Matrix Multiplication

Luke Brom

October 15, 2018

1 Strategy

1.1 Partitioning the Data

I decided to do a 1D partition of the data. Matrix A will be partitioned by row, while matrix B will be partitioned by column. While each processor will only have to see a certain subset of rows in A, each processor will have to see all of B. This model restricts the size of the matrix that can be processed, because each processor will have to allocate space for a large section of both A and B, however, the amount of communication is lower than that of a 2D partitioning. Since I am only working with a small data-set (I don't want to be up for hours running tests), 1D partitioning seems like a good choice since the size of the matrices won't be a problem.

1.2 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".

1.3 Verification

In order to verify that the matrix multiplication was correct, I wrote my own program that takes in two matrices, does sequential matrix multiplication 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".

1.4 Experimental Parameters

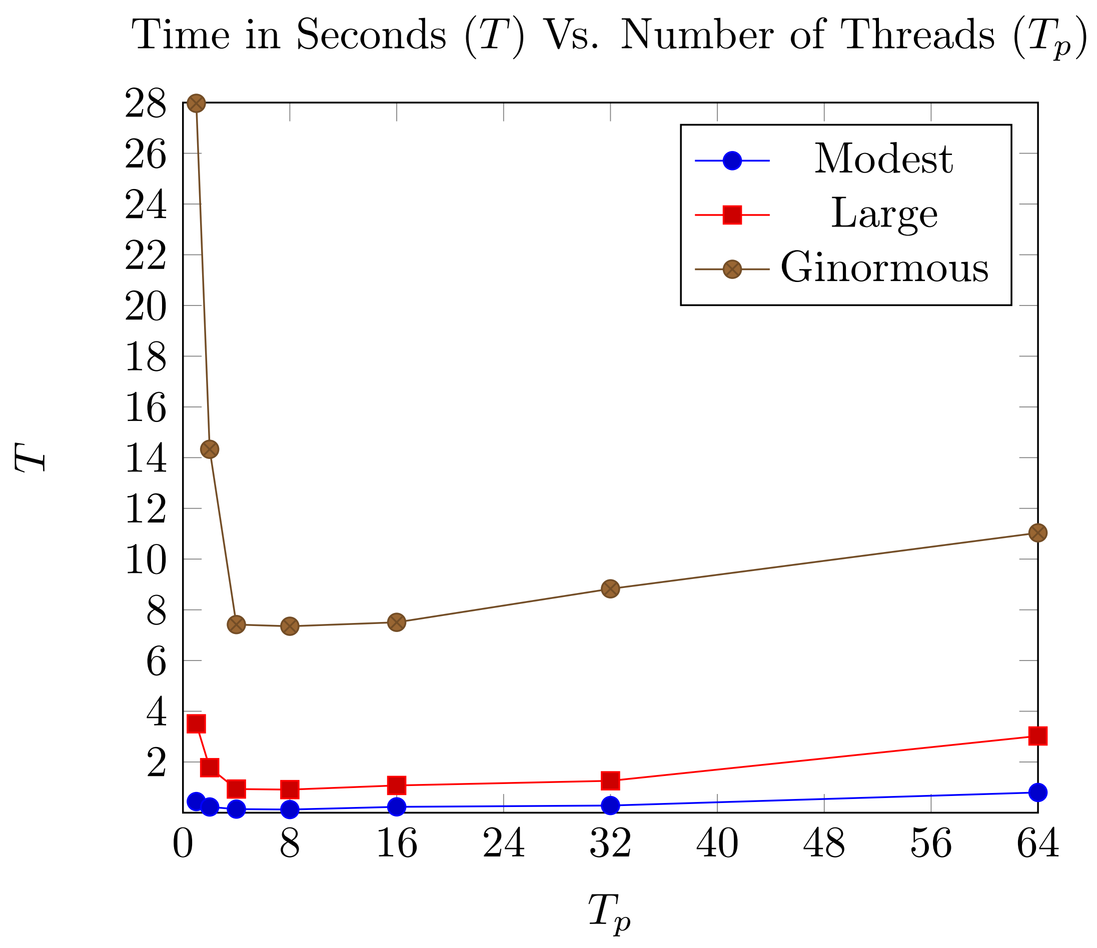
I have selected three matrix sizes as outlined below, my program will square each one of these:

File Name	Dimensions
modest.txt	500x500
large.txt	1000x1000
ginormous.txt	2000x2000

