Group-4
CP431-A
2023/03/30
TP

CP 431 Parallel Programming Term Project

Group 4

Name: Mitchell DeCarlo
E-mail: deca8530@mylaurier.ca

Name: Owen Milne
E-mail: miln6570@mylaurier.ca

Name: Lan You
E-mail: youx5710@mylaurier.ca

Name: Nicholas Sam
E-mail: samx8430@mylaurier.ca

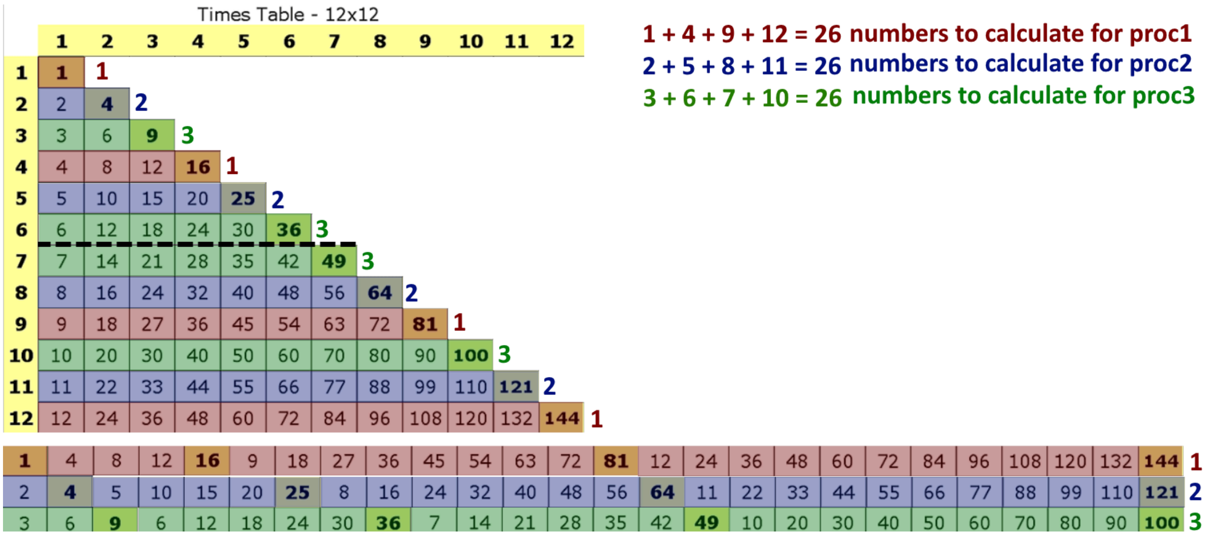
Group-4
CP431-A
2023/03/30
TP

We choose MPI programming for multiplication tables to be our project. To implement this algorithm, we used C. The files used in this assignment is: a3.c

This is our list of output when N = 5, 10, 20, 40, 80, 160, 320, 640, 1000, 2000, 4000, 8000, 16000, 32000, 40000, 50000, 60000, 70000, 80000, 90000, 100000.

We also have a time elapsed table, where we calculated the maximum speedup is about: 1.1667.

Due to the symmetric nature of the matrix, we do not need to consider the data above the diagonal. So, in effect, the algorithm needs to deal with a triangle. To solve the payload allocation problem, we use the algorithm as shown in the figure, which can guarantee a maximum deviation of N+1.



As shown in the figure, each processor calculates which row to check up until N/2, they get (i*num_procs)+my_rank from N/2 to N, they get N-(i*num_procs)+my_rank each processor then calculates products and stores in a hash set. This allows for the algorithm to quickly check if a certain number already exists.

Due to the size of the number (long long), this algorithm can use more than 64gb of memory on Ns upwards of about 95,000. This means on the TEACH cluster, we must use at least 2 nodes in these circumstances.

