Subject: CAAM 420/520 – Homework 3
Date Posted: February 25, 2023
Date Due: 7:00pm, March 10, 2023 via Canvas

Total number of points available: (CAAM 420), (CAAM 520)

- **420 and 520:** *40 pts* All problems.

Problem 1 : OpenMP Tasks *(32 pts)*

Associated Files: `main.cpp, func_tree.h, func_tree.o`

Name your file: `<netID>_func_tree.cpp`. Copy the starter code in `func_tree.cpp` to your file.

Expected compile command:

```
g++ -o prob1 -std=c++11 -fopenmp func_tree.o main_prob1.cpp <netID>_func_tree.cpp
```

Running the program: `export OMP_NUM_THREADS=<num threads> ./prob1`

WARNING: do not modify `main_prob1.cpp` or `func_tree.h`.

Many software tools allow symbolic representation of arbitrary functions inputted by the user. Some examples of this are tools like Mathematica, WolfraAlpha, MatLab's Symbolic Toolbox, Maple, and MathCAD.

One way for representing functions symbolically on a computer is to use a data structure called a function tree. These trees tie the operations and operands of a function together in a tree structure. The *leaves* of the tree (nodes with no children) store the variable and constant values while all other nodes store operators. Figure 1 shows an example of a function tree and the function it represents.

Notice that the number of child nodes an operator node has equals the number of arguments that operator requires. Here, we are only going to allow operators that take at two arguments; this will allow us to represent our function trees as *binary* trees, i.e. trees where a given node can have at most two child nodes, called (creatively) the left and right nodes.

To evaluate a function that is represented as a function tree we can walk through the tree (called a traversal[1]) and propagate information upwards to accumulate the final result.

The tree representation of a function has the benefit of being parallelizable. This advantage would become more important the more times a function is evaluated.

(a) (3pts) What nodes can be processed at the same time in the function tree?

(b) (3pts) Where do you need synchronization in the evaluation of a function tree? In other words what operations must be completed before you can evaluate an intermediate answer at a given node?

(c) (12pts) Parallelize the evaluation of the function tree using OpenMP tasks. The files given (including the .o file) have all the necessary code and implmenation to read and build a function tree.

[1]The specific traversal we will do here is what is known as a postorder traversal.

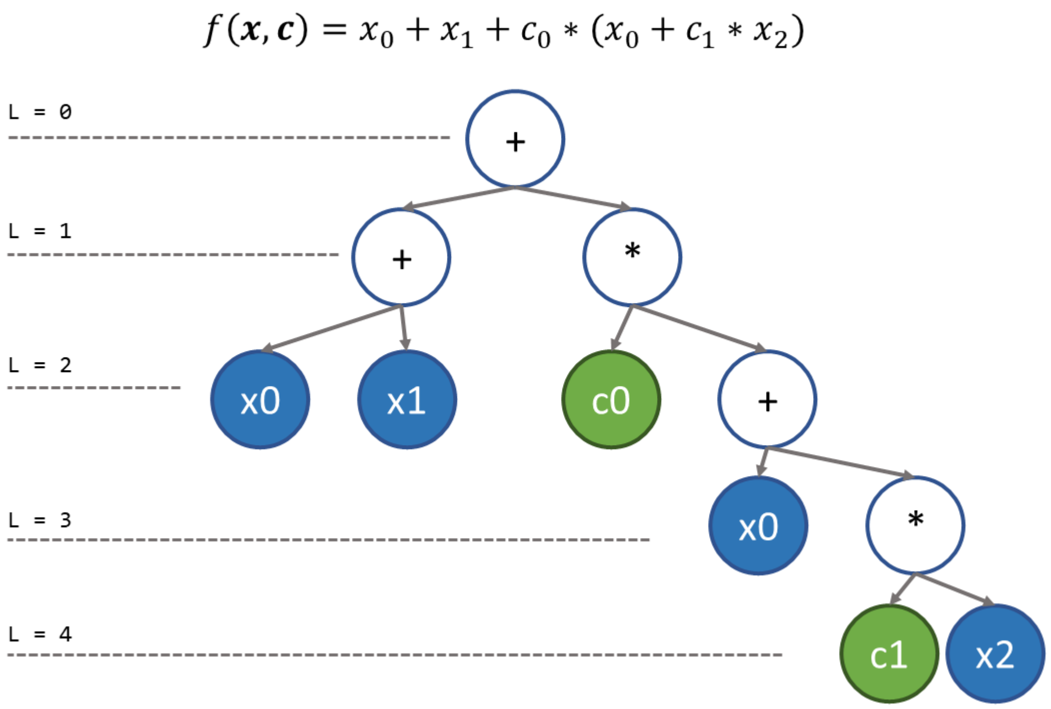
$f(\boldsymbol{x}, \boldsymbol{c}) = x_0 + x_1 + c_0 * (x_0 + c_1 * x_2)$
L = 0
L = 1
L = 2
L = 3
L = 4
+
+
*
x0
x1
c0
+
x0
*
c1
x2

