OpenMP exercise: Binary tree traversal

October 29, 2012

1 Binary tree traversal

This programming assignment consists in parallelizing a binary tree traversal in topological order. The binary tree is built by means of the `TreeNode` data type defined as such

```
typedef struct treenode{
  /* Pointers to the left and right children */
  struct treenode *left, *right;
  /* The node number */
  int n;
  /* The node level in the tree */
  int l;
  /* A value associated to the node which is computed during the traversal */
  long v;
  /* Some data associated to the node which is used during the traversal */
  double *data;
} TreeNode;
```

The tree has l levels and, therefore, $2^l - 1$ nodes. The levels are numbered starting from the leaves and, thus, leaves are at level 1 and the root is at level l. Each node of the tree has a unique id n. The initial state of a tree with three of depth 3, before the traversal, is shown in the left part of Figure 1.

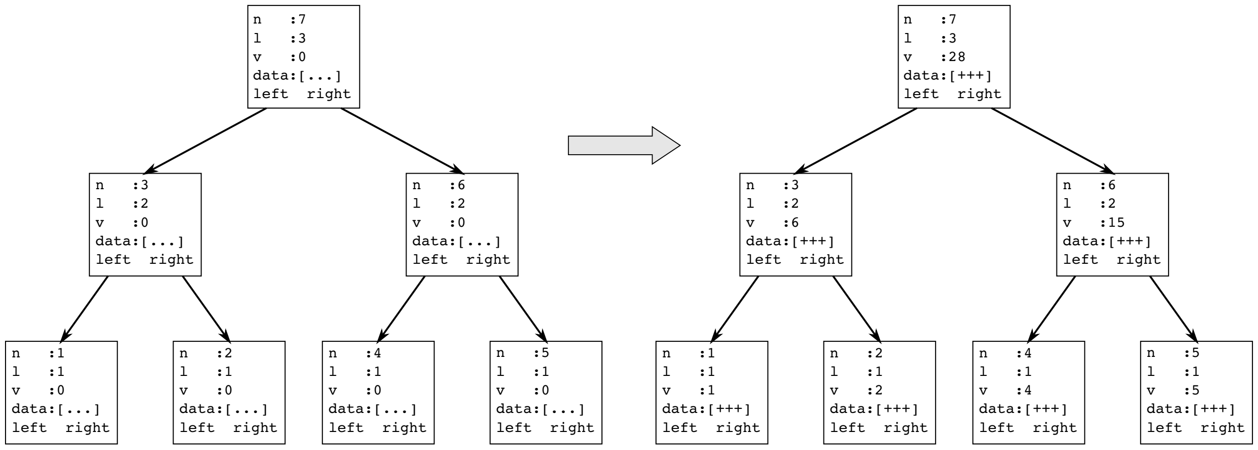
n :7 l :3 v :0 data:[...] left right
n :3 l :2 v :0 data:[...] left right
n :6 l :2 v :0 data:[...] left right
n :1 l :1 v :0 data:[...] left right
n :2 l :1 v :0 data:[...] left right
n :4 l :1 v :0 data:[...] left right
n :5 l :1 v :0 data:[...] left right
n :7 l :3 v :28 data:[+++] left right
n :3 l :2 v :6 data:[+++] left right
n :6 l :2 v :15 data:[+++] left right
n :1 l :1 v :1 data:[+++] left right
n :2 l :1 v :2 data:[+++] left right
n :4 l :1 v :4 data:[+++] left right
n :5 l :1 v :5 data:[+++] left right

Figure 1: The binary tree before and after traversal

The tree traversal consists in visiting all the nodes of the tree; each time a node is visited, the following operations are performed:

1. the data in the `data` array associated with the node is processed in some (unimportant) way;
2. the `v` variable associated with the node is set as such

```
root->v = (root->right)->v + (root->left)->v + root->n;
```

3. a counter is incremented in order to count the number of nodes visited by the current thread (the sum of the values computed by all the threads has to be clearly equal to $2^l - 1$).

Note that the value of `v` on a node, depends on the value of `v` on its children; this means that the tree has to be traversed in **topological order** (or bottom-up), i.e., a node can be visited only after both of its children.

The state of the tree after the traversal is shown in the right part of Figure 1.

The package contains the following files:

- `treetrav.h`: this file contains various declarations and should not be changed.
- `treetrav_seq.c`: this file contains sequential tree traversal routines. Precisely it contains the following two routines:
  - `void treetraverserec_seq(TreeNode *root)`
    This routine does the tree traversal of a tree defined by its root node in a recursive fashion.
  - `void treetraverse_seq(TreeNode *root)`
    This routine is just a wrapper and performs some initializations (sets the counters to zero) before calling the routine above.

  Besides, this file contains also tree initialize and destroy routines. This file should not be modified.
- `treetrav_par.c`: this file is meant to contain the parallel version of the tree traversal routines. Note that at the beginning the code in this file is an exact copy of the code in the `treetrav_seq.c` one. All the OpenMP directives must be inserted here. **This is the only file that has to be modified.**
- `main.c`: this is just the main program that initializes the tree, calls the sequential and parallel tree traversal and check the correctness of the results.

The main program can be compiled by typing `make` at the command line. This will generate an executable file called `main` that can be run as such:

```
./main l nthreads
```

where l is the chosen number of levels in the tree and nthreads the number of threads to use in the parallel tree traversal. Note that the number of nodes in the tree grows exponentially with the number of levels so be moderate: choose a small value to check the correctness of your parallelization (say, 5 to 10) and a bigger one to evaluate the scalability/performance of the code (say, 15 to 22/23). The main program will print on the screen the value of the `v` field of the root node (to verify that the sequential and parallel codes return the same value), the time taken by the traversals and the number of nodes visited by each threads in both the sequential and parallel cases.

2 Part 1: parallelization by task

Use the OpenMP `task` construct to parallelize the binary tree traversal.

Once the code is parallelized,

- run the main program and check that the parallel version returns the same final value for `root->v` as the sequential one.
- verify that a good load balance is achieved, i.e., that the number of nodes visited by each thread is roughly the same.
- analyze the performance and scalability of the parallel code. Is the parallel code faster than the sequential one? compare the execution time varying the number of threads. Keep in mind that your computer may have only two or four cores.

A naïve parallelization based on tasks is likely to give a slower code. How can you interpret this result? Think about the number of nodes in the tree and the number of working threads.

3 Part 2: reducing the task scheduling overhead

In the cases where it is worth parallelizing the tree traversal, the number of nodes in the tree is much larger than the number of working threads. Considered that in Part 1 we created a task for each node in the tree, do we really need all these tasks? Is it likely that this enormous number of tasks overloads the scheduling system and slows don the execution?

Think about a case where the number of generated tasks is limited depending on the number of working threads. This can be achieved through the `if` condition of the `task` construct by identifying a layer in the tree such that nodes above the layer are treated in deferred tasks and nodes below are treated in undeferred tasks.

Implement this version and redo the tests done in Part 1 and see if anything has changed.