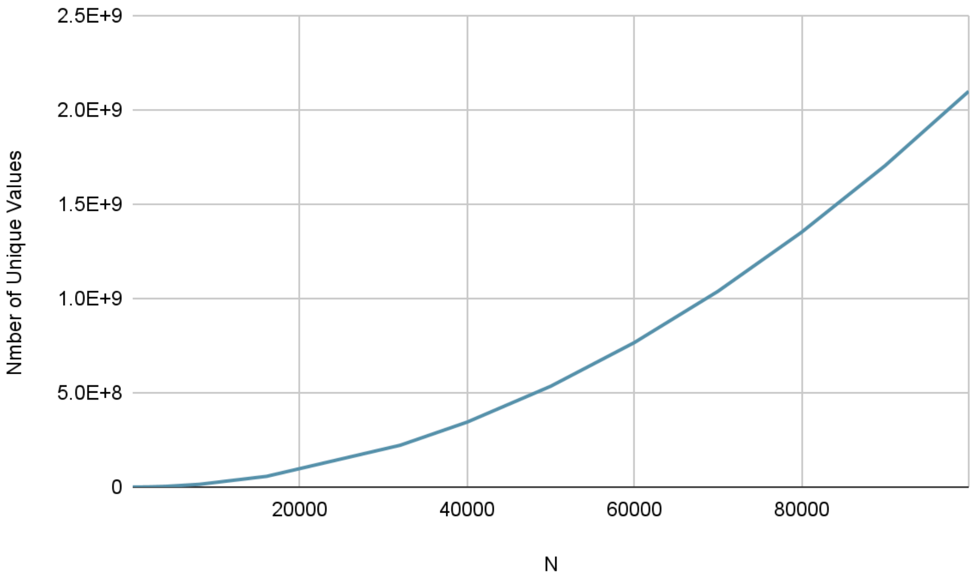
After each processor finds which rows to calculate and saves it in an array, the algorithm will start to populate the hash set. We decided to use a hash set because while it may use more memory, it is significantly faster than using a list. Once populating the hash set is complete, the processor then creates a list of numbers that exist in the hash set, and sends it to the main processor. While this may seem counter-intuitive, MPI struggled to send the massive hash set, so we had to resort to this instead. We found this was still quicker in general.

In the main processor 0, it gathers all the results and performs radix sort. Once sorted, it does one final pass on the list to check for duplicates, then prints the final result. The reason we chose to do a sort is because removing duplicates from a sorted array is O(n). So combined with radix sort, we can remove duplicates from the final list in O(n) time.

Group-4
CP431-A
2023/03/30
TP
Result:

N	M(N)
5	14
10	42
20	152
40	517
80	1939
160	7174
320	39880
640	103966
1000	248083
2000	959759
4000	3723723
8000	14509549
16000	56705617
32000	221824366
40000	344462009
50000	534772334
60000	766265795
70000	1038159781
80000	1351433197
90000	1704858198
100000	2099198630

Group-4
CP431-A
2023/03/30
TP



Output:
5:

```
1 load: 6
2 load: 3
0 load: 6
3 load: 0
number of elements received: 14

Final output: 14
```

10:

```
1 load: 11
2 load: 11
3 load: 11
0 load: 22
number of elements received: 52

Final output: 42
```

20:

```
1 load: 63
2 load: 42
0 load: 63
```

```
3 load: 42
number of elements received: 189

Final output: 152
```

40:

```
1 load: 205
2 load: 205
0 load: 205
3 load: 205
number of elements received: 710

Final output: 517
```

80:

```
1 load: 810
2 load: 810
0 load: 810
3 load: 810
number of elements received: 2713

Final output: 1939
```

160:

```
1 load: 3220
2 load: 3220
0 load: 3220
3 load: 3220
number of elements received: 10309

Final output: 7174
```

320:

```
1 load: 12840
2 load: 12840
0 load: 12840
```

```
3 load: 12840
number of elements received: 39880

Final output: 27354
```

640:

```
1 load: 51280
2 load: 51280
0 load: 51280
3 load: 51280
number of elements received: 154096

Final output: 103966
```

1000:

```
1 load: 125125
2 load: 125125
0 load: 125125
3 load: 125125
number of elements received: 369366

Final output: 248083
```

2000:

```
1 load: 500250
2 load: 500250
0 load: 500250
3 load: 500250
number of elements received: 1441857

Final output: 959759
```

4000:

```
1 load: 2000500
3 load: 2000500
0 load: 2000500
```

```
2 load: 2000500
number of elements received: 5638794

Final output: 3723723
```

8000:

```
1 load: 8001000
2 load: 8001000
0 load: 8001000
3 load: 8001000
number of elements received: 22107744

Final output: 14509549
```

16000:

```
1 load: 32002000
2 load: 32002000
0 load: 32002000
3 load: 32002000
number of elements received: 86882526

Final output: 56705617
```

32000:

```
1 load: 128004000
2 load: 128004000
3 load: 128004000
0 load: 128004000
number of elements received: 341912802

Final output: 221824366
```

40000:

```
1 load: 200005000
2 load: 200005000
0 load: 200005000
```

```
3 load: 200005000
number of elements received: 531659583

Final output: 344462009
```

50000:

```
0 load: 312506250
1 load: 312506250
2 load: 312506250
3 load: 312506250

number of elements received: 826729456
Final output: 534772334
```

