High Performance Computing

MPCS 51087

Problem Set 3

Dynamic Load Balancing

Spencer Zepelin

August 5, 2019

Compiling and Running the Code

The programs were 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 files “latencybandwidth.c”, “julia_serial.c”, and “julia_modes.c”. A makefile has been provided to assist in compiling. Running the command “make” will make the executables named “julia_serial” and “julia_modes" by running the following commands:

gcc -std=c99 -o julia_serial julia_serial.c
mpicc -std=c99 -o julia_modes julia_modes.c

Running the command “make trials” will make the “latencybandwidth” executable for the latency and bandwidth tests by running the following commands:

mpicc -std=c99 -o latencybandwidth latencybandwidth.c

Calling “make mpi” will compile the Julia executable and then run it in both modes with MPI enabled with the following default commands:

mpiexec -n 2 ./julia_modes static 10000
mpiexec -n 2 ./julia_modes dynamic 10000 10000

In both cases, the command should be of the form:

mpiexec -n RANKS ./julia_modes RUNTYPE GRID-DIMENSION (CHUNKSIZE)

Where CHUNKSIZE is only defined if running in dynamic mode.

“make serial” will run the serial version of the code with the following default:

./julia_serial 1000

Calling “make latband” will ensure the latency and bandwidth trials have been compiled and then run it serially. No arguments are necessary for the latency and bandwidth tests since all

parameters are hardcoded. Similarly, the Julia set parameters have been hardcoded in the relevant C files. The code could easily be adapted to support parameterization if necessary.

Data from latency and bandwidth tests are printed to the screen. Data from the serial code will be output in binary to “data.bin”. Data from the MPI Julia code will be output to either “data_mpi_static.bin” or “data_mpi_dynamic.bin” depending on the mode in which the code is run. 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 both modes. After first initializing the MPI conditions, the code moves into one of the two code branches. In static mode, the code distributes the grid cells roughly evenly to all ranks, performs the calculations, and use collective IO to output the data.

In dynamic mode, the boss keeps a log of the latest chunk not yet assigned, the chunks each worker is currently handling, and the number of workers still running. The boss assigns initial chunks by sending integer values for the array indices to each of the workers. The workers then perform their calculations, return their results to the boss, and await their next assignment. Once they receive an assignment beyond the boundaries of the grid, they know to stop working. During this time, the boss receives data from any worker, immediately gives that worker its next assignment, and subsequently writes the received data to file. It uses its log to ensure it is writing to the correct part of the file and updates its log with the new assignment. When the boss sends out an assignment beyond the bounds of the array, it knows that worker is finished and decrements the count of active workers.

Stunningly, the entire Julia codebase does not require a single barrier. Each rank in the static decomposition requires memory resources on the order of the total problem size divided by the number of ranks. In dynamic load balancing, however, each rank only requires memory on the order of the chunk size. As the ratio of total problem size to ranks continues to increase, dynamic load balancing gives us more flexibility to ensure each rank can hold the necessary data in memory while static decomposition has a tighter upper limit.

Both the serial implementation and the latency and bandwidth testing follow the structures outlined in the prompt very closely and are of little design interest.

Latency

Average intra-node latency (100,000 runs): 0.851 microseconds
Average inter-node latency (100,000 runs): 86.568 microseconds

Based on my tests, inter-node latency is approximately two orders of magnitude greater than intra-node latency. The value of trying to keep communications on the same node is readily apparent.

Bandwidth

The table below shows the measured values for bandwidth in gigabytes per second between two ranks on the same node and on different nodes for different packet sizes. Intra-node bandwidth is just under 10x larger in the worst case. Interestingly, I recorded significant increases in intra-node bandwidth as packet size increased. I speculate that less amortization of MPI overhead for smaller packet sizes is largely to blame. We do not see the same magnitude of increase for inter-node message passing likely because the higher latency between nodes permits greater amortization even for smaller packets.

Bandwidth Study

	Intra-Node (average GB/s)	**Inter-Node (average GB/s)**
KB packets	0.961	0.102
MB packets	1.418	0.107
GB packets	3.942	0.109

MPI Data Plotting

The plots on the following page were generated with a 1000 x 1000 grid with the serial, statically decomposed, and load balancing code, respectively.

