Multicore Programming

Project Java Fork/Join: An Exact Solver for the Traveling Salesman Problem

Assistant: Steven Adriaensen
Mail: steadria@vub.ac.be
Office: 10G711 (CoMo lab)

Deadline: 10 June 2016

For the final programming project, you will implement a parallel exact solver for the Traveling Salesman Problem (TSP) using Java Fork/join. The Traveling Salesman Problem can be formulated as follows: Given a set of cities and the distances between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the origin city. In general, solving TSP exactly involves an exhaustive enumeration of all possible tours. As even for relatively small instances, there are many possible tours, an efficient parallelization of this search is of paramount importance.

1 Procedure Overview

This project consists of three parts: The parallel implementation of an exact solver for the Traveling Salesman Problem using Java Fork/Join, an evaluation of its performance and a report that describes both implementation and evaluation.

Deadline 10^{th} of June at 23:59 CEST. The deadline is fixed and cannot be extended.

Deliverables Package the implementation into a single ZIP file, including the report as a PDF. The ZIP file should be named `Firstname-Lastname-jfj.zip`.
On the PointCarré page of the course, http://pointcarre.vub.ac.be/index.php?application=weblcms&go=course_viewer&course=3092, go to *Assignments (Opdrachten)* > *Project Java Fork/Join* and submit the ZIP file.

Grading This project will account for one third of your final grade. It will be graded foremost based on the *quality of the report* and the *project defense* at the end of the year. However, the *code quality* will be taken into account too. So, please comment your code where necessary to guide the reader.

2 The Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a combinatorial optimization problem, which can be formulated as follows: *Given a set of n cities and the distances between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the origin city.* In particular, we'll consider the (2D) Euclidean TSP problem, where each city is assigned a 2-dimensional coordinate and inter-city distances are Euclidean.

More formally, let $C = \{c_0, c_1, \ldots, c_{n-1}\}$ be a set of n cities, let $L \subset \mathbb{N} \times \mathbb{N}$ the set of locations $l_i = (x_i, y_i)$ of each city c_i, and let a tour t be a correspondence $\{1, 2, \ldots, n\} \to C$, where $t(i)$ is the i^{th} city visited in the tour. Find a tour $t^* = \arg\min_t (d(l_{t(n-1)}, l_{t(0)}) + \sum_{i=1}^{n-1} d(l_{t(i-1)}, l_{t(i)}))$, where $d((x_i, y_i), (x_j, y_j)) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

Instances of the TSP as formulated above, appear in many practical settings, such as planning, logistics, and the manufacturing of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. However, no efficient (polynomial time) algorithm solving them is known or is likely to exist (TSP is NP-hard). Instead, exact solvers perform an exhaustive search, enumerating all possible tours (i.e. the search-space) in a structured way. Even for relatively small instances, this search space is immense (e.g. for $n = 13$, 239500800 tours exist). Luckily, disjoint areas of the search-space can be explored in parallel and an efficient parallelization of this search is therefore of paramount importance to solve these problems in practice.

3 Implementation

In your implementation you are to extend the project code provided alongside this assignment. It provides an implementation of the traveling salesman problem class, allowing you to generate random TSP instances with given n. In addition, we provide you with `SequentialTSPSolver`, a sequential implementation of a simple exact TSP solver. The solver recursively constructs all possible tours and returns the one that minimizes the total distance travelled.

In this assignment you are to implement `ParallelTSPSolver`, a parallel version of the `SequentialTSPSolver` using Java Fork/Join. This implementation should have the following properties:

- All unit tests provided should succeed, i.e. it should be correct.
- It should be functionally identical to the `SequentialTSPSolver`, i.e. given the same problem `ParallelTSPSolver` should return the same solution as its sequential version. The unit tests provided (to some extent) verify this property.
- The efficiency of the implementation is important, use the fork/join optimizations seen in class and avoid expensive operations (e.g. dynamic memory allocations, frequent non-cached memory accesses, ...) that do not occur in the sequential implementation. On the other hand don't micro-optimize in a way that would hurt the code's readability.