60000:

```
0 load: 450007500
2 load: 450007500
3 load: 450007500
1 load: 450007500

number of elements received: 1185919514

Final output: 766265795
```

70000:

```
1 load: 612508750
0 load: 612508750
2 load: 612508750
3 load: 612508750
number of elements received: 1609058176

Final output: 1038159781
```

80000:

```
1 load: 800010000
2 load: 800010000
```

```
0 load: 800010000
3 load: 800010000
number of elements received: 2095853086

Final output: 1351433197
```

90000:

```
0 load: 1012511250
1 load: 1012511250
3 load: 1012511250
2 load: 1012511250
number of elements received: 2646268992

Final output: 1704858198
```

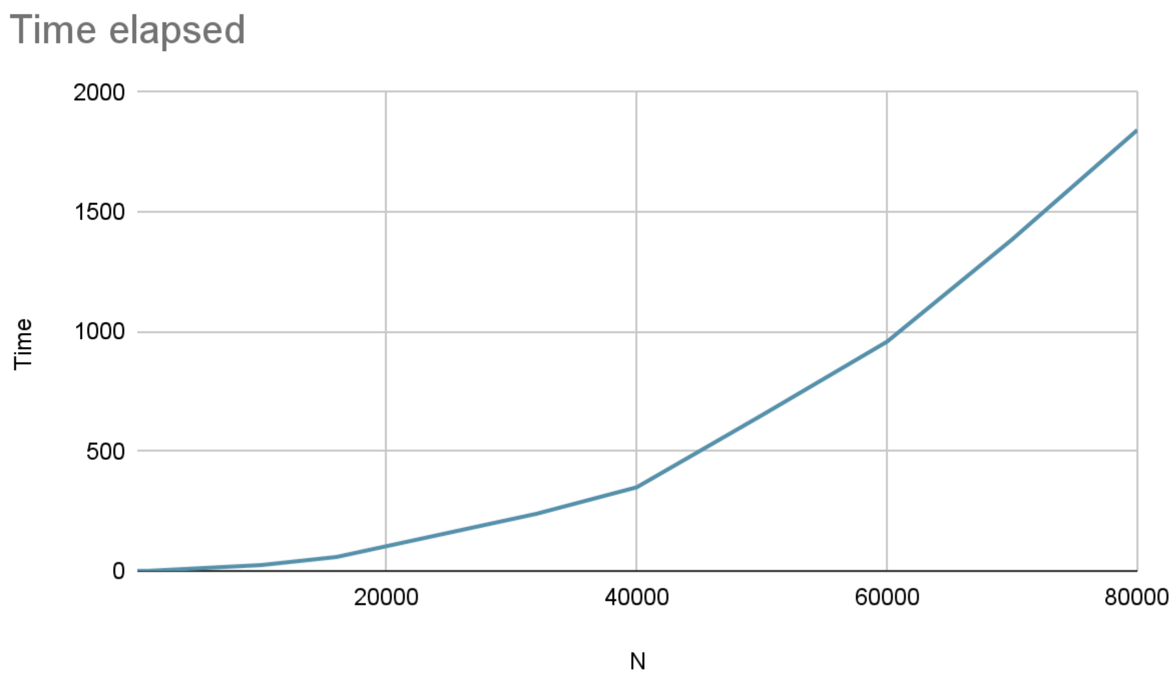
100000:

```
1 load: 1250012500
0 load: 1250012500
3 load: 1250012500
2 load: 1250012500
number of elements received: 3260318690

Final output: 2099198630
```

Group-4
CP431-A
2023/03/30
TP
Time elapsed:

N	Time(s)
100	0.002477
1000	0.187905
10000	24.481330
16000	58.126903
32000	238.807853
40000	349.494097
50000	650.521570
60000	957.821856
70000	1385.125379
80000	1842.105527



Group-4
CP431-A
2023/03/30
TP
Calculate efficiency:

The serial part of the program has a complexity of: O(MaxNumOfDigits(N)*N^2) + O(N^2)

The parallel part of the program has a complexity of: O(N^2)

Thus,we get(for 4 processors. p=3):

Size(n)	Serial	Parallel	Ratio(Serial)	Ratio(Parallel)	ψ(n,3)
80000	384x10^8	64x10^8	6/7	1/7	1.1053

