Matrix Addition

Luke Brom

October 29, 2018

1 Strategy

1.1 Partitioning the Data

I decided to partition the data by individual elements in both matrix A and B. Each processor is responsible for E/P elements in each matrix where E is the total elements in the matrix and P is the number of processors. I choose this partitioning strategy because it is simple, and it guarantees near perfect load balancing. If I had partitioned by rows then the last processor might have to pick up a substantial amount of work (if the size of the rows are large), but with partitioning the data by element, the last processor will only have to pick up a minuscule amount of extra work if the dimensions are not divisible by P.

1.2 Constraints

I did not place any constraints on the dimensions of the matrices or the number of processors, except that the dimensions of matrix A and matrix B must match.

1.3 Generating Matrices

I decided to store my matrices in the following way:

```
3 3
1 2 3
1 2 3
1 2 3
```

The first two numbers are the rows and columns respectively, followed by the data with the rows separated by a newline. The program used to generate this is called MatrixGenerator, here is the output from the -h command.

```
usage: ./matrixGenerator [flags]
-h print help
-o <output file> set output file
-r <rows> number of rows to generate, defaults to 100:
-c <columns> number of columns to generate, defaults to 100:
-m <max> max integer to generate, defaults to 999:
```

You can make this program by calling "make MatrixGenerator".

1.4 Verification

In order to verify that the matrix addition was correct, I wrote my own program that takes in two matrices, does sequential matrix addition on them, then compares the result to a specified file. This program is called MatrixChecker, here is the output from the -h command.

```
usage: ./matrixChecker [flags]
-h print help
-a <input file A> input file for matrix A
```

-b <input file B> input file for matrix B
-o <output file> set output file
-c <compare file> file to compare matrix to

You can make this program by calling "make MatrixChecker".

1.5 Experimental Parameters

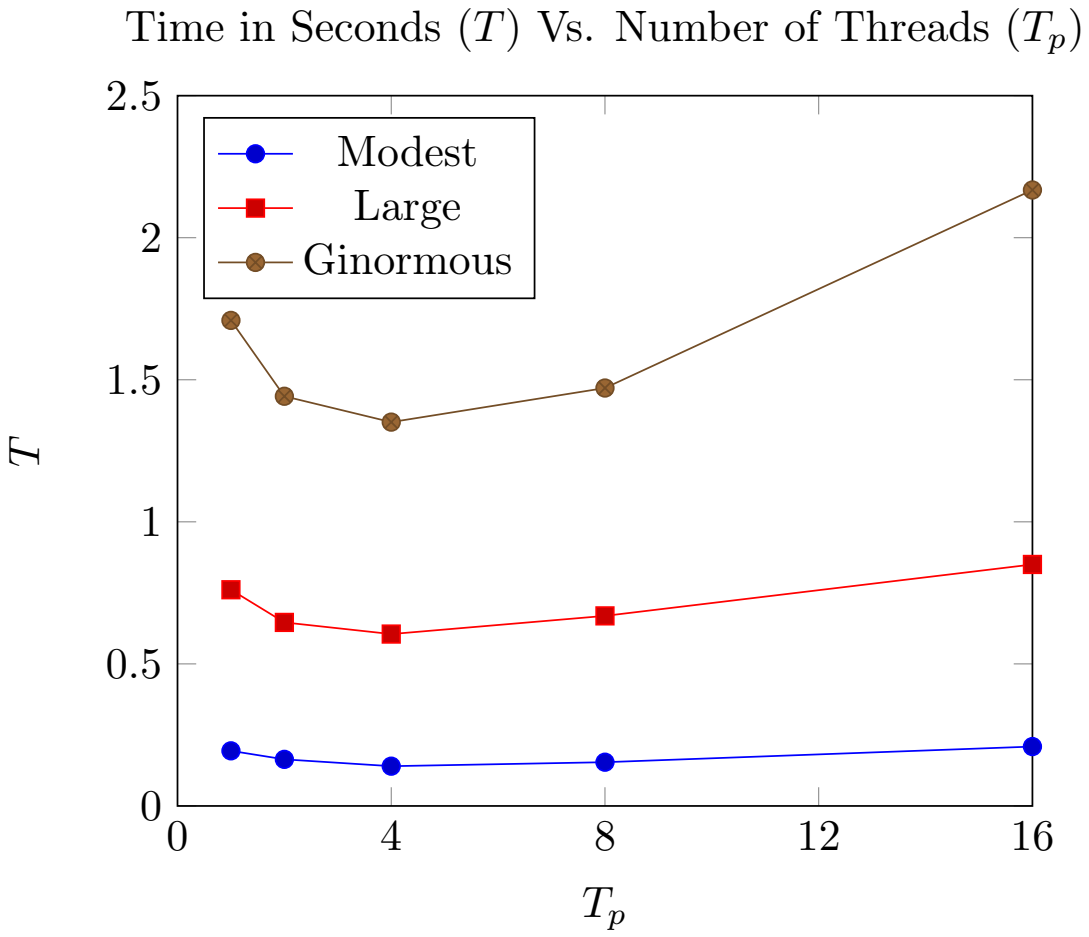
I have selected three matrix sizes as outlined below, my program will square each one of these:

