Multi-thread Implementation of the Interpolation of Elevations given an n x n matrix with t threads

Aron Resty Ramillano, *Member, ACSS,* Lyco Sheen Lacuesta, *Fellow, ACSS,*

***Abstract*—The abstract goes here.**

***Index Terms*—IEEE, IEEEtran, journal, LaTeX, paper, template.**

I. Introduction

THE way that we usually calculate things is through a serial method of calculation. Meaning that we do things one by one until we finish all tasks. There is also the matter of being able to do multi tasking but, in reality, it is similar to how CPU's implement Context Switching, where they are really not doing tasks simultaneously, but rather they are doing small bits of it interchangeably at a very fast frequency.

This is good enough for small and fast tasks like normal browsing, normal calculations of small arithmetic, and more. But when we are talking about calculating millions of data to come up with an answer that is dependent on that big data, we need to have more efficient ways to execute the calculations needed. This is where multi-threading comes in.

arr
March 21, 2023

A. Multi-threading

Lets talk about what is a thread. A thread is basically a single operation that is running on a CPU that is executing one task. These threads are whats responsible for executing the computations needed to carry out the tasks of your programs. Imagine if we only had 1 thread that is needed to calculate numbers ranging from 1 to 10,000 indices, it would take that thread 10,000 more time needed to calculate each index one by one.

Now, imagine if we had 5 threads to calculate the sum of two consecutive numbers from 1 to 10, in this case, we make pairs of (1,2), (3,4), ..., (9,10) that is equally given to all of 5 threads to calculate a specific formula within them. In this way, we have cut down our calculations from having 1 thread to 10 operations, to having 5 threads do one operation each, resulting to a time elapsed of only 1/5 the original time needed.

Luckily, we can implement multi-threading in various languages that is available today, but we will focus on the C Programming language, as the basis of all our research and activities.

Lyco Sheen Lacuesta is part of the ACSS, under my Academic Affairs Team, and has helped greatly with my C-language inquiries

1) Multi-threading in C: The C programming language has been around for quite some time, yet is still widely used by programmers, especially those who are teaching Computer Science to universities, as the basis of teaching the fundamentals of programming. C is a low-level programming language that is the basis of many other languages such as C++, Java, Python, and more. Enough with the introduction, my main motivation for using C as my language in this endeavor is:

1) To improve my skills in using the C Language
2) I believe that it is a very fast language that can help me achieve more in my interpolation problems
3) Multi-threading is available.
4) I know how to code multi-threading into my algorithms

We can implement multi-threading by making use of *pthreads*, which is part of the GCC library for C, which enables the C Programming language to have multiple threads that execute different functions as needed, inside our program.

The basic syntax would be first including the structure of the arguments of the threads in order for it to have and contain data.

```
typedef struct
{
    float **M;
    int n;
    int start;
    int end;
} ThreadArgs;
```

Then we can implement threads using the following snippet of code:

```
pthread_t threads[num_threads];
ThreadArgs args[num_threads];
```

We can now call the threads one by one, by index, and have them execute our function which is found in this code:

```
pthread_create(&threads[i], NULL,
    ↪ thread_func, &args[i]);
```

After solving the multi-threading part, we need to know how will we divide our matrix into proper sub-matrices that can be solved by our individual threads that we have implemented in our code.

```
// This is basically the submatrices that we
    ↪ are going to make.
    int chunk_size = n / num_threads;

    for (int i = 0; i < num_threads; i++)
    {
        args[i].M = M;
        args[i].n = n;
        args[i].start = i * chunk_size;
        args[i].end = (i + 1) * chunk_size;

        // Because the matrix should include the
            ↪  0th coordinate, we would have to
            ↪  adjust our calculation to have
            ↪ the last submatrix to be able to
            ↪ handle a second row.
        if (num_threads > 1 && i == num_threads
            ↪ - 1)
        {
            args[i].end++;
        }

        // Actual implementation of creating a
            ↪ thread
        // We pass the arguments stated above on
            ↪  where on the matrix they should
            ↪ start
        // In this case, I made it so they will
            ↪ have separate row lines to start
            ↪ iterating from
        pthread_create(&threads[i], NULL,
            ↪ thread_func, &args[i]);
    }
```

We can see that from my code, I have implement the sub-matrix division in a way that it will be in a row-major order, that it will divide rows into equal distances that my program will be able to take advantage of computing, because in the first place, I have used row-major order in my algorithms.

II. Research Questions

A. What do you think is the complexity of estimating the point elevation of a square matrix with given/randomized values at grid points divisible by 10 when using n concurrent processors? The obvious processor assignment is one column of M for each processor.