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

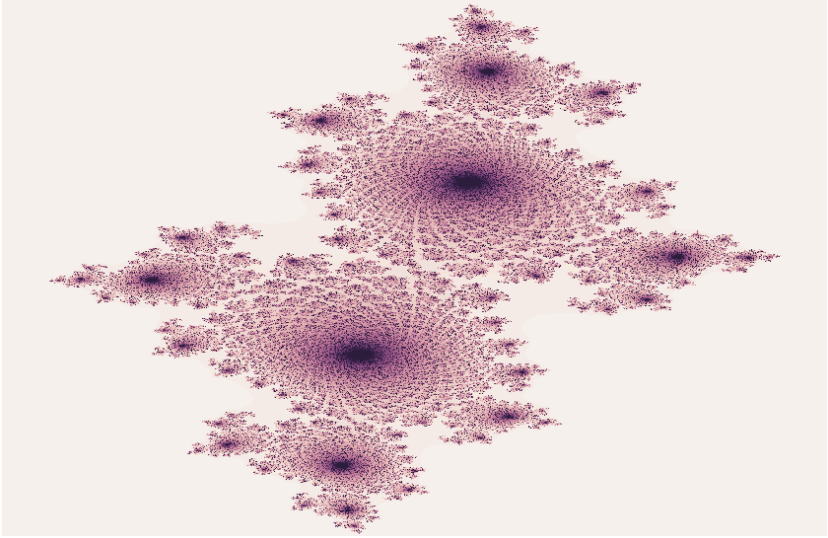
diff data.bin data_mpi_static.bin
diff data.bin data_mpi_dynamic.bin

The commands did, in fact, indicate the data files were identical.

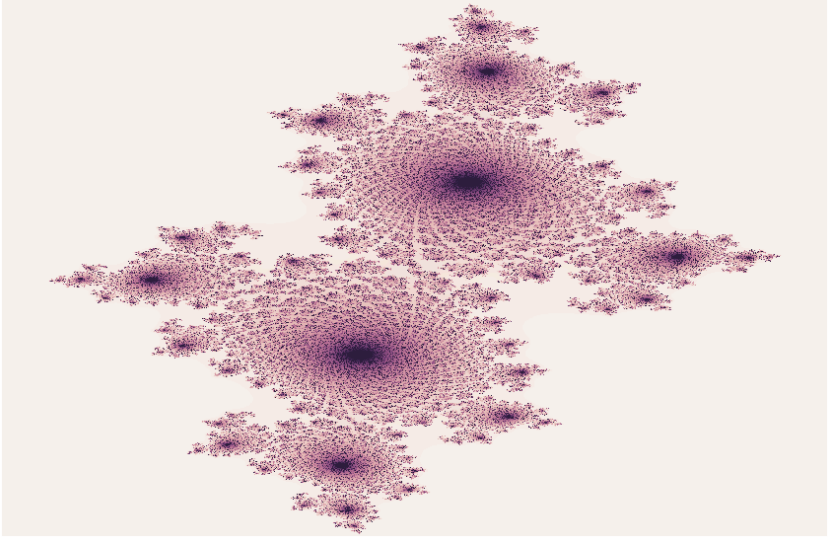
To plot the data, I used the REPL environment of Jupyter Notebook with Python and the numpy, matplotlib, and seaborn libraries. The notebook file is saved as “ps3_plotting.ipynb”. For convenience, I have copied the code therein to “plotting.py”, but the code has not been modified to save to file and, as such, should simply be used as a reference. Both files can be found in the “plotting” directory. Row zero, column zero is in the top left corner of the plots and the highest row and column in the bottom right.

