Project 2

Gabriel Mongaras

October 16, 2024

1 Profiler System

1.1 Profiler

The profiler has three main objects within it:

1. std::map<char const*, std::vector<TimeRecordStart>> startTimes;
   This object maps section names to their corresponding start times. This allows for interleaving of different named sections as all start time vectors are independent based on the name. A map is used over other data structures as it easily separates sections by name, but it is also very fast and memory efficient as lookup time is constant and memory is just as large as the vector for each section.

2. std::vector<TimeRecordStop> elapsedTimes;
   Uses a TimeRecordStop object to record the section name and when it stopped. This vector stores every single time the ExitSection function was called. A vector is an efficient way to store all the times as it is easy and quick to add items (assuming it was preallocated to the correct size) and only takes up as much memory as the total number of section calls.

3. std::map<char const*, ProfilerStats*> stats;
   When the calculateStats function is called, all the stats are stored here. This map stores stats by section name.

The profiler has the following public methods:

1. void EnterSection(char const* sectionName);
   Takes a section name as input and records the start time in the startTimes map. Before the start time is calculated, the map is examined to make sure a vector for this section exists. If it doesn't it is created. This is done before the start time is calculated to prevent the allocation from being timed.

2. ExitSection(char const* sectionName); or ExitSection(char const* sectionName, int lineNumber, const char* fileName, const char* functionName);
   Takes the section name and optionally the line number, file name, and function name. It then pops off the time under section name from the startTimes map, calculates the elapsed time, and stores the time as well as the section name in the elapsedTimes vector.

3. void calculateStats();
   Calculates the following statistics for each section name:

   (a) count
   (b) total time
   (c) min time
   (d) max time
   (e) filename of end call
   (f) function name of end call
   (g) line number of end call
   (h) cumulative time
   (i) average time
   (j) cumulative time for every exit call

Note that this function is ok to be slow as it assumes that all the profiling is done when this function is called.

4. void reset();
   Resets all statistics and clears all stored times.

5. static Profiler* GetInstance();
   Get the instance of the profiler

6. void saveStatsToCSV(const char* filename);
   save all statistics to a CSV file of the given filename

7. void saveStatsToJSON(const char* filename);
   save all statistics to a JSON file of the given filename

Note that CSV and JSON were chosen as the formats to save data as these formats are most commonly used. JSON specifically is easy to read in with arbitrary vector sizes which is needed for the visualization tool. Specifically for the cumulative time on every exit call to show the cumulative time over exit calls.

Since a singleton is being used, one cannot create objects and instead must get the instance of the profiler via $Profiler* \ profiler = Profiler :: GetInstance();$

1.2 Easily interfacing the profiler with macros

The main class is the Profiler class which is interfaced by three macros after importing the *profiler.hpp* file:

1. $PROFILER\_ENTER(sectionName)$ is a macro that takes a string as input and tells the profiler to start profiling a section named $sectionName$ until told to stop. Under the hood this calls the $EnterSection(sectionName)$ in the profiler class.

2. $PROFILER\_EXIT(sectionName)$ is a macro that takes a string as input and tells the profiler to stop profiling the seciton named $sectionName$. It then stores information on the profiling done. Under the hood this calls the $ExitSection(sectionName)$ in the profiler class.

3. $PROFILER\_STATISTICS(sectionName)$ is a macro that returns the statistics of $sectionName$ as a $ProfilerStats$ object.

1.3 Example usage

These three macros allows the user to profile their code like the following:

```
PROFILER_ENTER("section 1")
...
$<$code to profile$>$
...
PROFILER_EXIT("section 1")
```

The profiler allows for nesting of ENTER statements like the following example shows:

```
PROFILER_ENTER("section 1")

for (int i = 0; i < 1000000; i++) {
    PROFILER_ENTER("Inner section")

    std::cout << i << std::endl;

    PROFILER_EXIT("Inner section")
}

PROFILER_EXIT("section 1")
```

Additionally, the profiler allows for interleaving like the following:

```
PROFILER_ENTER("section A"); // Enter section A
int a = 0;
for (int i = 0; i $<$ 1000000; i++) {
    a += i;
}
PROFILER_ENTER("section B"); // Enter section B
int b = 0;
for (int i = 0; i $<$ 1000000; i++) {
    b += i;
}
PROFILER_EXIT("section A"); // Exit section A
int c = 0;
for (int i = 0; i $<$ 1000000; i++) {
    c += i;
}
PROFILER_EXIT("section B"); // Exit section B
```

Once all profiling has been done, the user can calculate, print out all the statistics, and save it to a JSON or CSV file as follows:

```
...
$<$code to profile$>$
...

// Get the profiler
Profiler* profiler = Profiler::GetInstance();

// Calculate and print stats
profiler-$>$$calculateStats();
profiler-$>$$printStats();

// Save to CSV and JSON
profiler-$>$$saveStatsToCSV("profiler_test4.csv");
profiler-$>$$saveStatsToJSON("profiler_test4.json");
```

