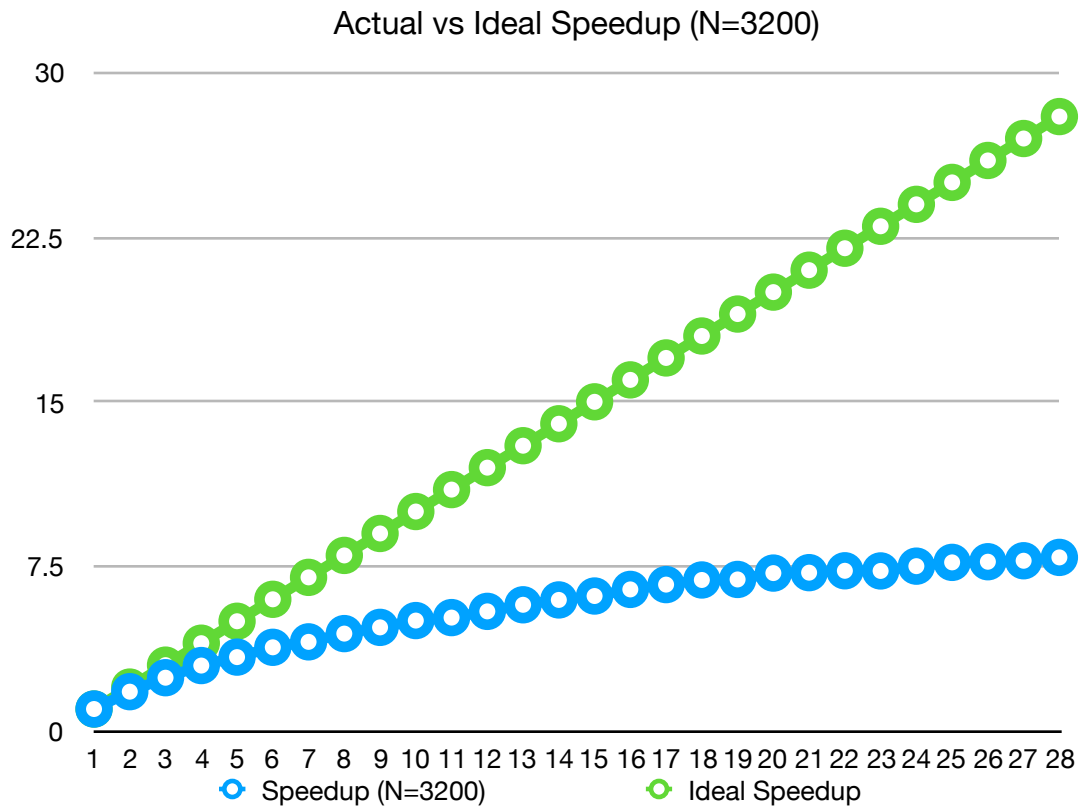
In both instances, the default version of static scheduling won out. Given the highly parallelizable nature of this problem and the extremely limited load imbalancing, this is the expected result. Guided also performed well, likely because the majority of the parallel regions were handled on the initial disbursement. Dynamic performed the worst overall, likely because there was little performance gain (that might have been generated by imbalanced work) to accommodate the overhead of handling the threads.