Plots For Each Run Type

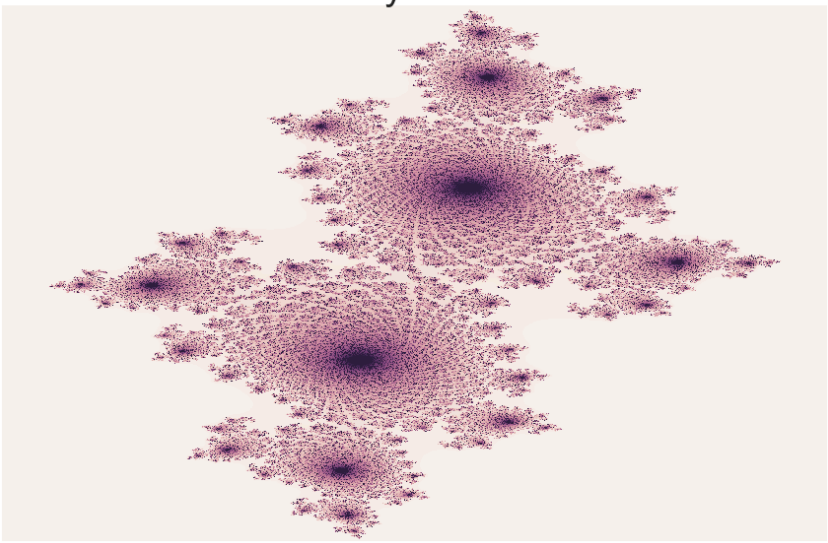
Julia - Serial



Julia - Static MPI

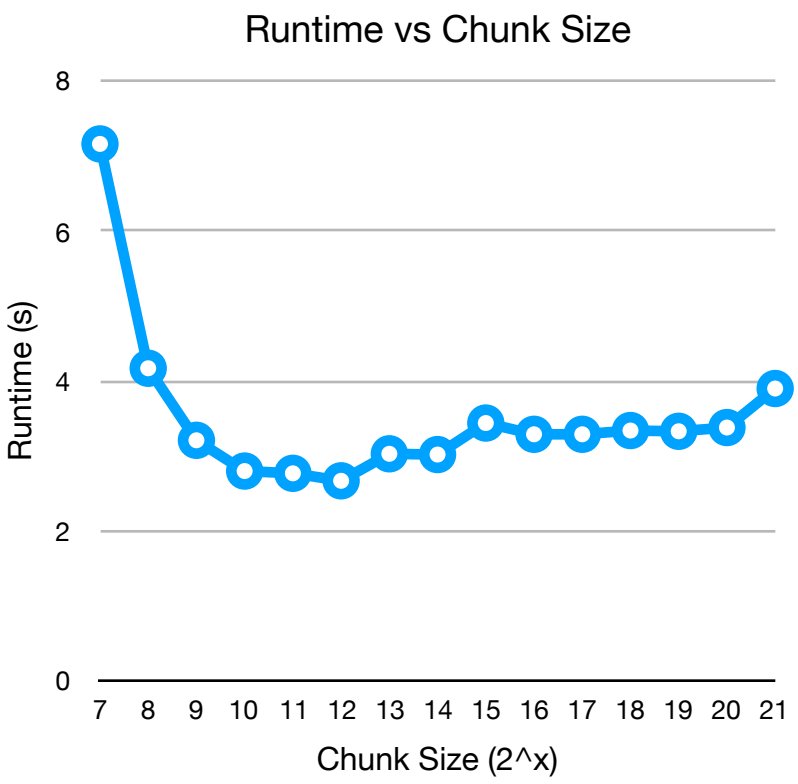


Julia - Dynamic MPI



Dynamic Load Balancing Optimal Chunk Size

The chart below and its corresponding data table show the results of the chunk size testing. Tests were run over a 12,096 x 12,096 grid on 56 ranks over two nodes. Powers of 2 from 7 to 21 were tested. As one might expect, the data show a sharp increase in performance as chunk size increases and then a worsening in performance as it gets larger. The optimal chunk size represents the best tradeoff between less message passing for larger chunks and more dynamism for smaller ones. As a result of this test, the strong scaling study was performed with a chunk size of 4,096 (2^{12}).



