ASSIGNMENT 4: PARALLELIZATION

Stefanos Tsampanakis
stefanos.tsabanakis@gmail.com

Sajad Sharhani
sharhai.sajad@gmail.com

Atefeh Aramian
.atefeh.aramian.0970@gstudent.uu.se

March 3, 2023

1 Introduction

This assignment is a continuation of the assignment 3 in which we are tasked to parallelize our serially optimized code of the gravitational N-body problem. Parallel computing is the process of executing multiple instructions simultaneously by dividing a program into multiple smaller tasks that can be run concurrently. Parallel computing has become increasingly important in modern computing, as it allows for faster execution times and more efficient use of resources. Two popular methods which we will implement in this assignment are Pthreads and OpenMP.

2 Parallel processing

2.1 Parallelizability analysis

The first step before the implementation is to find out which part of our code is parallelizable. The bulk of the computations is performed in our nested 3 loops. The outer loop which iterates through the timesteps is not parallelizable becauce the positions and vector speeds of the stars is dependent on the previous step.

The timestep loop in our simulation can be divided into two distinct parts. In the first part, we have utilized nested for-loops to iterate through all the stars in our simulation. For each star, we calculate the forces that are applied to it by every other star in the simulation. Using these calculated forces, we update the velocity (vx and vy) of each particle.

In the second part of the timestep loop, we use the updated velocity values to compute the new positions (x and y) of each star. By performing these computations in each timestep loop, we are able to simulate the motion of particles in our system over time.

Parallelizing the two parts within the timestep loop can result in significant improvements in simulation time and overall performance. Both parts are perfectly parallelizable, but it's crucial to ensure that the first part completes before the second part begins. This is particularly important for the first part, which has a time complexity of $\mathcal{O}(N^2)$ and involves the majority of our computations.

2.2 Pthreads

Pthreads is a library in the C programming language that allows for multi-threaded programming. Pthreads stands for POSIX Threads, where POSIX refers to the Portable Operating System Interface, a set of standards for operating systems. It was designed to provide a standardized interface and a low level control for creating and managing threads

Pthreads provides a number of features that make multi-threaded programming easier and more efficient. These features include:

- Support for thread creation and termination
- Synchronization mechanisms, such as mutexes and condition variables

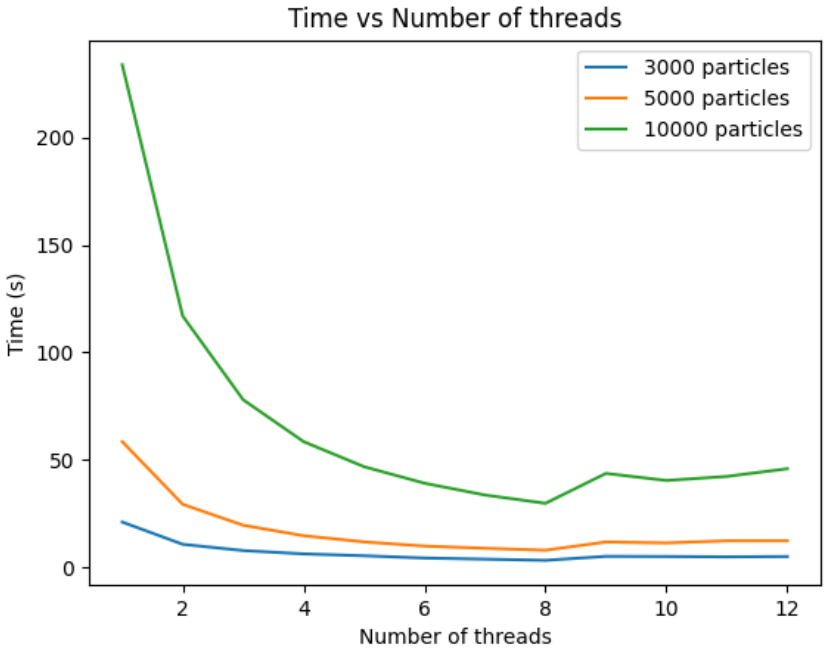

Figure 1: Pthread running time in second

- Thread-specific data
- Thread scheduling and priority management
- Signal handling

In pthreads, a thread function is a function that is executed by a thread. The thread function is passed as an argument to the pthread_create() function, which creates a new thread that executes the thread function.

The thread function takes a single void pointer argument arg which can be used to pass data to the thread function. The return value of the thread function must be a void pointer (void *), which can be used to pass data back from the thread.

Inside the thread function, the thread performs the work it was created to do. This can include any number of tasks, such as processing data, performing calculations, or communicating with other threads or processes.

It is important to note that each thread has its own stack, and therefore has its own set of local variables. This means that any data that needs to be shared between threads must be passed through arguments or global variables.