$O(n^2/n)$, since we know that the calculation of the whole matrix is already $O(n^2)$, and we now have, at our hands, n processors, which means we can divide the whole complexity into the processors that we have.

B. What do you think is the complexity of estimating the point elevation of a square matrix with given/randomized value sat grid points divisible by 10 when using n/2 concurrent processors (what is the obvious processor assignment here)? What about with n/4 concurrent processors (i.e. processor assignment)? What about with n/8 concurrent processors?

It will depend on the number of processors, therefore the simplified answer would be $O(n^2/(n/2))$ or $O(n^2/(n/4))$ or $O(n^2/(n/8))$.

C. Why do you think the running time of t=1will be higher than the average that was obtained in Exercise 01?

My answer here would be false, since it neither improved, nor decreased in performance, it stayed roughly within margins because my implementation is roughly the same, and my Exercise 01 is also a serial programming implementation.

D. Do you think you can go as far as $= n$? If not, what about $= n/2$? Or,t$= n/4$? Or,t $= n/8$?

Yes, but, it would be inefficient and would actually be slower if the cores or processors in your CPU is not assigned 1:1 with the threads that your program is creating. The reason being is that there is a small delay in the creation and execution of the threads, therefore separating the function execution of an n x n matrix with a separate thread each n, means that it is more likely to have latency issues because of the other parts of the code that is causing delay, rather than it having t = 1.

But there is a sweet spot of it having the lowest runtime, with the most threads that it can, which can be seen in this figure here:

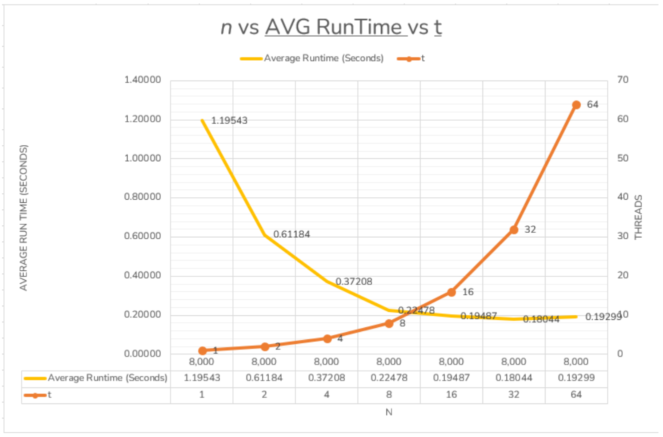
III. Research Activity 1

n	t	Time Elapsed			Average Runtime (Seconds)
		Run 1	**Run 2**	**Run 3**	
8000	1	1.18922	1.17313	1.22394	1.19543
8000	2	0.59971	0.59533	0.64048	0.61117
8000	4	0.39293	0.34035	0.38297	0.37242
8000	8	0.22456	0.22133	0.22845	0.22478
8000	16	0.19903	0.19233	0.19326	0.19487
8000	32	0.17674	0.18352	0.18107	0.18044
8000	64	0.18431	0.19296	0.20171	0.19399

IV. Research Activity 2

A. Repeat research activity 1 for $n = 16,000$ and $n = 20,000$. Do you think you can achieve $n = 50,000$ and evenn $= 100,000$? Try to see if you can. If you were able to do so, why do you think you can now do it? If not yet, why do you still can not?

I can still run them and I can see the differences in the calculation times when we reach the 64 thread part. But my machine is clearly struggling with the absurd amount that it has to calculate and the memory that it requires, that I am starting to think that memory is my enemy this time around. It kills itself at around the 50,000 x 50,000 matrix calculation



N	8,000	8,000	8,000	8,000	8,000	8,000	8,000
Average Runtime (Seconds)	1.19543	0.61184	0.37208	0.22478	0.19487	0.18044	0.19299
t	1	2	4	8	16	32	64

Fig. 1. N vs T graph

V. Conclusion

Multi-threading is an important part of the parallelization of lots of algorithms and programs in our world. It is what enables faster repetitive computing across a large data set. But it does not come easy in terms of implementing these types of programming in a very efficient way.

Yes, we can make do with our current multi-threading algorithms in these small data sets that we offer to our program, but it can be difficult to scale it up even further into real-world scenarios and data sets.

Acknowledgment

The authors would like to thank Lyco Lacuesta for being an integral part of my understanding of this C code and its underlying implementations. I would not have done it without you

References

[1] https://www.geeksforgeeks.org/interesting-facts-about-c-language/

Aron Resty Ramillano He is currently a 3rd Year BS Computer Science Student from the University of the Philippines - Los Banos. He aims to achieve new heights with his fresh endeavor in the Computer Science field and hopes to create more programs that are aimed towards helping the community go towards proper .