Multiplication Tables in MPI Programming

Parallel Programming

CP 431

Professor Kotsireas

December 7, 2018

Joshua Nguyen

Tyler Allen

Jose Romero

Purpose

The purpose of the Multiplication Tables in MPI Programming is to count the number of different values in the multiplication table. This process of counting includes finding repeated numbers but not counting their occurrence in the final value.

Hints

We know that anything presented on one side of the diagonal line (the squares i.e. 1^2, 2^2, 3^2…N^2) of the multiplication table, will also be duplicated on the other side of the diagonal by mathematical nature (The table is symmetric). The N x N multiplication table only requires one side of the multiplication table in addition to the diagonal itself.

$$\text{Total Values} = \frac{N\ x\ N - N}{2} + N$$

The numbers in the observable multiplication table is bound to repeat unless it is a unique number. This is the catalyst to solving our problem.

We define the following function to quantify the repetition:

M(N) = # of different elements in an N x N multiplication matrix

To Do

Verify by hand:

MULTIPLICATION TABLE

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2		4	6	8	10	12	14	16	18	20
3			9	12	15	18	21	24	27	30
4				16	20	24	28	32	36	40
5					25	30	35	40	45	50
6						36	42	48	54	60
7							49	56	63	70
8								64	72	80
9									81	90
10										100

Let's start with the main table

M (5) = 14 white squares = normal countable squares

blue squares = the N^2 diagonal

green squares = border

Count the white and blue squares only

	1	2	3	4	5
1	1	2	3	4	5
2		4	6	8	10
3			9	12	15
4				16	20
5					25

M(10) = 42 *Count the white and blue squares only*

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2		4	6	8	10	12	14	16	18	20
3			9	12	15	18	21	24	27	30
4				16	20	24	28	32	36	40
5					25	30	35	40	45	50
6						36	42	48	54	60
7							49	56	63	70
8								64	72	80
9									81	90
10										100

Load Balancing

To effectively distribute the workload over *p* processors, we divided the total values by *p* to get an even distribution. We then took the total values modulo *p* to determine the remainder, which added 1 extra value to each processor starting from 0 until the remainder was exhausted.

1	2	3	4	5	6
	4	6	8	10	12
		9	12	15	18
			16	20	24
				25	30
					36

For example: Using 4 processors on a 6 x 6 table of 21 numbers, we divide the number of elements by the number of processors (4):

Minimum amount of numbers in a processor = 21/4

= 5 Remainder 1

As pictured: Blue (Processor 0) = 6 Elements (5 minimum + 1 remainder)