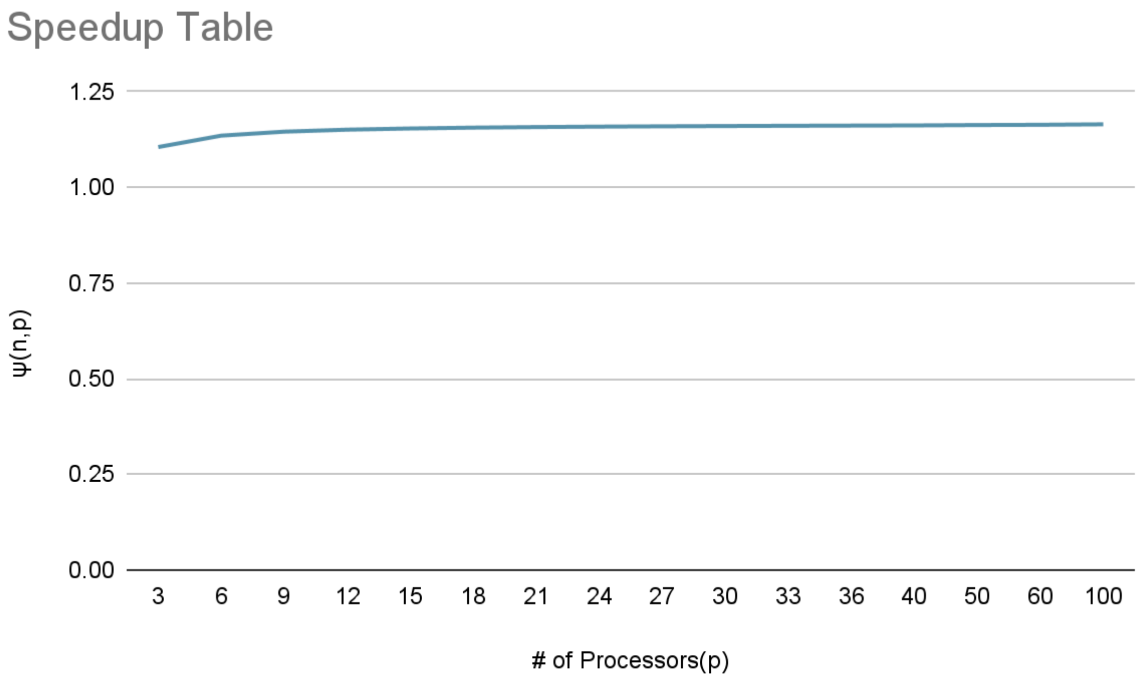
Now test for what is the limit of number of processors:

# of Processors(p)	ψ(n,p)
3	1.1053
6	1.1351
9	1.1455
12	1.1507
15	1.1538
18	1.1560
21	1.1575
24	1.1586
27	1.1595
30	1.1602
33	1.1608
36	1.1613

From here the increase of speedup becomes extremely slow:

