University of Groningen

Parallel Computing

Homework 5: Tree traversing using OpenMP

Author:
Konstantinos Mazgaltzidis
(s5100453)

May 31, 2022

rijksuniversiteit
groningen

1 Introduction to the problem

Tree traversal refers to the process of visiting each node in a tree data structure, exactly once. The act of summation over a tree means to recursively sum the values of the subtrees. The summation can be performed in parallel at each subtree.

2 Algorithm parallelization

Inspecting the provided code in the binary tree (`binaryTree.c`) and in the red black tree (`rbTree.c`). Both of the tree have the same travese function.

We observe that we need to shift our focus to the function `computeSumSerial` that function computes recursively the sum at each node, a work that can be parallelized.

```c
1  long computeSumSerial(Tree root) {
2      if (root == NULL) return 0;
3      long val = root->value;
4      val += computeSumSerial(root->left);
5      val += computeSumSerial(root->right);
6      return val;
7  }
```

We can try to create new tasks at each function call (adding `#pragma omp task`) to line 4 and line 5), but this could result in stack overflow for large trees, because at each recursion level the number of threads will grow rapidly.

In order to combat the rapidly growth, a check has to be placed to not create more threads than they should be. Something extra we need to consider is the race condition that could occur with the variable `var`.

We can create a new recursive function `computeSumParallel(Tree root, int depth)` that recursive function will take the number of threads/depth and at each recursive level the depth will be decreased by one, when the depth reaches 0 that means no more threads can be spawned, so the serial implementation will be called.

For the race condition on the `var`, we can create a two new private variables and give them ot each thread `v1, v2`, in the end we just sum all of them.

3 Execution time

3.1 Methodology

Because of the nature of randomness that exist in the construction of the tree but also the fact that execution time can differ per run with the same input, I have decided to run multiple times the same code with the same arguments and average the result.

I wrote a small Bash script (5) to help me run the code 15 times and average the result.

3.2 Table

Tree	N	Mode	Metric	Serial	1	2	4	8	16
Binary	20 000 000	`RANDOM` = 0	Execution time	2.44	2.27	2.34	1.00	0.62	0.35
			Speed-up	0.9	1.0	1.0	2.3	3.7	6.4
			Efficiency	93%	100%	49%	57%	46%	40%
Binary	10 000 000	`SEQUENTIAL` = 1	Execution time	0.05	0.05	0.07	0.07	0.07	0.07
			Speed-up	1.0	1.0	0.7	0.7	0.7	0.7
			Efficiency	96%	100%	37%	18%	9%	4%
Red-black	20 000 000	`RANDOM` = 0	Execution time	1.23	1.18	0.71	0.54	0.32	0.25
			Speed-up	1.0	1.0	1.7	2.2	3.7	4.7
			Efficiency	96%	100%	83%	54%	46%	29%
Red-black	10 000 000	`SEQUENTIAL` = 1	Execution time	0.2	0.3	0.1	0.1	0.1	0.3
			Speed-up	1.2	1.0	1.9	2.9	5.0	0.9
			Efficiency	116%	100%	97%	72%	63%	5%

The raw data I used for the calculations can be found in the spreadsheet : `https://docs.google.com/spreadsheets/d/1IXw5xoXj-KqbTzxIbGnTTdjnxB7x3bcOkfq90En1uqY` *Some values of time of execution were putted with 2 decimal points because truncating them will result in loss of precision.*

From the results, we observe that the speed-up was not so great for the binary tree because it is easy for a binary tree to get into an array because it is unbalanced. But for the Red-black tree that is balanced based on some rules, so the parallelization is more balanced resulting in better speed-ups.

4 Behavioural differences

Sequential vs Random : When the tree is generated in sequence, $(1, 2, ..., n)$ the tree becomes a simple array that has only children on the right, and the parallelization is not balanced between the threads, resulting in poor parallelization performance. When the tree is generated randomly, the tree is more balanced and the parallelization is more balanced thus the performance is better compared with the sequential way.

Binary vs Red-black : The difference between a binary and a RB tree is that the latter is self-balancing to combat the tree becoming unbalanced. The binary tree can be easily become unbalanced, resulting in poor parallelization performance. But the RB tree that is self-balancing based on some rules, so the parallelization is more balanced, resulting in better performance.

5 Bash Code

```bash
#!/bin/bash
threads=1
size=20000000
mode=0
repetitions=15
for i in $(seq 1 $repetitions)
do
./binaryTree $threads 2> >(sed -n "s/^.*Execution time:\s*\(\S*\).*$/\1/p") << EOF
$size $mode
EOF
done
```