Titolo appropriato

Anna Bonaldo Vassilikì Menarin

Email: anna.bonaldo@studenti.unipd.it vassiliki.menarin@studenti.unipd.it

***Abstarct* - Write something interesting here**

1 Introduction

1.1 Work Staling Scheduler

Work stealing is a scheduling strategy designed to efficiently manage a dynamically multi-threaded computation using a fixed number of processors. The work stealing scheduler bounds the number of concurrently active threads within a limit, thus affecting the memory requirements too. This causes an improvement in the execution time and memory usage. Moreover, the scheduler tries to mantain related threads on the same processor, minimizing the communication between different threads. Work stealing differs from work sharing because underutilized processors take the initiative, without relying soley on the scheduler: each processor owns a deque and, when it-s empty, it tries to steal work from others. The number of migrations is lower in work stealing, because it is done only when absolutely necessary.

We built a work stealing scheduler that runs a Quick Sort application. We performed some analysis on the results, comparing the clock-time and CPU time for a work stealing and a sequential scheduling, searching for a set of paramentrs that minimizes wall-clock time.

2 Our work

We developed a work-stealing scheduler and a Quick-Sort application using Java 8. The main routine takes three parameters, and we tried assigning different values to them to find a set that minimizes wall-clock time when compared to sequential sorting. These parameters are:

- `arraySize`: sets the size of the array tp sort;
- `numServers`: sets the number of servers;
- `cutOff`: sets the array size under which to perform sequential sorting and it is used to improve the performances of the work stealing scheduler.

The classes that implement the algorithm are `Scheduler`, `Tasklet` and `QuickSort` We then use `BatchQuickSortExecutors`, `IOFromCSVFile` and `Statistics` to efficiently gather and compute data. Details on the analysis performed can be found in Section 3.

2.1 `Scheduler`

The work stealing scheduler is implemented using the `WorkstealingScheduler` class, which implements the `Scheduler` interface. The `WorkstealingScheduler` class contains a private `ServerThread[]` array named `servers`. The array represents the number of servers available. `ServerThread` is private inner class.

`ServerThread` Each server is represented using a `ServerThread`. Each server has the fields:

- `private int myIndex`: the identifier of the server;
- `public ConcurrentLinkedDeque<Tasklet> deque`: each server has a deque to store the tasklets it needs to run. When dealing with its own deque, a server always adds and polls from the bottom of the deque;
- `private Stack<Tasklet> stack`: the stack stores the Tasklet the server is currently running.

The `ServerThread` class implements `Runnable`, and its two most important methods are:

- `public void run()`: defines the standard behaiour of the server. If the server's deque is void, the server calls the steal method; otherwise it polls the last Tasklet from the bottom of the deque, pushes it into the stack and then executes it;
- `public void steal()`:hagving its own deque empty, the server tries stealing Tasklets from the top of someone else's deque. If the steal is successful the server may proceed with the Tasklet execution.

2.2 `Tasklet`

`Tasklet` is an abstarct class designed to recursively perform a task. We mainly use the `QSortTasklet` class.

`QSortTasklet` This class was designed to recurively perform Quick Sort.Its most notable fields are:

- `public final int start, end` which are necessary to perform Quick Sort;
- `private ConcurrentLinkedDeque<Tasklet> originDeque` keeping a reference to the deque the Tasklet was piqued from we can add one there one child while we execute the other;

As for its methods, since `QSortTasklet` extends `RecursiveAction`, it implements `compute()`.

- `public void compute()`: it's the main method of the class. It checks the size of the array or portion of array to sort and it confronts it with `cutOff`. If the size exceeds the cutoff we recursively call compute, otherwise we sort the remaining array sequentially.

2.3 The QuickSort class

It contains the `main` and the methods necessary to perform Quick Sort, `public void sequentialSortArray (int[] array, int left, int right)` and `public int partition (int arr[], int left, int right))`.

3 Analysis performed

3.1 Results of analysis

4 Conclusion