Based on all that we performed the following modifications to our code.

- We made certain variables as well as the *star* array and our struct of arrays *comp* global so as they would be accessed by our main function and our threaded function. This allows us to share data and perform the computations without needing to pass it through arguments
- We created a new structure called Loop_positions which would specify where each thread will start and end its work. By using this structure, we can divide the work among multiple threads and avoid having them interfere with each other.

  The two threaded functions can be viewed in Listing1. And the thread creation in the main can be viewed in Listing2

2.3 OpenMP

OpenMP is an API for shared memory multiprocessing programming in C, C++, and Fortran. It provides a portable, scalable, and easy-to-use programming model for parallel processing using compiler directives, library functions, and environment variables. It supports several types of parallelism, including task, loop, and section parallelism, and provides constructs for managing data sharing and synchronization among threads.

OpenMP provides a number of features that are useful for shared memory multiprocessing programming. Some of the key features of OpenMP include:

- Compiler Directives: OpenMP provides a set of compiler directives that allow the programmer to specify how the program should be parallelized. These directives are added to the source code and tell the compiler how to distribute the work among the threads.
- Task Parallelism: OpenMP supports task parallelism, which allows the programmer to create parallel tasks that can be executed independently of each other. This feature is useful for applications that require a lot of independent computations.
- Loop Parallelism: OpenMP provides constructs for parallelizing loops, which can greatly improve the performance of many numerical algorithms.
- Section Parallelism: OpenMP supports section parallelism, which allows different sections of the code to be executed in parallel.
- Data Sharing: OpenMP provides constructs for managing the sharing of data between threads, including private variables, shared variables, and thread-local storage.
- Synchronization: OpenMP provides constructs for synchronizing the execution of threads, including barriers, critical sections, and atomic operations.
- Runtime Library: OpenMP provides a runtime library that manages the creation and synchronization of threads, as well as other runtime tasks.

OpenMP works by adding directives to a program's source code that specify which parts of the code can be executed in parallel. These directives are then interpreted by the compiler and used to generate code that will execute the program in parallel on a multi-core processor.

In our case, we used the "OpenMP parallel for" directive to parallelize the loop that calls the "calc_velocity" function. The "default(none)" clause tells OpenMP to generate an error if a variable is referenced in the parallel region but is not explicitly declared as shared or private. The "shared(num_threads, data)" clause specifies that the "num_threads" and data variables should be shared among the threads.

Each thread will execute the "calc_velocity" function on a different set of elements of the data array, as specified by the loop index i.

The implementation can be viewed in Listing3.

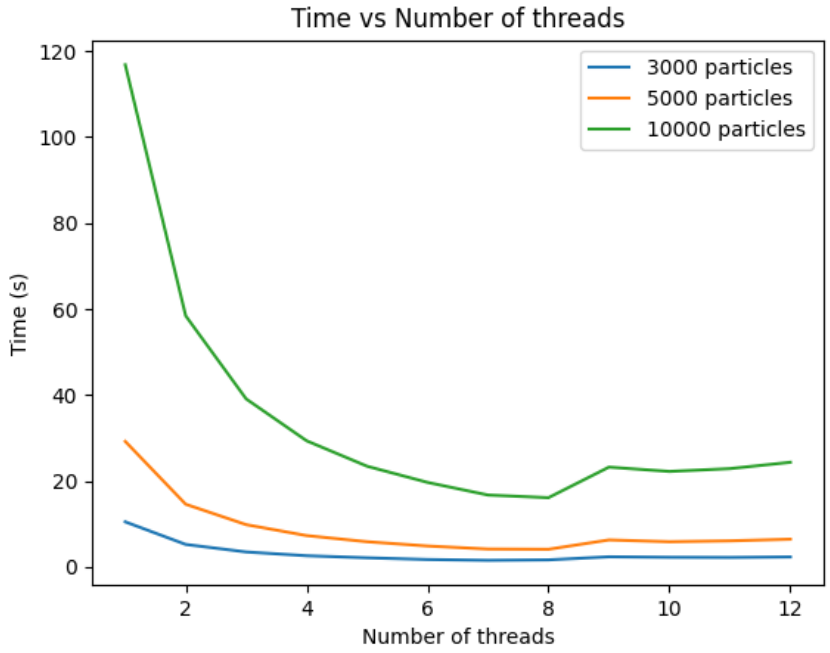

Figure 2: OpenMP running time in second

3 Conclusion

Our experiment, utilizing an Intel(R) Xeon(R) CPU E5520 @ 2.27GHz with 8 available cores, has resulted in a significant and nearly linear increase in performance for both Pthreads and OpenMP implementations. We have observed linear speedup when dividing the workload into a number of threads that is smaller or equal to the available cores, which in this case is 8. However, when dividing the work to a number of threads that exceeds the available cores, we have observed a decrease in performance.

