McMaster University

Parallel Alphabetical Sort of Names

SFWRTECH 4CC3 - Parallel Programming

Aiden WangYang Li - 400374502
Jinyao Ma - 001433428
11-25-2022

Introduction

In this project, we as a group have implemented and tested Quick Sort, Merge Sort, and Bubble Sort in comparison with the default .Net LINQ sorting method. This report will detail what we found and the problems we encountered. Topics include comparisons of Quick Sort, Merge Sort, .Net LINQ sorting methods, and different implementations of bubble sorting in sequential and parallel methods.

Code Execution

Our program is written in the programming language C#, and the compiler of Visual Studio 2022 is used in this project. The following steps can guild you to the results calculated by the program:

1. Open the "ProjectParallelSort" project by using Visual Studio 2022 with net6.0 installed
2. Change the configuration of the project to release mode
3. Build the Solution if necessary
4. Put the names.txt file under the path of *ProjectParallelSort\bin\Release\net6.0*
5. The results will be displayed on the terminal screen

Quick Sort vs. Merge Sort vs. .NET LINQ Sort Algorithms

The results are calculated on a computer with 6 cores and 6 threads.

```
> Loaded names.txt successfully, and 10000 names had been read.

The number of processors on this computer is 6.
Setting MinDepth of the Parallel Process as 6.

Sorting by .NET Linq Algorithm
> Code took 13 milliseconds to execute.

Sorting by Quicksort Algorithm
> Code took 17 milliseconds to execute.

Sorting by Quicksort Algorithm using Parallel Invoke
> Code took 7 milliseconds to execute.

Sorting by Mergesort Algorithm
> Code took 20 milliseconds to execute.

Sorting by Mergesort Algorithm using Parallel Invoke
> Code took 10 milliseconds to execute.
```

Figure 1 – Result generated from the code.

- The default .NET LINQ sorting method took **13** milliseconds to execute
- The Quick Sort algorithm took **17** milliseconds to execute
- The Quick Sort algorithm used parallel invoke took **7** milliseconds to execute
- The Merge Sort algorithm took **20** milliseconds to execute
- The Merge Sort algorithm used parallel invoke took **10** milliseconds to execute

The results show that the basic Quick Sort and Merge Sort algorithms are slower than the .NET LINQ sorting method. And when we apply the Parallel Invoke Methods to these two sorting algorithms, the

calculation speed is greatly improved. The Parallel Invoke method in C# is used to launch multiple tasks that are going to be executed in parallel. (Parallel Invoke Method in C# With Examples, 2022) [1] By using the Parallel Invoke Method, we can easily run sub-methods of the divide and conquer algorithms in parallel.

```
// QuickSort Using Parallel Invoke
5 references
public void Quick_Sort_Parallel(List<Name> data, int left, int right, int depth)
{
    int middle;

    if (left < right)
    {
        middle = Partition(data, left, right);
        if (depth > minDepth)
        {
            Quick_Sort_Parallel(data, left, middle - 1, depth + 1);
            Quick_Sort_Parallel(data, middle + 1, right, depth + 1);
        }
        else
        {
            Parallel.Invoke(
                () => Quick_Sort_Parallel(data, left, middle - 1, depth + 1),
                () => Quick_Sort_Parallel(data, middle + 1, right, depth + 1)
                );
        }
    }
}
```

Figure 2 – Example of Quick Sort using Parallel Invoke Method.

In conclusion, this Parallel Invoke Method greatly improves the calculation speed of the sorting algorithms with the divide and conquer logic. Improved speed even surpasses the .NET LINQ sorting method.

Attempts of Thread Pool Method

We have tried to use the thread pool to implement the methods, but we did not find a way to know when the threads complete their jobs and were recycled. Without knowing the time of when the work is done, we cannot rejoin the threads, then let the timer stop recording. In the example below, we can see that the timer is stopped, but the thread pool is still working on sorting names.

```
Sorting by Quicksort using ThreadPool
ERROR: failed to save QuickSort ThreadPool.txt, Collection was modified;
enumeration operation may not execute.
> Code took 9 milliseconds to execute.
```

Figure 3 – Example of Quick Sort using Thread Pool

[1] Parallel Invoke Method in C# with Examples. (2022, November 24). Dot Net Tutorials. https://dotnettutorials.net/lesson/parallel-invoke-method-csharp/

Bubble Sort Algorithm

We implemented a sequential bubble sort and a parallel bubble sort. The parallel bubble sort could accept options to enable using Batches which meant limiting the number of threads to be less or equal to the number of processors. Also, it could accept another option to enable using the ThreadPool package instead of the Parallel package when using Batches. The ThreadPool package seemed to have a better ability to distribute threads to idle processors, or the Parallel package spent more time because it used and depended on the ThreadPool package.