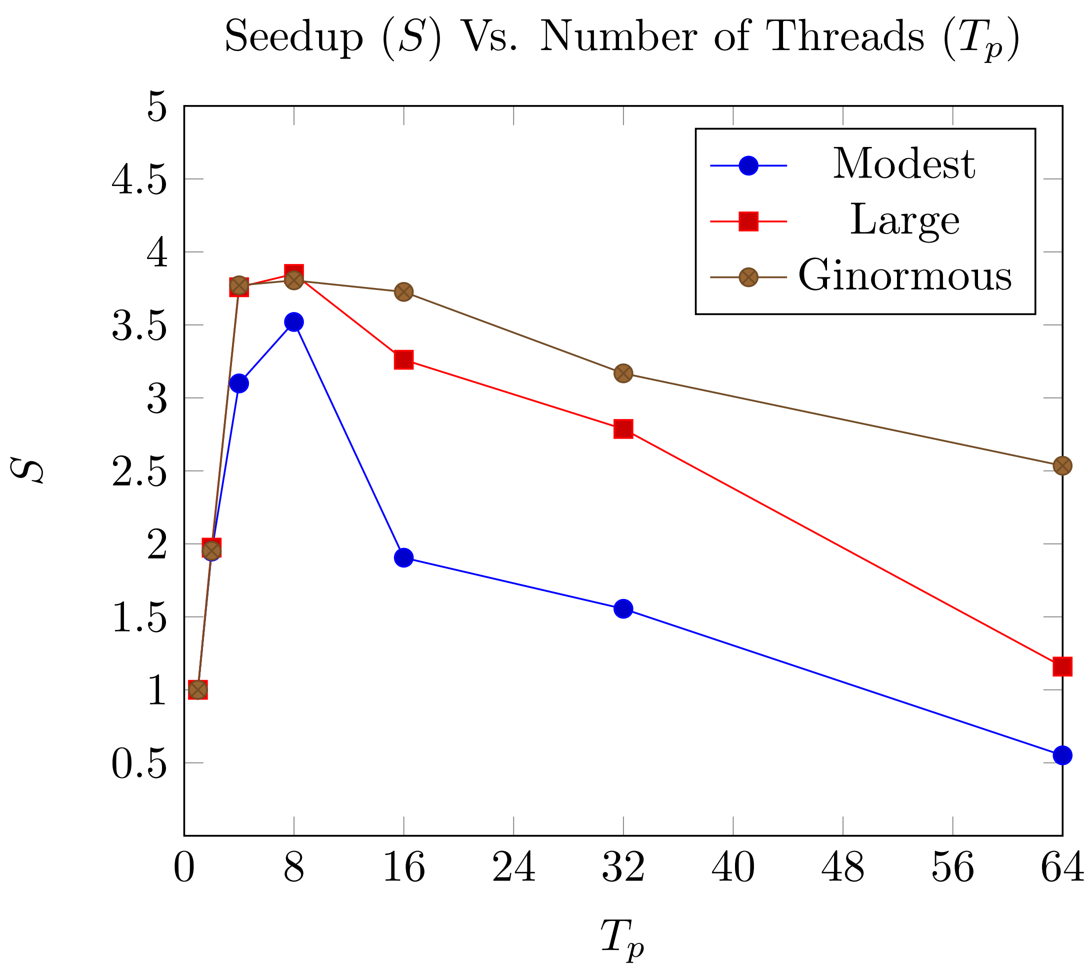
2 Data

Image	p	$T_p(s)$	s	e
500x500 * 500x500	1	0.440	1.000	1.000
	2	0.226	1.947	0.973
	4	0.142	3.099	0.775
	8	0.125	3.520	0.440
	16	0.231	1.905	0.119
	32	0.283	1.555	0.049
	64	0.798	0.551	0.009
1000x1000 * 1000x1000	1	3.506	1.000	1.000
	2	1.776	1.974	0.987
	4	0.933	3.758	0.939
	8	0.911	3.849	0.481
	16	1.075	3.261	0.204
	32	1.258	2.787	0.087
	64	3.025	1.159	0.018
2000x2000 * 2000x2000	1	27.971	1.000	1.000
	2	14.329	1.952	0.976
	4	7.418	3.771	0.943
	8	7.352	3.805	0.476
	16	7.507	3.726	0.233
	32	8.828	3.168	0.099
	64	11.035	2.535	0.040