- The span of the computation should be logarithmic w.r.t the number of tours (for any problem instance), i.e. you should recursively split work in 2 as seen in class.
- It should implement a sequential threshold with the following properties:
  - If a task's size is below the threshold it should be computed sequentially using the code provided in `SequentialTSPSolver`.
    (Hint: re-use the static `find_optimal_tour` method).
  - For a threshold value of 0, each task should evaluate at most 1 tour.
  - For a threshold value of $+\infty$, only a single task should be created.
  - Increasing this threshold should gradually (and monotonically) lead to more coarse-grained tasks.

If you are unsure your implementation fulfills these requirements or if they're unclear, feel free to contact Steven Adriaensen.

4 Evaluation

The main goal of your experiments is to evaluate the performance of your parallel implementation. To guide you in this process we ask you to do some performance measurements and answer some performance related questions. We expect you to provide us with sufficient experimental results to motivate your answers. We offer a lot of freedom in how you perform/design these experiments and present the results. However, we do expect you to do so in a proper scientific way. Below we give some guidelines.

To properly measure the parallelism of the implementation, experiments should be performed on a multi-processor machine with at least 4 *physical* cores. If you do not have such a machine at your disposal, you can use one of the lab computers to perform your experiments. For the sake of comparison we ask you to use the same machine and JVM for all experiments and describe as much as possible the details of these in the report.

Before running any experiments make sure your implementation is correct, i.e. all unit tests provided pass. Also be aware of common benchmarking pitfalls: Does your CPU support features like Intel's Hyper-Threading[1] or Turbo Boost[2]? Also, avoid running any processor (or memory) intensive applications alongside your experiments as this might (strongly) influence their outcome. Furthermore, take into account that accurately benchmarking Java applications is challenging as many advanced features of the JVM affect performance[3]. You are free to decide which benchmarks you use, but specify the details of the benchmark used

[1] On processors with Hyper-Threading, several (usually two) hardware threads run on each core. E.g, your machine might contain two cores, which each run two hardware threads, hence your operating system will report four "virtual" (or "logical") cores. However, you might not get a 4× speed-up even in the ideal case.

[2] Turbo Boost allows your processor to run at a higher clock rate than normal. It will be enabled in certain situations, when the work load is high, but it is restricted by power, current, and temperature limits. For example, on a laptop it might only be enabled when the AC power is connected.

[3] In general JVM performance will be worst in the beginning and stabilize after some time (known as the JVM warm-up). The use of third party benchmarking tools (such as Caliper, JMH) is optional. However, if you do, make sure you understand how they work.

in each experiment (i.e. n). While parameters should make the problem sufficiently hard, also take into account that experiments must be run multiple times (to measure variability) and be completed by a fixed deadline. E.g. A single run of the solver should probably take somewhere between a few seconds to a few minutes on your machine.

Report results appropriately: Take into account variability, i.e. report mean/median performance over multiple runs and measure this variability (e.g. best/worst performance, standard deviations, confidence intervals). Graphical representations are preferred over large tables with many numbers. Interpret the results: make clear what the diagrams depict, and explain the results. Did you expect these results? If the results contradict your intuition, try to explain what caused this. Where possible, refrain from making wild conjectures, perform additional experiments to motivate your explanation (e.g. if you think an observation might be related to Intel's Hyper-Threading being enabled, disable hyper-threading and compare the results).

There are multiple metrics you can use to measure the efficiency of your implementation. While execution time is the most straightforward, it is often not the most meaningful. Especially when evaluating parallelism we strongly advise you to use relative metrics such as speedup instead. Also, make clear whether you use application or computational speedup: The former measures the speedup compared to a sequential implementation ($\frac{T_{seq}}{T_p}$), the latter to the parallel implementation using only a single worker thread ($\frac{T_1}{T_p}$). The ratio of both speedups gives a measure for overhead ($\frac{T_1}{T_{seq}}$) in the parallel implementation.