Figure 1. Example of a function tree and the function is represents. The variable L represents the level of each set of nodes in the tree.

(d) (10pts) Assume the number of threads, N_T and the number of nodes in the tree, n, follow $N_T = 2^T < n = 2^m - 1$. One such case where $n = 2^m - 1$ is when our function tree represents a simple sum:

$$f(\boldsymbol{x}) = \sum_{i=0}^{m} x_i = x_0 + x_1 + \cdots + x_m.$$

Note that in this case, the tree is said to be *full*; every node is either a leaf (it had no children) or it has two children.

A simple sum can easily be parallelized using an OpenMP parallel-for loop. Doing so gives a parallel time complexity of $\mathcal{O}(m/N_T)$.

Find the parallel time complexity **without dropping terms or coefficients** of evaluating the sum using a function tree for the assumptions on N_T and n given. Report your answer in terms of n and N_T.

Hint: From the assumptions made on n, the number of nodes on a given level is exactly 2^L, with L as defined in Figure 1, and at $L = T$ the number of nodes on that level equals N_T. Assume an evaluation costs the same amount of time for every node and your synchronization is perfect (i.e. threads start and end their work at exactly the same time; you can neglect synchronization time). What is the parallel time complexity of evaluating a single layer? The cost of computing the whole tree is the sum of the costs of evaluating all its layers. Notice the problem has a spin-down phase; the cost of these layers must be special cased.

(e) (4pts) Compare your answer for the function tree's parallel time complexity with the complexity of the for-loop implementation. Which would be faster?

Notes:

- Note that in order to ensure correctness, you will need to synchronize tasks.
- You should only modify the function `evaluate_tree`.
- Notice `evaluate_tree` is called by a single thread but within a parallel environment. You do not need to introduce any more parallel blocks.
- **Supported operators**: The implementation of function trees given supports the operators -,+,*, /, and ˆ. To simplify the behind the scenes implementation of the function trees, note that if you want to negate a variable x_i, you must say $c_k - x_i$. with $c_k = 0$; you cannot say just $-x_i$. Additionally you must use `'*'` to denote multiplication, you cannot omit it.
- **To input a function:** variables must be given as `x#` where # is the index of that variable. You cannot name variables anything else. Variable indices are zero indexed, so # ≥ 0. Similarly, constants must be entered in the form `c#` where # ≥ 0. The values of the constants (if any ar present) is set once; you will be prompted by the program to set these.

  Example: $f(x, y) = x + 3(x + y) + 2 \implies$ `x0 + x1*c0(x0 + x1) + c1`.

Problem 2 : MPI Send/Recv On Multiple Ranks *(8 pts)*

Associated Files: `main_prob2.cpp, prob2.h`

Name your file: `<netID>_prob2.cpp`

Expected compile command:

```
mpic++ -o prob2 -std=c++11 main_prob2.cpp <netID>_prob2.cpp
```

Running the program:

```
mpirun -n <number of ranks> ./prob2
```

WARNING: do not modify `main_prob2.cpp` or `prob2.h`. For testing you can write your own main file if you like and compile your program using the same command as above with your main file in place of `main_prob2.cpp`.

In the function `group_exchange`, implement the sending and receiving of a single integer with a value equal to the sending rank's index from rank r to rank $(r+1)\%$ `num_ranks` using `MPI_Send` and `MPI_Recv`. You may not use the asynchronous send/recv functions here, and all ranks must participate in the communication. The routine needs to work for an arbitrary number or ranks without causing a deadlock.