INF-3201 Parallel Programming Shared Memory

Enrico Tedeschi
ete011@post.uit.no

I. Introduction

The goal of this assignment is to parallelize a piece of code using shared memory techniques preferably using **OpenMP**.

The piece of code to parallelize could be the given one, that is a Markovian chess engine, or a chosen piece of code which has to be approved by the professor.

A. Requirements

- Choose a piece of software to parallelize
- Parallelize it with shared-memory techniques
- Evaluate speed up

II. Technical Background

- Concurrency and parallelism concepts
- Parallel programming concepts
- Basic programming approach
- Knowledge of C language
- Notion of design pattern principles
- Theory about software engineering
- Knowledge of git to manage the software versions

III. Analysis

The given **Markovian** code was found to be too long and quite hard to understand, in addition, it has no deterministic solution so it would have been really difficult to test.

The code chosen to be parallelized is a sequential version of the **TSP** (Traveller Salesman Problem). It asks the following question: *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?* [2]

The sequential code was refined by adding a random function which generate distances between cities and also the possibility of run the programme with a size passed as a parameter was implemented.

To parallelize a sequential code is necessary to analyse which function involves the biggest amount of time. The c given program *tsp-seq.c* has been tested with *profile* and the following table is the result of this test:

```
  %     cumulative    self
 time     seconds     calls     name
100.32     2.90          1      zRoute
  0.00     0.00        196      zEuclidDist
  0.00     0.00         28      random_at_most
  0.00     0.00          2      second
  0.00     0.00          1      zReadRoute
```

the function to parallelize is obviously *zRoute*.

The function has a recursive structure for each possible successor of the city chosen as initial. For instance, if the function would have a $iSize = 5$ (number of cities) then the execution tree would look like the one in Fig 1. In the first 'level' of the tree the first city executes recursively the *zRoute* function for the number of the cities left to visit, and so for the other levels. Having an image of the execution tree helps to get the parallelization easier.

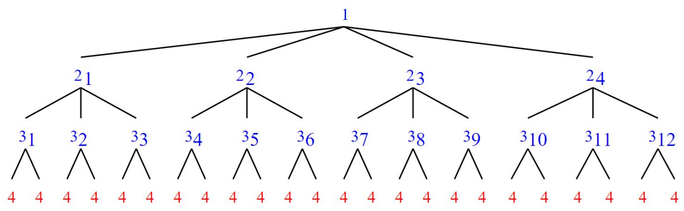

Fig. 1. Execution of the zRoute function

IV. Implementation

A. Environment

The code has been developed using JetBrains CLion 1.1 on Windows 10 and due to a more easy way to compile and test the MPI programs with a Linux based console, the compilation and the execution of the code has been made on an Ubuntu Virtual Machine with 4 core assigned, using VirtualBox 4.2 and Ubuntu version 14.04.3

To synchronize the cluster and the local machine and to keep trace of all the changes in the code, a git repository

was created and the git command on linux were used to commit and to push/pull data from repositories.
The benchmarking and the test with more than 4 cores has been done in the UiT uvrocks cluster. The connection with the cluster was established by using ssh on a linux machine:

```
ssh -X -A ete011@uvrocks.cs.uit.no
```

To simplify the execution of the code, a proper Makefile was implemented.

B. Fixing the Sequential Version

Before starting the parallelization some changes to the sequential version were implemented. The way to verify if the solution found is reliable is to check the *length* of the path to visit all the cities. The length of the path changes if either the number of cities or the size of the map is changed, but it's not dependent from where the cities are located. Considering that, a function which generates random numbers from *0 to max_size* is created, so every time, given a number of cities and a max_size defined, is possible to generate random coordinates for the cities which won't influence the correct *crc* anyway. The sequential version, *tsp-seq.c*, takes as input the number of cities and it has a fixed map size.

C. Approaching the Problem

Due to the tricky recursive function the first goal of the implemented parallelization was: "get the speed up keeping the code simple". The Fig 2 shows the logic used for the parallelization. The parts of the tree enhanced in red are the component which will run in parallel; in that way the speed up is strongly dependent from the relation between the $number\_of\_cores$ and the $number\_of\_cities$. Using this kind of approach the parallelization is made in the first for cycle, and each core will execute the recursive function, solving all the sub-branches problems related to it.

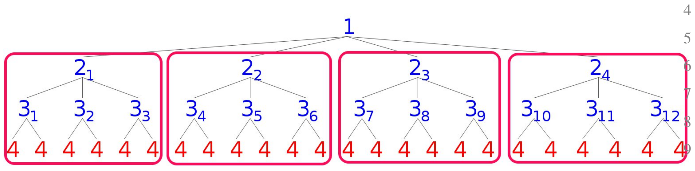

Fig. 2. Execution of the parallel version of the zRoute function, with 5 cities using 4 parallel cores

D. Parallelization

For the parallelization **OpenMP** API were used. They provide to standardize directive-based multi-language high-level parallelism that is performant, productive and portable[1]. Since the *zRoute* function is recursive is necessary to define two types of *zRoute* functions, a **sequential** one and a **parallel** one. If the code is in the first level of the tree, then it will call for each sub-branches the recursive parallel function, otherwise it will call the sequential one.
The biggest problem to solve was related to shared memory and how the variables were used while parallelizing the code. OpenMp indeed takes care about the balancing of the work but is necessary to have well defined which variable needs to be global and which ones local.
To parallelize the function the following logic was used:

```
count = 0;
void zRoute(...){
    if (count == 0) {
      count++;
      #pragma omp parallel num_threads(#)
      {
      #pragma omp for
      --core of zRoute function--
      }
    }
    else{
      --core of zRoute function--
    }
    return;
}
```

To keep the variables local, a strong analysis on them has been done. All the variables which change their value during the execution (called var_c in the listing below) of zRoute must have been declared locally. To achieve that, a function called *for_cycle_zRoute* has been created and it is called only when $count = 0$, that is when the code needs to be parallelized. Therefore the function it receives as input all these variables which need to be locally declared and it executes the core of the zRute function.

```
void zRoute(...){
    if (count == 0) {
      #pragma omp parallel num_threads(#)
      {
        for_cycle_zRoute(var_c);
      }
    }
    else{
      --core of zRoute function--
    }
    return;
}
```

The logic for the *for_cycle_zRoute* function is:

```
void for_cycle_zRoute(var_c){
    --definition of local variables--
    #pragma omp for
    --core of zRoute function--
    return;
}
```

V. Result and Benchmarking

For the benchmarking the parallel

VI. Discussion

VII. Conclusion

References

[1] Openmp. http://openmp.org/wp/about-openmp/.

[2] Wikipedia - tsp. https://en.wikipedia.org/wiki/Travelling_salesman_problem, 2015.