After testing in Debug mode, the parallel bubble sort with Batches and Thread Pool used had the highest speed up which was 4.20x. Running in Release mode, it speeded up to 4.34x. However, if I ran the built-release version of the executable outside the Visual Studio, the speed-up of it went down to 2.93x, and the parallel bubble sort without Batches and using Parallel directly had a speed-up of 3.27x which became the highest.

The 4x speedup was caused by busy IDE affecting the sequential sorting, and when the parallel sorting started, IDE cooled down and the parallel sorting actually got more resources to use. Because it was an NP type of computation complexity problem, more resources and cores available should provide faster speedup for parallel bubble sorting.

```
Start sequential bubble sorting...
> Sequential bubble sort took 10178 milliseconds to execute.

Start parallel bubble sorting (batch: N)...
> Parallel bubble sort (batch: N) took 3262 milliseconds to execute.
> Parallel bubble sort (batch: N) is 3.12x faster than Sequential bubble sort.

Start parallel bubble sorting (batch: Y, package: Parallel)...
> Parallel bubble sort (batch: Y, package: Parallel) took 3664 milliseconds to execute.
> Parallel bubble sort (batch: Y, package: Parallel) is 2.78x faster than Sequential bubble sort.

Start parallel bubble sorting (batch: Y, package: ThreadPool)...
> Parallel bubble sort (batch: Y, package: ThreadPool) took 2522 milliseconds to execute.
> Parallel bubble sort (batch: Y, package: ThreadPool) is 4.04x faster than Sequential bubble sort.
```

Figure – 4 Result Generated from Bubble Sort Code

The below are test results of the bubble sort algorithm under Debug mode.

Before pre-run Parallel.For() when initialize algorithm

- When 500 names, speed up (no batch) is 1.04x.
- When 1,000 names, the speed up (no batch) is 0.95x.
- When 1,500 names, the speed up (no batch) is 0.98x.
- When 2,500 names, the speed up (no batch) is 1.39x.
- When 5,000 names, the speed up (no batch) is 2.54x.
- When 7,500 names, the speed up (no batch) is 3.25x.
- When 10,000 names, the speed up (no batch) is 2.85x.
- When 500 names, the speed up (use batch) is 1.07x.
- When 1,000 names, the speed up (use batch) is 0.94x.
- When 1,500 names, the speed up (use batch) is 0.75x.
- When 2,500 names, the speed up (use batch) is 1.59x.
- When 5,000 names, the speed up (use batch) is 2.19x.

- When 7,500 names, the speed up (use batch) is 2.51x.
- When 10,000 names, speed up (use batch) is 2.53x.

After pre-run Parallel.For() when initialize algorithm

- When 500 names, the speed up (no batch) is 1.17x.
- When 1,000 names, the speed up (no batch) is 1.10x.
- When 1,500 names, the speed up (no batch) is 0.88x.
- When 2,500 names, the speed up (no batch) is 1.87x.
- When 5,000 names, the speed up (no batch) is 3.38x.
- When 7,500 names, the speed up (no batch) is 4.08x.
- When 10,000 names, the speed up (no batch) is 4.45x.
- When 500 names, the speed up (use batch) is 1.23x.
- When 1,000 names, the speed up (use batch) is 1.06x.
- When 1,500 names, the speed up (use batch) is 0.81x.
- When 2,500 names, the speed up (use batch) is 1.95x.
- When 5,000 names, the speed up (use batch) is 2.93x.
- When 7,500 names, the speed up (use batch) is 3.14x.
- When 10,000 names, the speed up (use batch) is 3.24x.

After switching the .NET Parallel package to the ThreadPool package for batch distribution

- When 500 names, the speed up (use batch) is 0.96x.
- When 1,000 names, the speed up (use batch) is 0.95x.
- When 1,500 names, the speed up (use batch) is 0.82x.
- When 2,500 names, the speed up (use batch) is 2.62x.
- When 5,000 names, the speed up (use batch) is 4.08x.
- When 7,500 names, the speed up (use batch) is 4.21x.
- When 10,000 names, the speed up (use batch) is 4.20x.

In conclusion, Parallel without using batches seems faster. Maybe .NET Parallel package already does some optimization about distributing threads to idle processors. Or the .NET Parallel package may not distribute threads to processors fairly like some batch threads may be lined on the same processor. Actually, if using batches, there are more basic operations inside a thread function, so if the number of swap pairs becomes too many, using batches will be slower than letting the Parallel package take care of those many threads to run. Since the Parallel package uses the ThreadPool package, it should reuse exist threads and not spend much time to create thread objects again.