Orange (Processor 1) = 5 Elements

Green (Processor 2) = 5 Elements

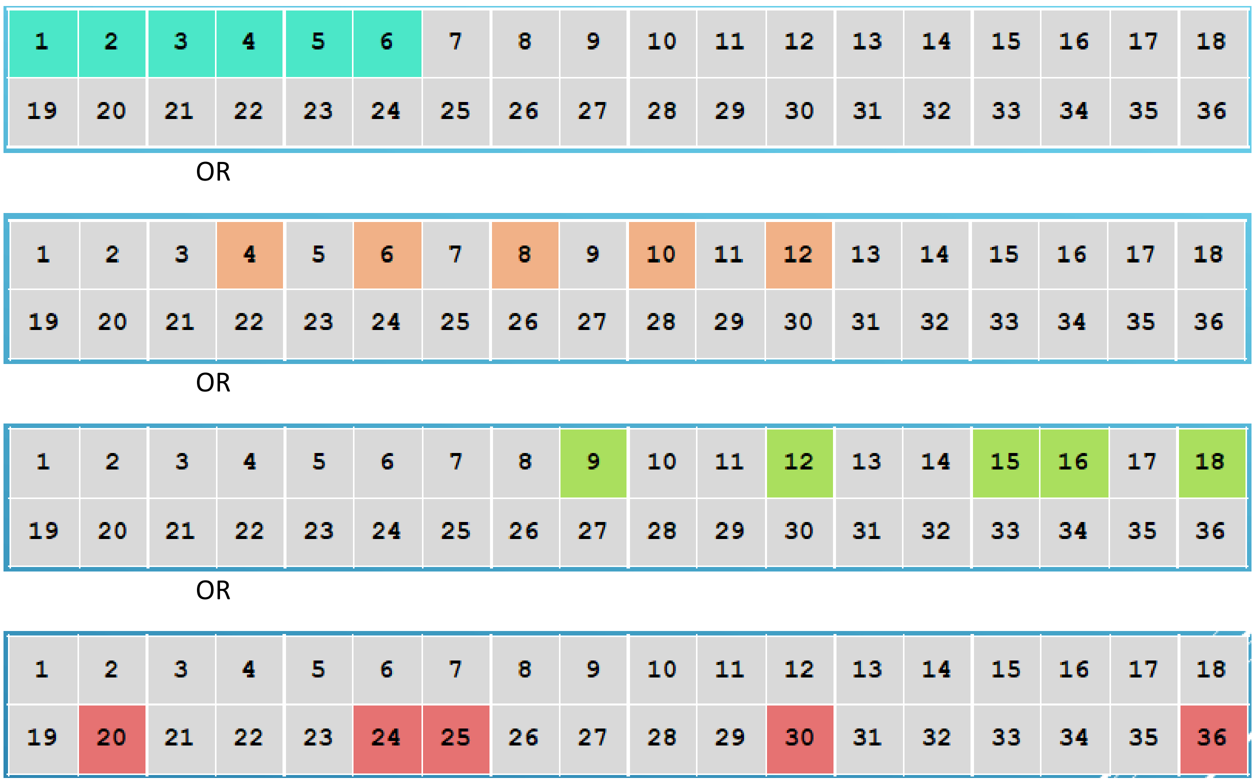
Red (Processor 3) = 5 Elements

Now that the math has been figured out, the load balancing algorithm begins to act. Using an *i by j* form, the load balancing algorithm assigns each chunk the correct number of elements that the processor will be working on.

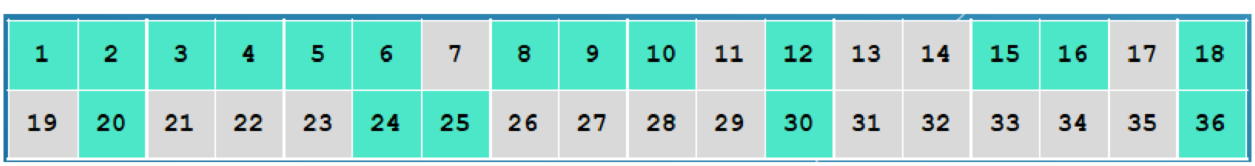
Design an efficient parallel algorithm to compute the function M(N) for values of N up to 10^5.

See Deliverables for the source code.

There is a master array (controlled by process 0) in which each processor will deliver the numbers they calculate to the master array. Numbers already delivered to the master processor will simply use the OR function (used in De Morgan's Law) to verify that the number exists. This method is called the "OR method".



=Blue (Processor 0) OR Orange (Processor 1) OR Green (Processor 2) OR Red (Processor 3) = Processor 0