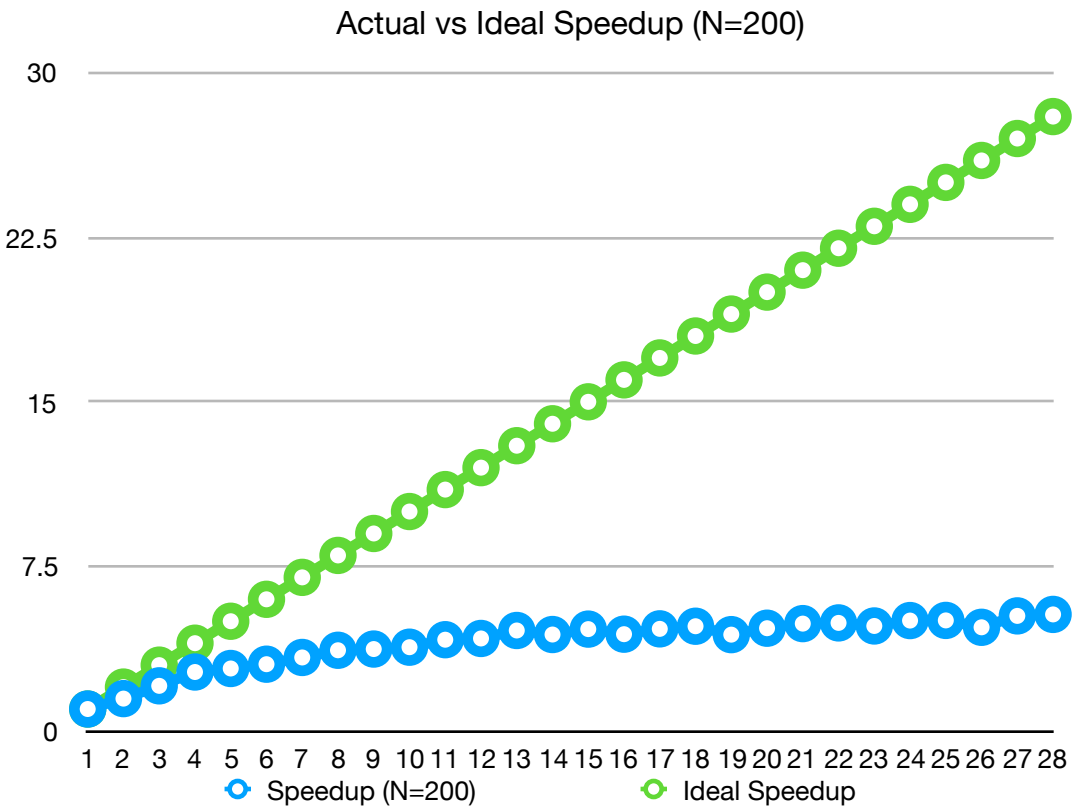
Strong Scaling Study

Data from the tests, as well as plots of the measured speedup versus ideal against the number of cores, are below.

Strong Scaling Study (Table 2)

Cores	Ideal Speedup	Time (N=3200) (s)	Speedup (N=3200)	Ideal Time (N=3200)	Time (N=200) (s)	Speedup (N=200)	Ideal Time (N=200)
1	1	80.44	1.00	80.44	0.3718	1.00	0.3718
2	2	44.94	1.79	40.22	0.2522	1.47	0.1859
3	3	33.15	2.43	26.81	0.1808	2.06	0.1239
4	4	26.99	2.98	20.11	0.1384	2.69	0.0930
5	5	23.90	3.37	16.09	0.1306	2.85	0.0744
6	6	21.10	3.81	13.41	0.1222	3.04	0.0620
7	7	19.81	4.06	11.49	0.1110	3.35	0.0531
8	8	18.13	4.44	10.06	0.1011	3.68	0.0465
9	9	17.04	4.72	8.94	0.0996	3.73	0.0413
10	10	15.99	5.03	8.04	0.0974	3.82	0.0372
11	11	15.57	5.17	7.31	0.0894	4.16	0.0338
12	12	14.76	5.45	6.70	0.0882	4.22	0.0310
13	13	13.99	5.75	6.19	0.0812	4.58	0.0286
14	14	13.47	5.97	5.75	0.0846	4.39	0.0266
15	15	13.08	6.15	5.36	0.0801	4.64	0.0248
16	16	12.47	6.45	5.03	0.0843	4.41	0.0232
17	17	12.06	6.67	4.73	0.0799	4.65	0.0219
18	18	11.69	6.88	4.47	0.0779	4.77	0.0207
19	19	11.65	6.90	4.23	0.0847	4.39	0.0196
20	20	11.18	7.19	4.02	0.0794	4.68	0.0186
21	21	11.15	7.21	3.83	0.0760	4.89	0.0177
22	22	11.02	7.30	3.66	0.0755	4.92	0.0169
23	23	11.03	7.29	3.50	0.0777	4.79	0.0162
24	24	10.71	7.51	3.35	0.0739	5.03	0.0155
25	25	10.47	7.68	3.22	0.0738	5.04	0.0149
26	26	10.43	7.71	3.09	0.0787	4.72	0.0143
27	27	10.37	7.76	2.98	0.0709	5.24	0.0138
28	28	10.17	7.91	2.87	0.0700	5.31	0.0133


Actual vs Ideal Speedup (N=3200)
30
22.5
15
7.5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Speedup (N=3200)
Ideal Speedup