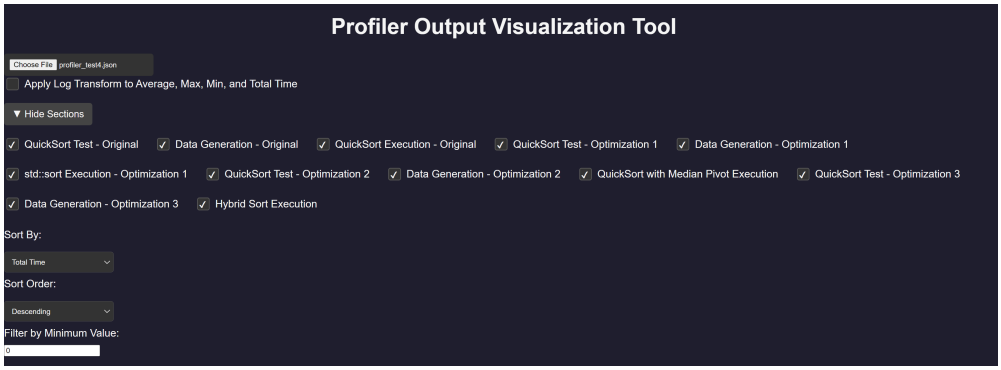

Figure 1: Website functions

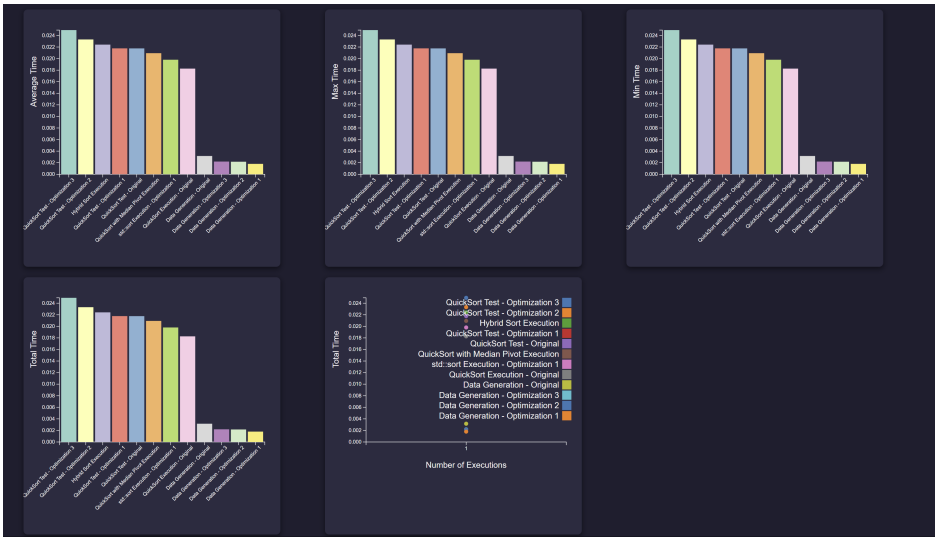

Figure 2: Wesbite Visualizations

2 Visualization Tool

The visualization tool is a web interface that allows the user to visualize the following:

1. The top of the website has a *choosefile* button. Select the output JSON file after running the program to display the results on the website.
2. The $ApplyLogTransformtoAverage, Max, Min, andTotalTime$ toggle turns the plots into log plots.
3. The buttons under the *hidesection* button allows the user to choose what sections names to display.
4. The *sortby* section is a dropdown that sorts by a specific metric.
5. The *sortorder* section is a dropdown that sorts by ascending or descending.
6. The $FilterbyMinimumValue$ section only displays sections that meet this value.

The following visualizations are on the website:

1. Average time
2. Max time
3. Min time
4. Total time
5. Cumulative time over time which shows that cumulative time over exit calls

3 Performance Analysis of QuickSort Profiling

3.1 Initial Algorithm

The algorithm being profiled is Quick Sort. Quick Sort is a recursive sorting algorithm which selects a pivot from the array and partitioning the other elements into two smaller arrays. These arrays are split according to whether they are less than or greater than the pivot or less than it. This happens recursively to each sub array. On average, Qick sort is $O(n \log n)$ in terms of time.

3.2 Performance Modifications

Three performance optimizations were applied to the original QuickSort algorithm:

1. **Using std::sort in place of the custom QuickSort**
   This first modification replaces Quick sort with C++'s quick sort which is much more optimized.
2. **Median Pivot Selection in QuickSort**
   This second optimization makes quick sort use the median-of-three pivot instead of always choosing the last element as the pivot. This reduces the chance of encodintering the worst case performance.
3. **Hybrid Approach with std::sort for Small Arrays**
   This third optimization is a hybrid approach use c++ sort if below 16 elements, otherwise it uses the custom quick sort algorithm.

3.3 Profiler Results

The following results were recorded for each modification:

3.3.1 Before Modifications (Custom QuickSort)

- **Total Time:** 1.70859 s
- **Min Time:** 0.0147629 s
- **Max Time:** 0.0241013 s
- **Avg Time:** 0.0170859 s

3.3.2 After Modification 1 (std::sort)

- **Total Time:** 1.72332 s
- **Min Time** 0.0163131 s
- **Max Time:** 0.0188778 s
- **Avg Time:** 0.0172332 s

3.3.3 After Modification 2 (Median Pivot Selection)

- **Total Time:** 1.6079 s
- **Min Time:** 0.0149334 s
- **Max Time:** 0.0193772 s
- **Avg Time:** 0.016079 s

3.3.4 After Modification 3 (Hybrid Sorting)

- **Total Time:** 1.62418 s
- **Min Time:** 0.0149003 s
- **Max Time:** 0.0224107 s
- **Avg Time:** 0.0162418 s

3.4 Impact of Each Modification

The median pivot made the largest impact. This is because it runs into less worst cases. Using the C++ library made it a little faster after using the median pivot, but not when using the last pivot. This is likely because std::sort does not use quicksort, but sometimes other sorting algorithms depending on the size.