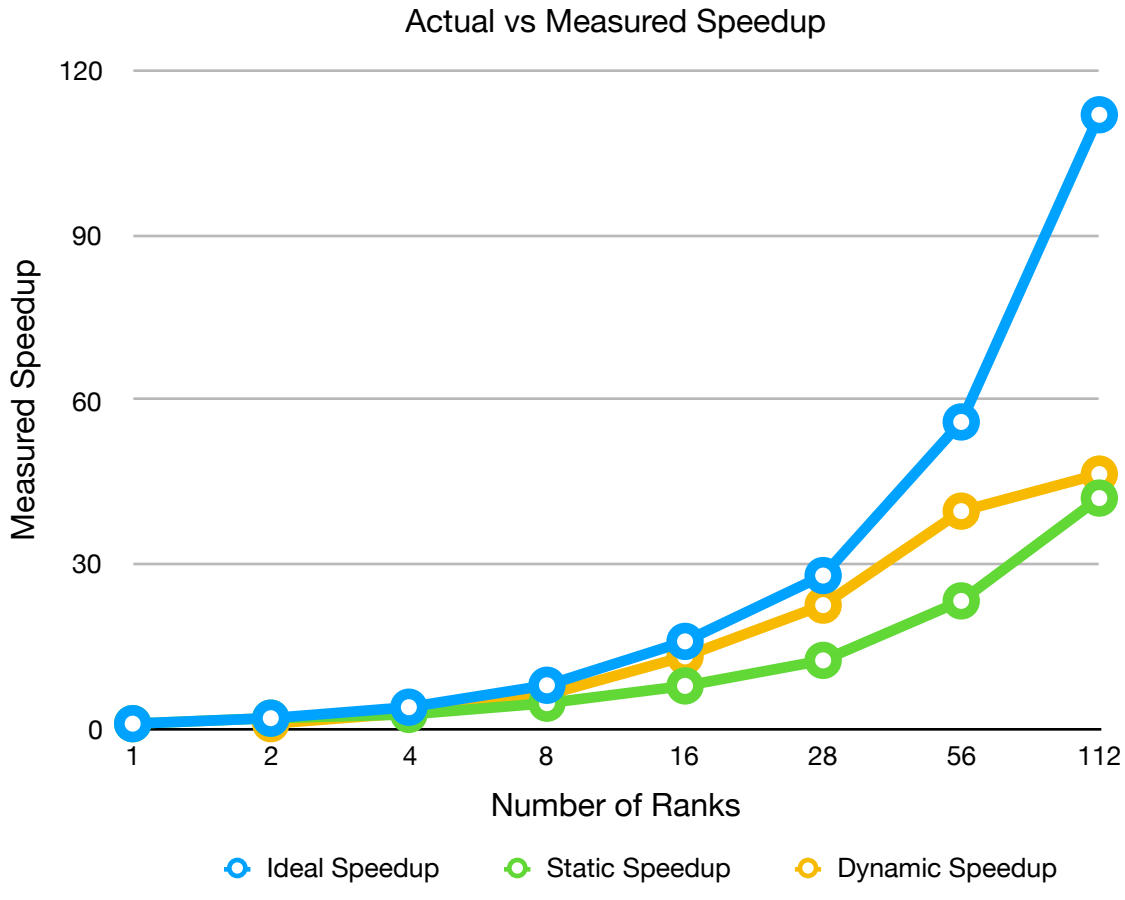
Chunk Size	$\log_2$ (Chunk Size)	Runtime (s)
128	7	7.16
256	8	4.17
512	9	3.21
1024	10	2.80
2048	11	2.77
4096	12	2.67
8192	13	3.03
16384	14	3.02
32768	15	3.44
65536	16	3.29
131072	17	3.29
262144	18	3.34
524288	19	3.33
1048576	20	3.38
2097152	21	3.90

Strong Scaling Study

Per the prompt, the tests were performed over a 12,096 x 12,096 grid, using a chunk size of 4,096 for the dynamic mode. Both modes were tested with 2, 4, 8, 16, 28, 56, and 112 processes. Static decomposition was also performed on a single rank. Dynamic load balancing could not be performed on a single rank given the boss-worker paradigm it employs. Data from the tests, as well as plots of the measured speedup versus ideal against the number of ranks, are below.

Given the overhead endured with MPI, the speedup in both cases is significant, albeit far from ideal. Both static decomposition and dynamic load balancing saw greater than 40x speedup at 112 ranks. Interestingly, the runtime for static decomposition on a single rank—the serial case—is nearly identical to that achieved by dynamic load balancing over two ranks, suggesting that the additional overhead endured by a boss with a single worker was minimal for intra-node communication.

As the number of ranks increased, dynamic load balancing quickly began to outperform static decomposition, permitting ranks to continue working throughout the execution rather than waiting on their peers to catch up. This result stands in stark contrast to static vs dynamic scheduling for OpenMP threads as measured in problem set 1. When each grid calculation takes roughly the same time to complete—as in problem set 1—the overhead incurred by dynamic scheduling costs performance. In this case, where grid cells can have markedly different calculation times, dynamic scheduling appears well worth the overhead. I imagine this result would hold were a scheme to apply dynamic load balancing through MPI to the advection problem implemented with it underperforming relative to domain decomposition.



Strong Scaling Study

Number of Ranks	Static Decomposition Runtime (s)	Dynamic Load Balancing Runtime (s)	Ideal Speedup	Static Speedup	Dynamic Speedup
1	114.57	-	1	1.00	-
2	57.28	114.85	2	2.00	1.00
4	42.64	38.99	4	2.69	2.95
8	24.20	18.34	8	4.73	6.26
16	14.53	8.66	16	7.89	13.26
28	9.14	5.08	28	12.54	22.61
56	4.90	2.89	56	23.38	39.74
112	2.72	2.47	112	42.12	46.50