5 Reporting

Please follow the outline below for your report and clearly answer the questions:

1. Overview: In this section you briefly summarize your overall implementation approach, and the experiments you performed. (max. 250 words)

2. Parallel Partitioning Approach: In this section you discuss and evaluate your parallel fork/join implementation without sequential threshold ($= 0$).

2.1 Implementation:

Briefly describe the strategy you use to divide the work and accumulate the results. (max 250 words)

For $n = 5$, provide a single graphical illustration showing the execution order dependencies between

- Insertions (calls to `Tour.visit`).
- Forks (creates a new task)
- Joins (causes 1 task to wait for the completion of another)

grouped per task (where only a fork creates a new task).

Hints:

- Use a graph representation, where nodes represent instructions and edges represent execution order dependencies.
- Fork instructions have an out-degree of 2, Join instructions an in-degree of 2, all other instructions have an in- and out-degree of at most 1.
- Group nodes belonging to the same task by encircling them.
- There should be 18 insertions and # forks = # joins = # tasks-1 ≥ 11.

2.2 Properties of the Computation:

Give exact mathematical expressions in function of n for the following properties of the resulting computation:

- Total # insertions performed ($\sim$*work* of the computation)
- The # insertions performed on the critical path ($\sim$*span* of the computation)
- The average parallelism (ratio of *work* over *span*).
- Total # tasks scheduled (# forks + 1)
- The average # insertions per task ($\sim$task granularity).

Hints:

- It might help to draw dependency graphs (as in 2.1) for several n.
- It might be easier to derive 2 formulas, one for $n < 4$ and another for $n \geq 4$.
- Use factorials $n! = \prod_{k=1}^{n} k$ to write your formula more compactly.
- In doubt, validate your answers experimentally (e.g. count #forks, insertions)

Now, plot these functions for different values of n. Briefly describe the trend you observe.

Measure the execution time of the sequential implementation for different values of n. Is it reasonable to measure work in terms of # insertions? Discuss.

2.3 Evaluation:

Evaluate the performance of your fork/join implementation.

- Compute the overhead ($\frac{T_1}{T_{seq}}$) of your implementation for different values of n. Discuss the results, focus on answering the following questions: How efficient is your parallel implementation? What is the overhead not present in the sequential implementation? Is this acceptable? What could you do to reduce this overhead? How does the overhead relate to n?
- Compute the speedup achieved by your implementation for different values of P (fix n sufficiently large!). Also compute the *efficiency*: $\frac{T_1}{PT_P}$. Discuss the results, focus on answering the following questions: How well does your parallel implementation exploit the parallel processing capabilities of your machine? Wat does *efficiency* measure? What if your machine had more cores, would it scale? Is your implementation an improvement over the sequential one?

3 Sequential Threshold: In this section you discuss your implementation of the sequential threshold and how it influences performance.

3.1 Implementation:

Briefly explain how you've implemented the sequential threshold. (max 250 words)

3.2 Properties of the Computation:

Update the mathematical expressions in Section 2.1 to include the effect of the sequential threshold T, i.e. you should derive mathematical expressions in function of n and T. (Hint: for $T = 0$, both should be equivalent).

Plot the average parallelism and average task granularity for different values of T (fix n sufficiently large!). Briefly describe the trend you observe.

Note: Depending on how you've implemented the sequential threshold, deriving exact expressions may be difficult. If so, you may derive and plot (tight) upper/lower bounds and/or empirical measures instead.

3.2 Evaluation:

Evaluate how the sequential threshold value influences the performance of your fork/join implementation.

- Compute the overhead ($\frac{T_1}{T_{seq}}$) of your implementation for different threshold values. How does the threshold value affect the overhead of the parallel implementation?
- Compute the computational and application speedup achieved by your implementation for different threshold values. What threshold value would you pick? Why not a higher or lower value? Is your implementation an improvement over the sequential one?

(Hint: Include average task-granularity and/or parallelism in these discussions).