These findings demonstrate the importance of optimal workload distribution and thread management for achieving optimal performance when parallelizing code on multicore processors. It is essential to consider the number of available cores and balance workload distribution accordingly to avoid negatively impacting performance.

Another point worth noting that there is a significant overhead associated with recreating threads in each step of the process. If we were able to move the thread creation outside of the step loop and optimize data locality in each thread, we could potentially see an even greater speedup and possibly even a superlinear performance increase. Unfortunately, with the current algorithm, we are unable to verify this due to the existing limitations.

Overall, parallelization with OpenMP and Pthreads can significantly improve the performance of code on multicore processors, but it requires careful design and implementation to achieve optimal results. With the increasing prevalence of multicore and multiprocessor systems, the ability to parallelize code effectively is becoming an increasingly important skill for programmers to have.

4 References

GNU Compiler Collection (GCC): https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
OpenMP Reference Guides: https://www.openmp.org/resources/refguides/

```
void* calc_velocity(void *arg){
    Loop_positions *data = (Loop_positions *) arg;
    int start=data->start;
    int end = data->end;
    double r,rx,ry;         //registers to caclulate the the distance vectors
    for(int i=start;i<end;i++)
    {

        vector a={0,0};
        double denominator;
        for(int j=0;j<i;j++)
        {
                rx=comp.x[i]-comp.x[j];
                ry=comp.y[i]-comp.y[j];
                r=sqrt((rx*rx)+(ry*ry))+1e-3;
                denominator=1.0/(r*r*r);
                a.x+=(-G*comp.m[j]*rx)*denominator;
                a.y+=(-G*comp.m[j]*ry)*denominator;
        }
        for(int j=i+1;j<N;j++){
                rx=comp.x[i]-comp.x[j];
                ry=comp.y[i]-comp.y[j];
                r=sqrt((rx*rx)+(ry*ry))+1e-3;
                denominator=1.0/(r*r*r);
                a.x+=(-G*comp.m[j]*rx)*denominator;
                a.y+=(-G*comp.m[j]*ry)*denominator;
        }
        stars[i].vx+=delta_t*a.x;
        stars[i].vy+=delta_t*a.y;
    }
}

void* positions(void *arg){
    Loop_positions *data = (Loop_positions *) arg;
    int start=data->start;
    int end = data->end;
        for(int i=start;i<end;i++)
        {
            stars[i].x+=delta_t*stars[i].vx;
            stars[i].y+=delta_t*stars[i].vy;
        }
}
```

Listing 1: Implementation of the threaded functions positions and calc_velocity

```
    for( int i = 0; i < num_threads; i++ )
    {
        data[i].start=i*range;
        data[i].end=(1+i)*range;
        if(i==num_threads-1)
        {
            data[i].end=N;
        }
        int start=data[i].start;
        int end = data[i].end;
    }
    for(int n=0;n<nsteps;n++)                    // loop through all the time stepts
    {
        for(int i=0;i<N;i++){
        comp.x[i]=stars[i].x;
        comp.y[i]=stars[i].y;
        comp.m[i]=stars[i].m;
        }
        //Initialize the threads
        for( int i = 0; i < num_threads; i++ )
            pthread_create(&thread[i], NULL, calc_velocity,(void*)&data[i]);

        // Wait for all threads to finish.
        for(int i=0;i<num_threads;++i)
            pthread_join(thread[i], NULL);

        //Initialize the threads
        for( int i = 0; i < num_threads; i++ )
            pthread_create(&thread[i], NULL, positions,(void*)&data[i]);

        // Wait for all threads to finish.
        for(int i=0;i<num_threads;++i)
            pthread_join(thread[i], NULL);
    }
```

Listing 2: Pthread creation and job division

```
    for (int i = 0; i < num_threads; i++)
    {
        data[i].start = i * range;
        data[i].end = (1 + i) * range;
        if (i == num_threads - 1)
        {
            data[i].end = N;
        }
        int start = data[i].start;
        int end = data[i].end;
        // printf("%d\n",start);
        // printf("%d\n",end);
        // printf("\n");
    }
    for (int n = 0; n < nsteps; n++) // loop through all the time stepts
    {
        for (int i = 0; i < N; i++)
        {
            comp.x[i] = stars[i].x;
            comp.y[i] = stars[i].y;
            comp.m[i] = stars[i].m;
        }

#pragma omp parallel for default(none) shared(num_threads, data)
        for (int i = 0; i < num_threads; ++i)
            calc_velocity(&data[i]);

        positions(); // calculate the x and y
    }
```

Listing 3: OpenMp Implementation