MPI Programming for N x N Multiplication Table

Brian Tiner
Adam Gumieniak
Matthew Buchanan
Duc Trung Nguyen

Wilfrid Laurier University
December 1st 2019

Introduction

Given an N x N multiplication table as an N x N array defined by $A_{ij} = i * j$, for i, j = 1….. N, we found the number of different elements in the table. Evidently this table is symmetric in its construction only the numbers above the diagonal are considered in our implementation.

M(N) = Number of different elements in an N by N multiplication matrix.

Table Verification By Hand

M(5) = 14

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

M(10) = 42

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

Implementation

The naive approach to this problem would be to produce a single array of size N by N and add unique elements to the array as you find them. Our algorithm does not use this approach since it would be far too inefficient. The only serial portion of our program is variable declarations so the entire algorithm is done in parallel. We use a work division algorithm that sets a start range and end range in the upper triangle for every processor to evenly distribute the numbers. Doing this solves the load balancing problem. We then loop through the entire matrix to find the correct start index in the upper triangle for each processor. Once the correct start index is found, we begin to loop through the rows of the upper triangle starting at that index, until the end index. Every value is passed to a function to check if the value is unique or not. The unique function calculates the maximum row that a value can be on by taking the floor of its square root. It then loops in descending order from that row to find the highest row that the current value occurs on. If the index of the value that is found is the same as the index passed to the function, then the function counts that as a unique value. If not then it is a duplicate value and is not counted as a unique value. Each processor finds the number of unique values in their section. Then the slave master approach is used for communicating the results. Every processor sends their unique number value to the zero processor and the zero processor adds them all together to get a final result.

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <mpi.h>

//function to determine if a number is unique
int unique(unsigned long long i, unsigned long long j, unsigned long long n){
	unsigned long long value = i * j; //actual value
	unsigned long long max_row = (int)floor(sqrt(value)); //maximum row the value can be
on
	unsigned long long row = max_row; //loop variable
	//counting down through rows
	while (row > 0){
		//if to see if value is on the row
		if (value%row == 0){
			//column will always be less then n, column and row will always be
unique
			//if its the closest value to the diagonal
			if (i == row){
				return 1;
			}
			//if else it is not the unique value
			else{
				return 0;
			}
		}
		row--;
	}
	return 0;
}
//standard work divsion function
void work_division(unsigned long long n, int p, int rank, unsigned long long *out_first, unsigned
long long *out_last){
	unsigned long long delta = n/p;
```

```
	unsigned long long range_start = delta * rank;
	unsigned long long range_end = delta * (rank + 1) - 1;
	if (rank < n%p){
		range_start += rank;
		range_end += rank + 1;
	}
	else{
		range_start += n%p;
		range_end += n%p;
	}
	*out_first = range_start + 1;
	*out_last = range_end + 1;
}

int main(int argc, char **argv){
	unsigned long long n = atoi(argv[1]); //problem size as argument
	int p, rank; //processors and rank
	unsigned long long result = 0, temp_result;
	unsigned long long start_range, end_range; //start and end range for work division
	MPI_Status status; //MPI_status
	//MPI Commands
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	//work division algorithm
	work_division(((n * (n + 1))/2), p, rank, &start_range, &end_range);

	unsigned long long count = 0; //to count through range
	//setting variables to index work division
	unsigned long long pre_i = start_range, running = start_range, x = 1;
	//counting through matrix to get to correct spot
	while(running > n - (x - 1)){
		running -= n - (x - 1);
		pre_i += x;
		x++;
	}
	//setting loop variables at correct spot in upper triangle
	unsigned long long _i = (pre_i - 1) / n + 1;
	unsigned long long _j = (pre_i - 1) % n + 1;
```

```
	//looping for first row for processor
	for (unsigned long long j = _j; j <= n; j++){
		//breaking loop if range is done
		if (count > end_range - start_range){
			break;
		}
		count++;
		//checking of value is unique
		if (unique(_i, j, n) == 1){
				result++;
		}

	}
	//adding 1 to i if the row was finished
	_i++;
	//looping for the rest of the range
	for (unsigned long long i = _i; i <= n; i++){
		for (unsigned long long j = i; j <= n; j++){
			//breaking if range is done
			if (count > end_range - start_range){
				break;
			}
			count++;
			//checking if value is unique
			if (unique(i, j, n) == 1){
				result++;
			}
		}
		//breaking loop if range is done
		if (count > end_range - start_range){
			break;
		}
	}
	//reciving from other processors
	if (rank == 0){
		for (int i = 1; i < p; i++){
			MPI_Recv(&temp_result, 1, MPI_UNSIGNED_LONG_LONG, i, 0,
MPI_COMM_WORLD, &status);
			result += temp_result;
```

```
                }
                printf("The number of unique values in a multiplication table of size %d by %d is
%lld\n", n, n, result);
        }
        //sending to 0 processor
        else{
                MPI_Send(&result, 1, MPI_UNSIGNED_LONG_LONG, 0, 0,
MPI_COMM_WORLD);
        }

        MPI_Finalize();
        return 0;
}
```

Results

Problem Size	Result	Wall Time (16 Processors)
5	14	00:00:09
10	42	00:00:06
20	152	00:00:07
40	517	00:00:07
80	1,939	00:00:08
160	7,174	00:00:07
320	27,354	00:00:07
640	103,966	00:00:07
1000	248,083	00:00:07
2000	959,759	00:00:07

8000	14,509,549	00:00:21
16000	56,705,617	00:01:48
32000	221,824,366	00:12:46
40000	344,462,009	00:24:31
50000	534,772,334	00:47:41
60000	766,265,795	01:21:19
70000	1,038,159,781	02:08:45
80000	1,351,433,197	03:11:19
90000	1,704,858,198	04:32:01
100000	2,099,198,630	06:11:29

Observations

According to the table above we observe that when the problem size is less than 8000 the wall time ranges from 00:00:07 to 00:00:09. The inconsistency in wall times for these problem sizes is likely from the small problem size itself. Evidently as the problem size increases so does the wall time but from our results we reach an interesting conclusion. For a problem size of 40,000 the multiplication table will have 1.6 billion values and for problem size of 80,000 the table will have 6.4 billion; doubling the problem size will quadruple the amount of numbers in the multiplication table. Based on this one would expect that the wall time scales accordingly to the numbers present in the multiplication table however we found that it increases by a factor of 10. The expected increase in time from test of size n to 2n should be the multiple m of the input

squared, in this case the multiple m = 2, m^2 = 4 times as long. However according to the table we see an increase of 2.5 * (m^2), in this case m = 2, 2.5 * (m^2) = 10 times as long.