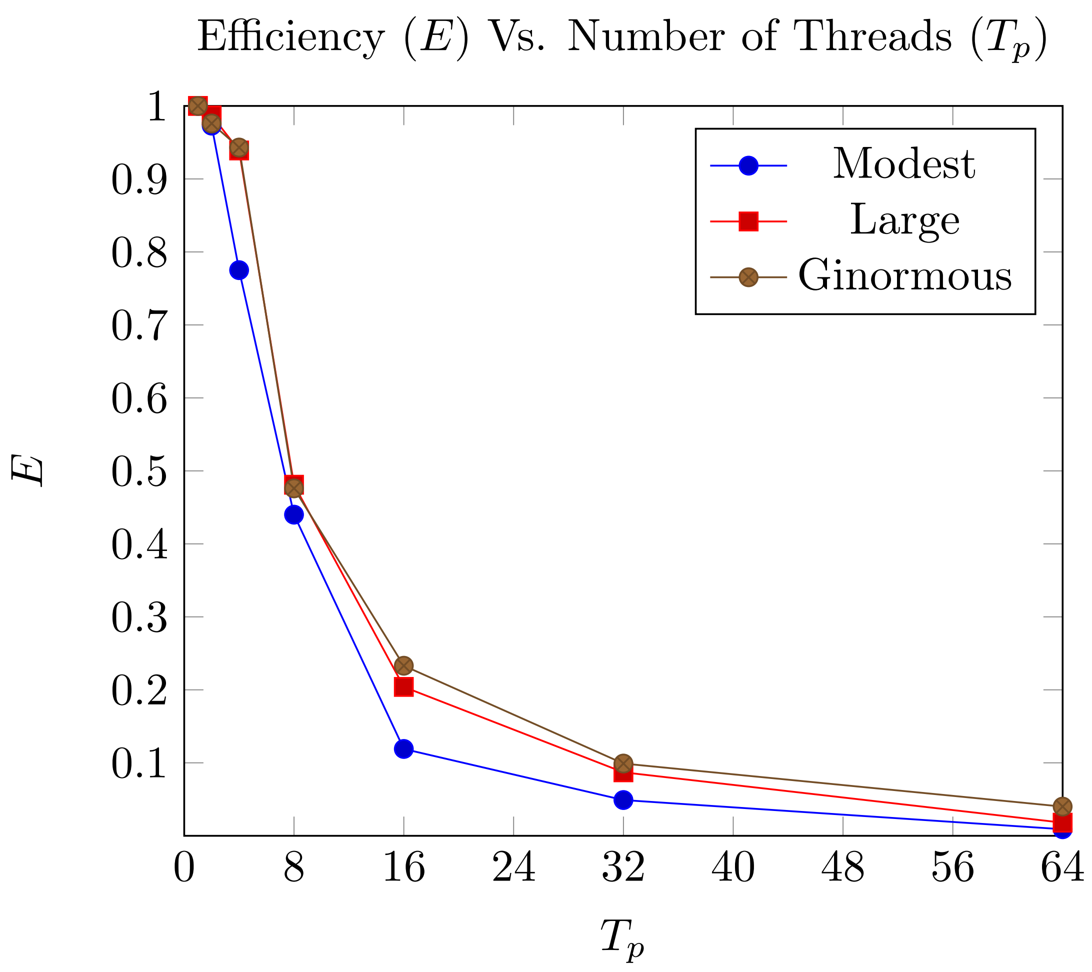
3 Graphs



Time in Seconds (T) Vs. Number of Threads (T_p)
28
26
24
22
20
18
16
14
12
10
8
6
4
2
T
0
8
16
24
32
40
48
56
64
T_p
Modest
Large
Ginormous



Seedup (S) Vs. Number of Threads (T_p)
Modest
Large
Ginormous
S
T_p
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
8
16
24
32
40
48
56
64



4 Analysis

The performance for each data-set followed the expected pattern. There was a speedup of around 2 with 2 processors, then as processors increase, speedup also increased until around 8 processors. After 8 processors, the speedup starts to decline again. There really weren't any deviations from the expected pattern in this experiment.

The hardest part of this assignment was the communication between processors. I particularly had trouble trying to figure out when and how to use blocking and non-blocking communication. I also ran into a problem where MPI optimizes its operations for small matrices, making the original iteration of my problem only work for small data-sets. In the interest of simplicity, I had only tested my program with a very small matrix, which MPI takes and buffer sends under the hood, meaning that things like communication to self worked just fine. However, when the data got bigger and MPI no longer buffer sent, my program stopped working. Then I had to debug a program that worked for small matrices but not large ones, which proved a little difficult.