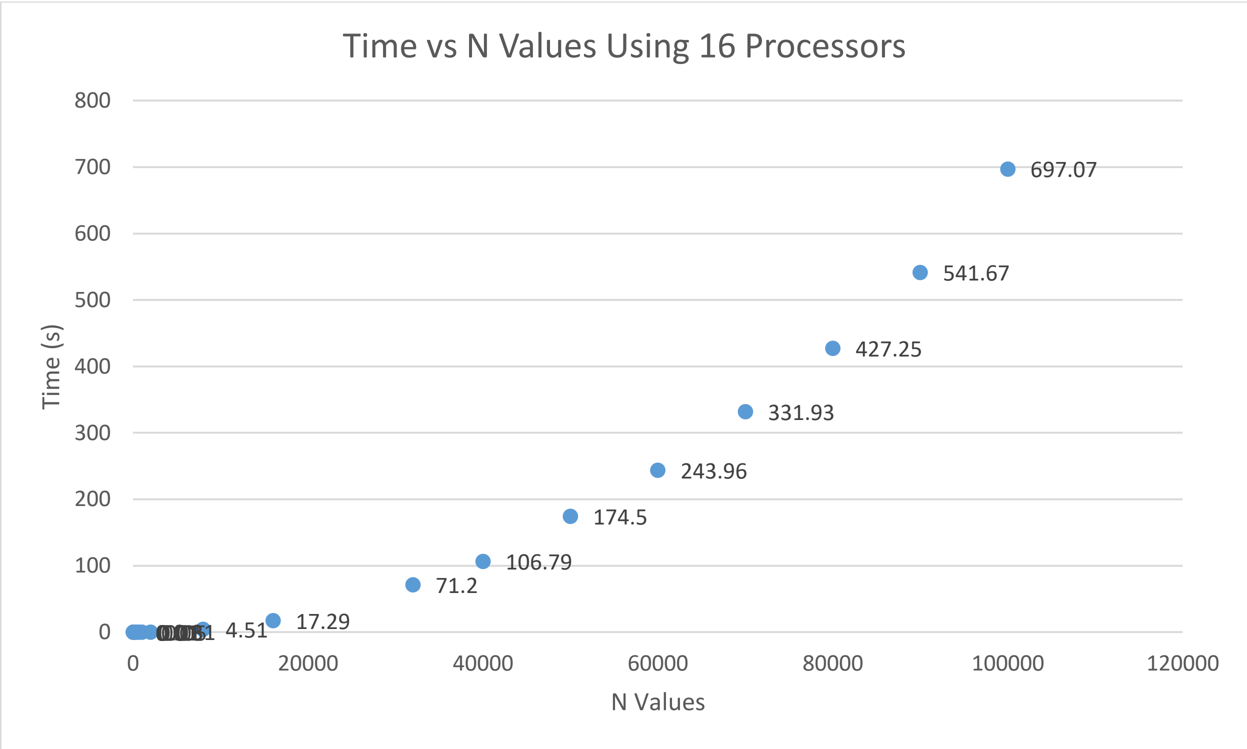


Results on 16 processors

X	M(x)	Time (seconds)
5	14	0.01
10	42	0.01
20	152	0.01
40	517	0.01
80	1939	0.00
160	7174	0.01
320	27354	0.02
640	103966	0.03
1000	248083	0.05
2000	959759	0.31
8000	14509549	4.51
16000	56705617	17.29
32000	221824366	71.20
40000	344461977	106.79
50000	534772302	174.50
60000	766265747	243.96
70000	1038159733	331.93
80000	1351433133	427.25
90000	1704858134	541.67
100000	2099198630	697.07

Output

All tests were run on 16 processors.



N = 5
Total: 14
Time elapsed: 0.01 seconds
N = 10
Total: 42
Time elapsed: 0.01 seconds
N = 20
Total: 152
Time elapsed: 0.01 seconds
N = 40
Total: 517
Time elapsed: 0.01 seconds
N = 80
Total: 1939
Time elapsed: 0.00 seconds
N = 160
Total: 7174
Time elapsed: 0.01 seconds
N = 320
Total: 27354
Time elapsed: 0.02 seconds

N = 640
Total: 103966
Time elapsed: 0.03 seconds
N = 1000
Total: 248083
Time elapsed: 0.05 seconds
N = 2000
Total: 959759
Time elapsed: 0.31 seconds
N = 8000
Total: 14509549
Time elapsed: 4.51 seconds
N = 16000
Total: 56705617
Time elapsed: 17.29 seconds
N = 32000
Total: 221824366
Time elapsed: 71.20 seconds
N = 40000
Total: 344461977
Time elapsed: 106.79 seconds
N = 50000
Total: 534772302
Time elapsed: 174.50 seconds
N = 60000
Total: 766265747
Time elapsed: 243.96 seconds
N = 70000
Total: 1038159733
Time elapsed: 331.93 seconds
N = 80000
Total: 1351433133
Time elapsed: 427.25 seconds
N = 90000
Total: 1704858134
Time elapsed: 541.67 seconds
N = 100000
Total: 2099198630
Time elapsed: 697.07 seconds

Problems Encountered

- i x j would sometimes be greater than N x N for example in a 6 x 6 array one instance would be 7 x 6. We solved it by clamping the value to N x N if it went over.
- MPI_Send has an int parameter "count" that was giving us errors because we were sending it an unsigned long. We solved it by splitting the char array up into chunk sizes of 30000.

Code

Source code can be found at https://github.com/TyllerAllen/CP431_Project