

High Performance and Parallel Computing

Assignment 3

Björn Mosten

Contents

Chapter 1

	Page
1.2 Introduction	2
Hardware — 2	
1.3 Solution	2
Data Structures — 2 • Algorithm — 2	
1.4 Optimization Techniques	2
Parallelization — 2 • Loop Unrolling — 3	
1.5 Compiler	3
1.6 End Results	3

Chapter 1

1.2 Introduction

The goal of this project was to create a well-performing quick-sort process that scales well with multiple threads. It was written in C and utilizes the OpenMP API for parallelization.

1.2.1 Hardware

CPU: AMD Ryzen 7 3700X 8-Core Processor, with a maximum clock speed of 3.6 GHz. It has 256 KB of L1i-cache, 256 KB of L1d-cache, 4 MB of L2-cache and 32 MB of L3-cache. It has 8 cores and 2 threads per core. RAM: 32 GB of DDR4 RAM.

1.3 Solution

1.3.1 Data Structures

Each thread is given a struct containing the following data:

- `int * data` - The data to be sorted.
- `int * data_free` - A pointer to the start of the data array. This is often, but not always, the same as *data*.
- `int * new_data` - Data array to be used when merging. Combines the data from *tmp_data* from another thread and *data* from the current thread.
- `int * tmp_data` - Temporary data array containing the data that is to be merged into another thread.
- `int data_n` - The number of elements in the data array.
- `int tmp_data_n` - The number of elements in the tmp_data array.

1.3.2 Algorithm

Each thread is given an equal amount of data to sort. Initially, each thread sorts its own data using the default qsort function. All threads are put into a single group and common pivot point is calculated by taking the average of the median values of the data in each thread.

1.4 Optimization Techniques

1.4.1 Parallelization

Parallelization was done with OpenMP.

OpenMP

The outer-most loop of the body calculation was parallelized using the `parallel for` directive with T number of threads. OpenMP is given the dynamic scheduling policy, which distributes the iterations between threads dynamically with no specific order. The dynamic scheduling policy gave a 20% performance improvement over the static scheduling policy.

OpenMP could handle the parallelization of the main loop without any major changes to existing code. A `omp reducer` directive was added to the force calculation loop to avoid having to manually add the forces together after the loop.

1.4.2 Loop Unrolling

GCC was given the `-funroll-loops` flag to avoid us having to unroll loops manually. This tells the compiler to unroll loops that are small enough, reducing the number of jumps the program has to make. Additionally, the loops dependent on the number of particles (N) were given a GCC pragma hint to unroll with a value of 10.

1.5 Compiler

GCC 11 was used for compilation with the following flags:

```
-Ofast
-flto
-ftree-vectorize
-march=native
-mtune=native
-fopenmp
-funroll-loops
-mno-vzeroupper
-fno-trapping-math
-fno-signaling-nans
-funsafe-math-optimizations
```

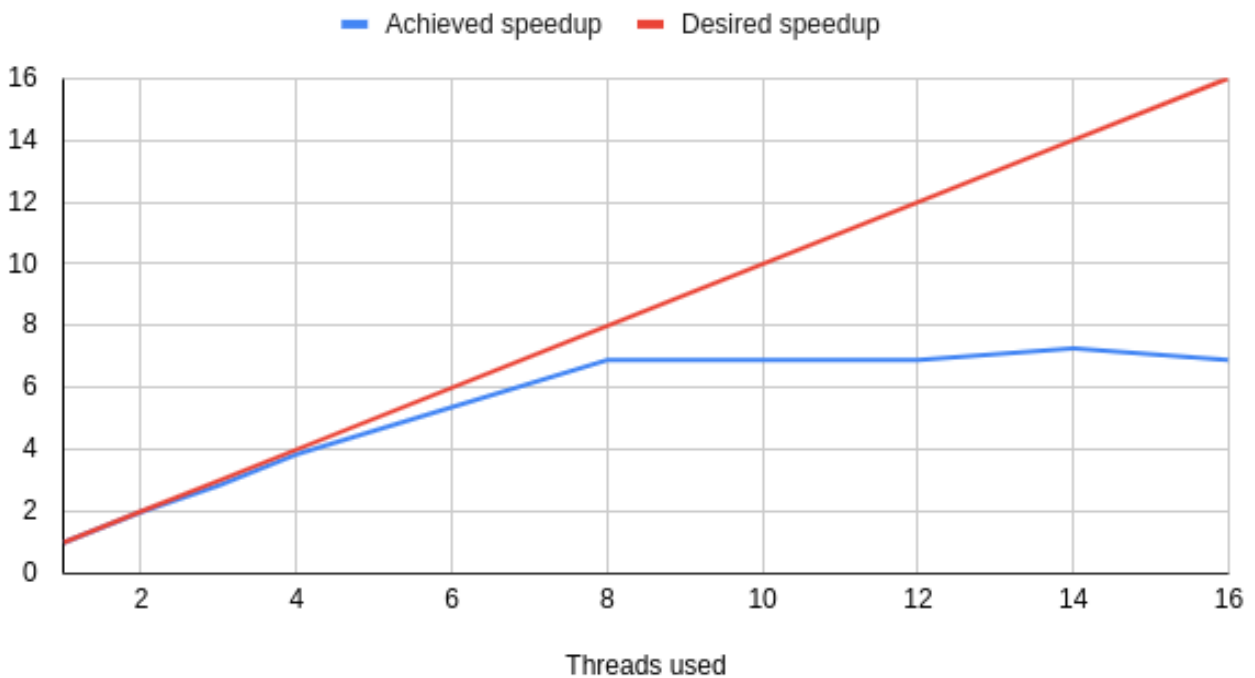
1.6 End Results

Parameters: 5000 particles, 100 timesteps, 0.001 timestep size.

Pthreads



OpenMP



Bibliography