40	1.1618
50	1.1628
60	1.1634
100	1.1647
1000000	1.1667

Group-4
CP431-A
2023/03/30
TP



Group-4
CP431-A
2023/03/30
TP
a3.c

```
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <ctype.h>
#include <math.h>
#include "mpi.h"

#define printload 1

bool isNumber(char number[])
{
    int i = 0;

    if (number[0] == '-')
        return false;
    for (; number[i] != 0; i++)
    {
        if (!isdigit(number[i]))
            return false;
    }
    return true;
}

int main(int argc, char *argv[])
{
    if (argc != 2 && isNumber(argv[1]))
    {
        printf("Usage: %s N\nN must be a positive integer greater than 0",
argv[0]);
        exit(1);
    }
    int my_rank, num_procs;
    double start_time, end_time, time_elapsed;
    MPI_Status status;
    long long *in = NULL;
    long long insize;
```

```
    long long *out = NULL;
    long long outsize = 0;
    int N = strtol(argv[1], NULL, 10);
    MPI_Init(&argc, &argv);
    MPI_Barrier(MPI_COMM_WORLD);
    start_time = MPI_Wtime();

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    // each processor finds which rows to calculate
    // this is O(n/p)
    int *calc_rows;
    int num_rows;
    num_rows = (N / 2) / num_procs;
    num_rows *= 2;
    if (my_rank < (N / 2) % num_procs)
    {
        num_rows += 2;
    }
    if (N % 2 != 0)
    {
        if (my_rank == (N / 2))
        {
            num_rows++;
        }
        else if (my_rank == 0 && (N / 2) + 1 >= num_procs)
        {
            num_rows++;
        }
    }
    calc_rows = malloc(num_rows * sizeof(int));
    long long counter = 0;
    for (int i = 0; counter < num_rows; i++)
    {
        calc_rows[counter++] = (i * num_procs) + (my_rank + 1);
        calc_rows[counter++] = N - ((i * num_procs) + (my_rank));
    }
    if (N % 2 != 0)
```

```
    {
        if (my_rank == (N / 2))
        {
            calc_rows[counter++] = (N / 2) + 1;
        }
        else if (my_rank == 0 && (N / 2) + 1 >= num_procs)
        {
            calc_rows[counter++] = (N / 2) + 1;
        }
    }
#ifdef printload // this code block prints the amount of multiplication
operations each processor must complete
    counter = 0;
    for (int i = 0; i < num_rows; i++)
    {
        counter += calc_rows[i];
    }
    printf("%d load: %d\n", my_rank, counter);
#endif

    // using a modified hash set, populate the hash set with numbers that
appear in the processor's products
    // this is O((n/p)^2)
    // using a list, where if you insert X, you traverse the list for X,
takes O((n/p)^2 log(n/p)) i think
    bool *nums;
    long long max_num = ((long long)N - my_rank) * (N - my_rank);
    nums = malloc((max_num) * sizeof(bool));
    memset(nums, false, (max_num) * sizeof(bool));
    for (int i = 0; i < num_rows; i++)
    {
        for (long long j = 1; j <= calc_rows[i]; j++)
        {
            nums[(calc_rows[i] * j) - 1] = true;
        }
    }
    free(calc_rows);
    for (long long i = 0; i < max_num; i++)
    {
```

```
        if (nums[i] == true)
        {
            out = realloc(out, (++outsize) * sizeof(long long));
            out[outsize-1] = i+1;
        }
    }
    free(nums);

    // send it all to master proc for final processing
    // uses a list so its O(n^2 logn), but at this point, n is pretty
heavily reduced, so its like, basically, n^2 but not really
    // i didn't use hash set here because im scared of crashing teach with
memory leaks
    printf("%d outsize: %lld\n", my_rank, outsize);
    if (my_rank == 0)
    {
        long long temp;
        for (int i = 1; i < num_procs; i++)
        {
            // MPI_Get_count only works up to int max, so we have to
send/recv the number manually before sending the array
            MPI_Recv(&insize, 1, MPI_LONG_LONG, i, 1, MPI_COMM_WORLD, NULL);
            in = realloc(in, insize * sizeof(long long));
            MPI_Recv(in, insize, MPI_LONG_LONG, i, 0, MPI_COMM_WORLD, NULL);
            temp = outsize;
            outsize += insize;
            out = realloc(out, outsize * sizeof(long long));
            memcpy(out + temp, in, insize * sizeof(long long));
        }
        printf("\nnumber of elements received: %lld\n", outsize);
        long long max = N * N;
        long long count[10] = {0,0,0,0,0,0,0,0,0,0};
        long long output[outsize + 1];
        for (long long place = 1; max / place > 0; place *= 10)
        {
            memset(count, 0, 10 * sizeof(long long));
            for (long long i = 0; i < outsize; i++)
            {
                count[(out[i] / place) % 10]++;
```

```
            }
            for (long long i = 1; i < 10; i++)
            {
                count[i] += count[i - 1];
            }
            for (long long i = outsize - 1; i >= 0; i--)
            {
                output[count[(out[i] / place) % 10] - 1] = out[i];
                count[(out[i] / place) % 10]--;
            }
            for (long long i = 0; i < outsize; i++)
            {
                out[i] = output[i];
            }
        }
        // then remove duplicates from sorted array O(n)
        printf("Radix sort completed\n");

        // long long j = 0; //use this if you want to print all numbers in the final table
        // temp = [];
        long long j = 1;
        for (long long i = 0; i < outsize - 1; i++)
        {
            if (out[i] != out[i + 1])
            {
                // temp[j++] = arr[i]; //use this if you want to print all numbers in the final table
                j++;
            }
        }
        // temp[j++] = arr[n - 1]; //use this if you want to print all numbers in the final table
        printf("\n");
        printf("Final output: %lld\n", j);
        /* to print all numbers in the final table
        printf("Final numbers: {");
        for (long long i = 0; i < j; i++)
        {
```

```
            printf("%lld, ", temp[i]);
        }
        printf("}");
        */
 }
        }
        printf("\n");
        printf("Final output: %lld\n", j);
        free(in);
    }
    else
    {
        MPI_Send(&outsize, 1, MPI_LONG_LONG, 0, 1, MPI_COMM_WORLD);
        MPI_Send(out, outsize, MPI_LONG_LONG, 0, 0, MPI_COMM_WORLD);
    }
    free(out);
    free(calc_rows);
    MPI_Finalize();
    return 0;
}
```