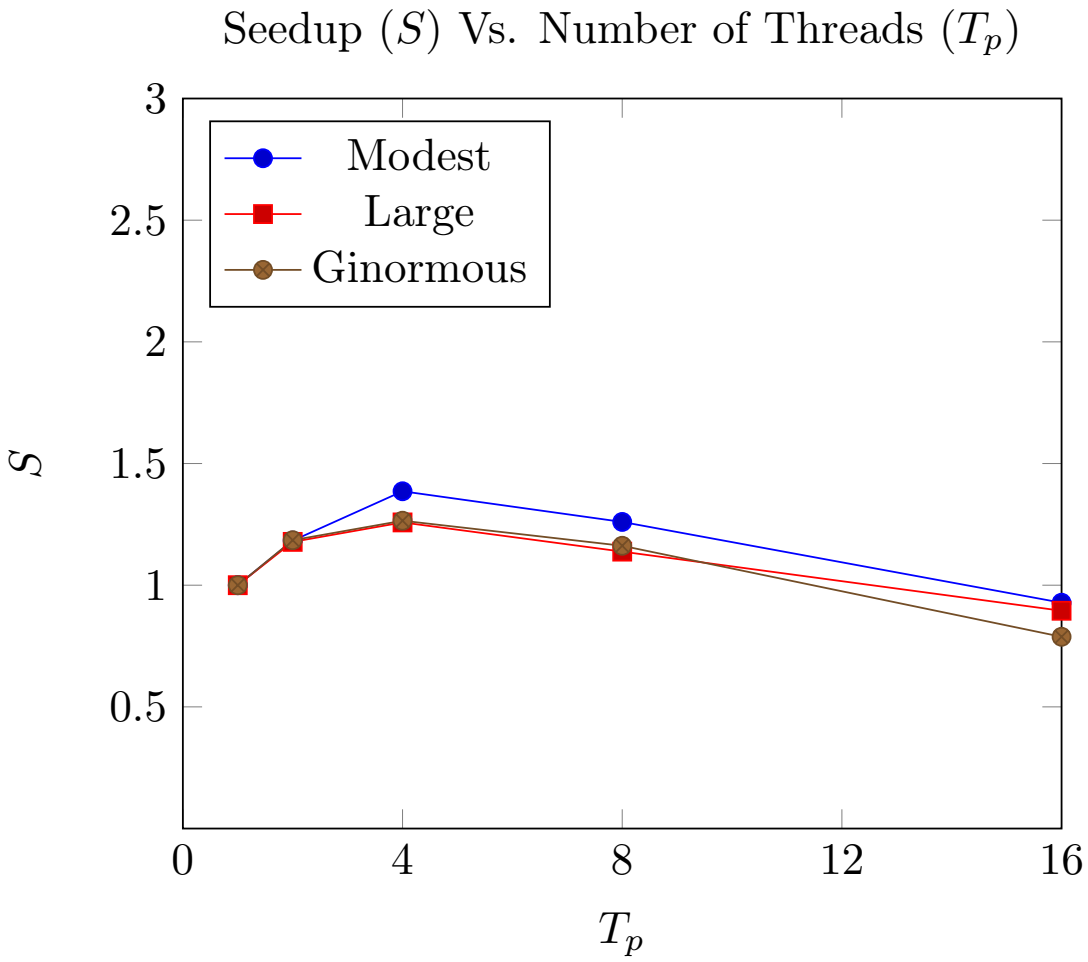
File Name	Dimensions
modest.txt	5000x5000
large.txt	10000x10000
ginormous.txt	15000x15000