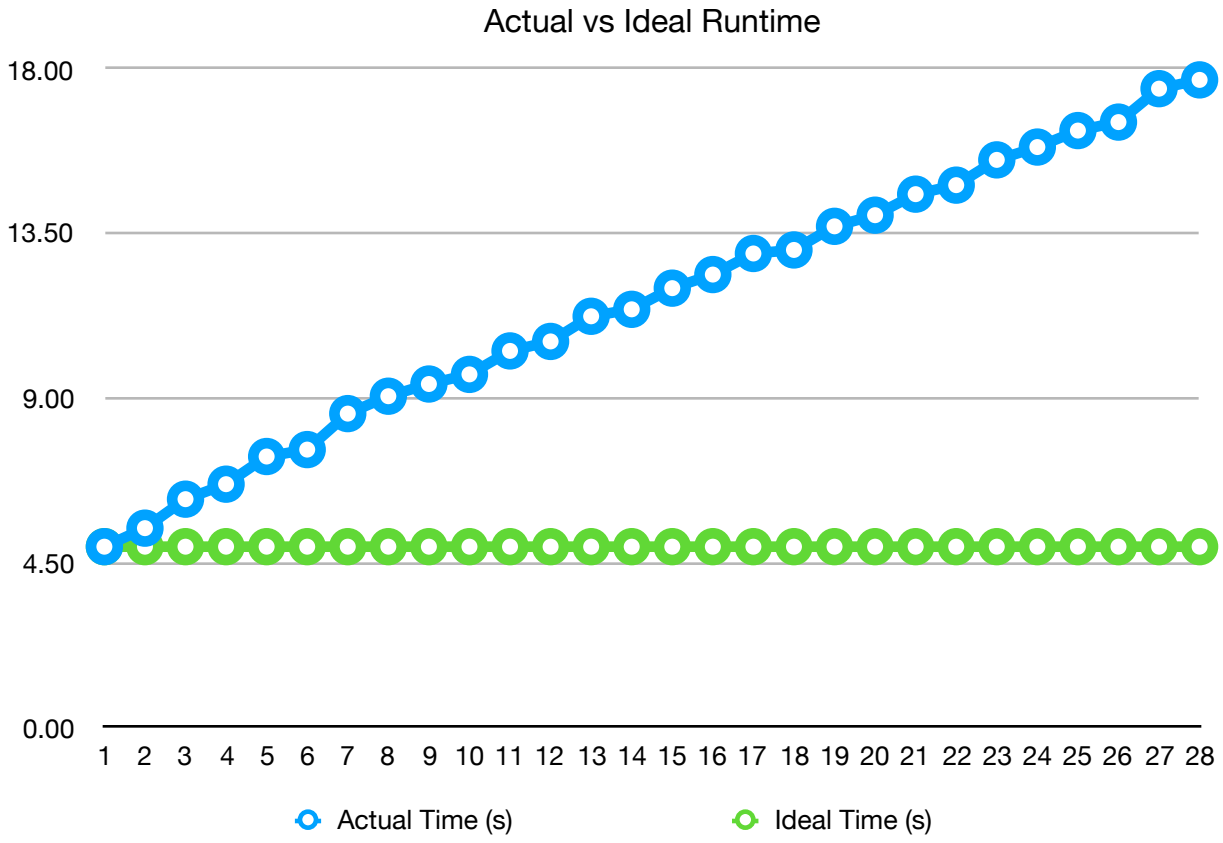

Actual vs Ideal Speedup (N=200)
30
22.5
15
7.5
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Speedup (N=200)
Ideal Speedup

The performance improvement, while significant, falls far short of the ideal improvement. At 28 cores, the N=3200 test experienced nearly 8x improvement. If only 10% of the serial runtime is unparallelizable, this is close to an optimal result. Given that neither the writes to file nor the timesteps are parallelized, this result is expected.

Interestingly—and unsurprisingly—the strong scaling study on the smaller sample size is unable to achieve the same speedup factor over 28 cores. I speculate that this is a result of the overhead of threading weighing more on a smaller test space.

Weak Scaling Study

Data from the tests, as well as a plot of the ideal time versus actual against the number of cores, are below.



The weak scaling study shows a linear increase at a factor of approximately N/4 for sample size N. As with the strong scaling study, I speculate the added cost is a consequence of the non-parallelizable regions of the code and the overhead of managing the threads. I imagine that as the scale continued up, we would see the relationship change from linear to logarithmic. That said, while I believe the code is functioning well, this result is somewhat disappointing, and I would be interested in learning what more could be done to optimize the weak scaling for this code.

Weak Scaling Study (Table 3)

Cores	Matrix Dimension (N)	Matrix Size (N^2)	Time (s)	Ideal Time (s)
1	800	640000	4.94	4.94
2	1131	1280000	5.45	4.94
3	1386	1920000	6.24	4.94
4	1600	2560000	6.65	4.94
5	1789	3200000	7.40	4.94
6	1960	3840000	7.59	4.94
7	2117	4480000	8.58	4.94
8	2263	5120000	9.05	4.94
9	2400	5760000	9.39	4.94
10	2530	6400000	9.65	4.94
11	2653	7040000	10.29	4.94
12	2771	7680000	10.55	4.94
13	2884	8320000	11.24	4.94
14	2993	8960000	11.43	4.94
15	3098	9600000	12.01	4.94
16	3200	10240000	12.38	4.94
17	3298	10880000	12.96	4.94
18	3394	11520000	13.05	4.94
19	3487	12160000	13.70	4.94
20	3578	12800000	14.00	4.94
21	3666	13440000	14.58	4.94
22	3752	14080000	14.82	4.94
23	3837	14720000	15.52	4.94
24	3919	15360000	15.87	4.94
25	4000	16000000	16.32	4.94
26	4079	16640000	16.55	4.94
27	4157	17280000	17.46	4.94
28	4233	17920000	17.70	4.94