Parallel Computing: Homework 5

Introduction

In this assignment, you are expected to parallellise the summation over a tree structure. In tree summation, we recursively sum the values of the sub-trees at a given node, and then add the value at the current node. In this manner, you will visit every node in the tree, and accumulate the sum in a recursive fashion.

Objective

When traversing a large tree, this sum procedure might take a long time. Parallelisation can help speed up this process. It is your task to implement the parallelisation, and evaluate how big its impact is in different scenarios.

On Nestor, we have provided two different tree structures: a basic binary tree, and a red-black tree implementation. The former should be familiar to everyone; the latter has been discussed during the course Advanced Algorithms and Data Structures. You can read more about the red-black tree here.

The tree structure and construction algorithm have been put in place, so you do not need to have in-depth knowledge of this implementation. Instead, we ask you to understand the properties of the tree and how these properties could explain the speed-up behaviour you will observe.

For your implementation, make sure to keep the following aspects in mind:

- The tree structure is recursive. How can OpenMP deal with such recursive structures? Which primitives should you use?
- OpenMP allows you to launch many tasks at the same time, potentially many more than the number of threads available. Think about whether, and if so, how you can limit the number of tasks OpenMP starts.

Execution

On Nestor, the implementations of both `rbTree.c` and `binaryTree.c` have been provided. They both share the same header file, `tree.h`, that is used by `execute.c` to perform operations on the tree. This way, the same code for `execute.c` can work with both the red-black and simple binary trees, depending on your compilation command.

The code can be compiled using the provided `Makefile`, by simply running `make` in your shell. It will build two binaries: `rbTree` and `binaryTree`, that should be called as follows: `./rbTree n`, where `n` is the number of threads to use, just like last week. The input, provided on `stdin` consists of two numbers: the number of nodes in the tree, and whether you want to have these numbers randomly generated (`0`), or sequential (`1`). This option affects the final tree structure, and thus, the overall performance.

Note that for large trees, the program can quickly run out of stack space. To combat this, `execute.c` tries to raise the stack limit, but this might not always work (for example, WSL1 does not allow raising it above 8MB). You can investigate the stack size limits on your machine using `ulimit -Ss` for the current limit (adjustable by the program), and `ulimit -Hs` for the hard limit (only adjustable by `root`). Peregrine (like most modern Linux systems) by default has no hard limit, so the program should work fine.

Deliverables

We ask you to submit your code to Themis, just like last week, which will only perform some basic sanity checks and will not extensively test the efficiency of your implementation. Themis will check that the output is still correct, ensure that the efficiency of serial execution is not hurt by the parallel implementation, and check whether you achieved at least *some* speed-up when running on 4 threads.

Note that you can sumbit ONLY `rbTree.c` or `binaryTree.c`, since Themis uses its own version of `execute.c` to evaluate your code. Consequently, your parallel implementation should be contained in `rbTree.c` or `binaryTree.c`. Furthermore, Themis will use the same `tree.h` header file, so you cannot make any changes to the signatures of the functions included in this file. Of course, you can change the implementation of these functions, and you can remove/add any other functions you need - as long as they are contained within `rbTree.c` or `binaryTree.c`.

Additionally, we ask you to submit (on Themis) a *short* and **concise** report (PDF, no more than 2 pages) outlining your approach to the parallelisation. This report should contain the following table, completed with the performance results you obtained on Peregrine. The LaTeX code for this table can be found on Nestor. Please note the following:

- The execution time should be reported in seconds (with no more than 1 decimal place).
- The speed-up and efficiency should be calculated relative to the execution time for 1 thread.
- The results can be (significantly) impacted by random variations in memory locality. For more stable results, you may want to run the tests a number of times and average the performance. In your report, describe your approach and elaborate on why this might be the case.
- The tree construction part takes the most time; that's why this part is excluded from the performance metrics as reported by `execute.c`. Constructing a tree can easily take 20 seconds, and only 1 second to be summed.
- Note that there is a difference between the 'Serial' and '1' column: for 'Serial' we ask you to run the *original, unmodified* code as provided, and for the '1' column we ask you to run your modified, parallelised code with 1 thread. If there are significant differences between these two, please elaborate on that in your report.
- In your report, briefly elaborate on any behavioural differences between the tests (sequential vs random, binary vs red-black) and try to explain where they originate.

Tree	N	Mode	Metric	Serial	1	2	4	8	16
Binary	20 000 000	`RANDOM` = 0	Execution time	X	X	X	X	X	X
			Speed-up	Y	Y	Y	Y	Y	Y
			Efficiency	Z	Z	Z	Z	Z	Z
Binary	10 000 000	`SEQUENTIAL` = 1	Execution time	X	X	X	X	X	X
			Speed-up	Y	Y	Y	Y	Y	Y
			Efficiency	Z	Z	Z	Z	Z	Z
Red-black	20 000 000	`RANDOM` = 0	Execution time	X	X	X	X	X	X
			Speed-up	Y	Y	Y	Y	Y	Y
			Efficiency	Z	Z	Z	Z	Z	Z
Red-black	10 000 000	`SEQUENTIAL` = 1	Execution time	X	X	X	X	X	X
			Speed-up	Y	Y	Y	Y	Y	Y
			Efficiency	Z	Z	Z	Z	Z	Z

Grading

Similar to the previous lab, we will grade on the critria *correctness*, *efficiency*, and *code style*:

Correctness corresponds to passing test cases on Themis. If you did not pass all test cases, you might still get partial points. A manual code check might override the results from Themis, e.g. if you hardcoded answers or circumvented explicit assignment instructions.

Efficiency correpsonds to the results you report in the performance table. We will perform a check of these performance numbers on Peregrine ourselves, so do not be tempted to alter them! Particularly efficient or sophisticated implementations might be awarded some bonus points.

Code style incldues, but is not limited to: sufficient documentation, clear naming of vairbales, proper code structring, proper code formatting, etc. *This is not an exhaustive list!*

Our grading will always be based on the **most recent** submission on Themis. Even though you might have submitted a 'better'(-performing) version before, we will only evaluate the most recent one.

This week there won't be exercises on Nestor.