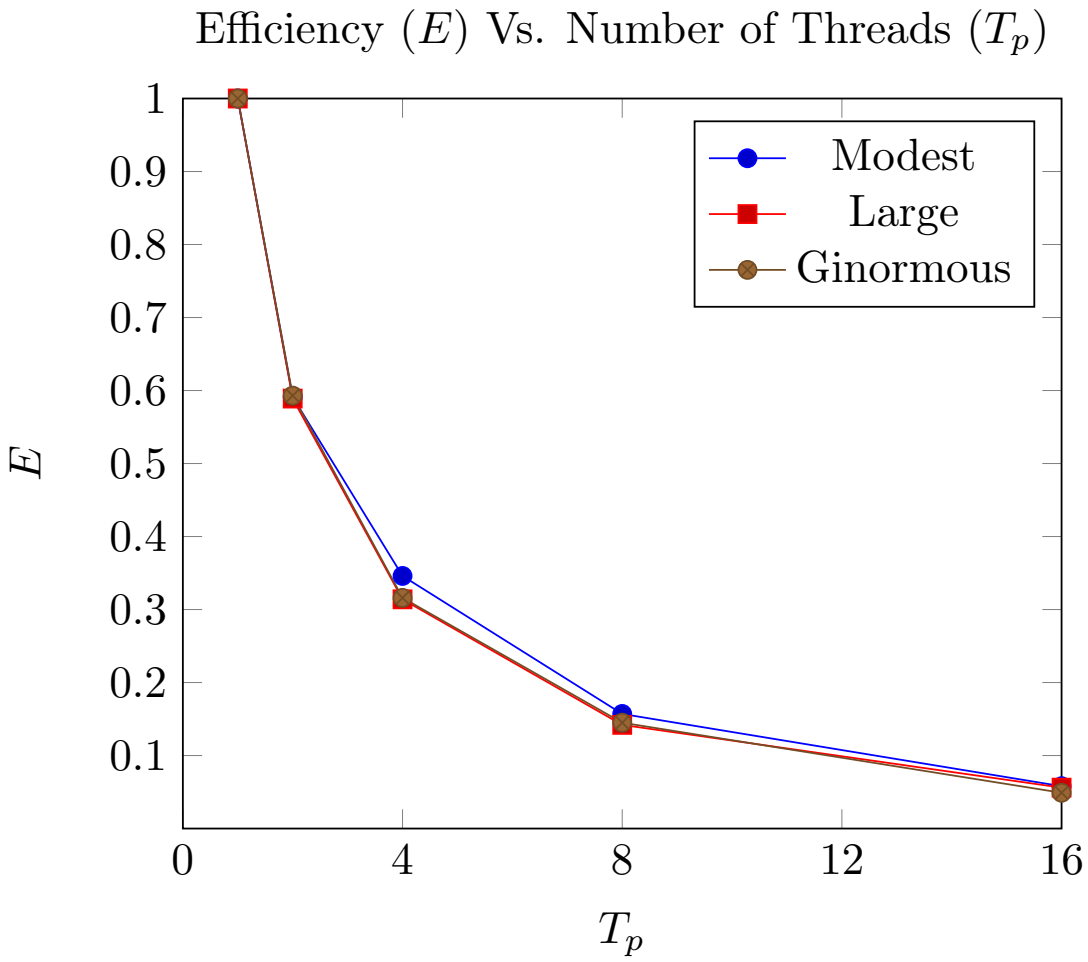
2 Data

Image	p	$T_p(s)$	s	e
5000x5000 + 5000x5000	1	0.194	1.000	1.000
	2	0.164	1.183	0.591
	4	0.140	1.386	0.346
	8	0.154	1.260	0.157
	16	0.209	0.928	0.058
10000x10000 + 10000x10000	1	0.761	1.000	1.000
	2	0.646	1.178	0.589
	4	0.605	1.258	0.314
	8	0.669	1.138	0.142
	16	0.850	0.895	0.056
15000x15000 + 15000x15000	1	1.709	1.000	1.000
	2	1.442	1.185	0.593
	4	1.351	1.265	0.316
	8	1.471	1.162	0.145
	16	2.168	0.788	0.049

3 Graphs


Time in Seconds (T) Vs. Number of Threads (T_p)
Modest
Large
Ginormous
2.5
2
1.5
1
0.5
0
T
0
4
8
12
16
T_p


Seedup (S) Vs. Number of Threads (T_p)
Modest
Large
Ginormous
S
3
2.5
2
1.5
1
0.5
0
4
8
12
16
T_p



4 Analysis

The data follows the expected pattern for every size matrix, with a speedup until around 4 processors followed by a gradual decrease in speedup. The one thing that is rather different is the fact that the speedup is really low for all matrix sizes. There wasn't a single run from 1 to 2 cores where the time halved as in previous assignments. My guess for why this is is that this algorithm is linear, which is pretty fast (especially in the context of the matrix multiplication in the last assignment). Since the sequential program can already add a large matrix pretty fast, the overhead of partitioning the work outweighs the benefits of running multiple processors by quite a bit, thus limiting the amount of speedup. In summary I would say that this problem would have to be very very big in order to justify parallelizing this algorithm.

After doing matrix multiplication this project really wasn't that hard. I spent most of my time on this project abstracting out functions and cleaning up my code.