Programming Assignment
Multi-core nested depth-first search in Java

Introduction
In this assignment, you will implement a multi-core nested depth-first search algorithm (MC-NDFS). Sequential NDFS is a memory-efficient algorithm with optimal linear time complexity for detecting cycles in a graph. It relies on the so-called post-order of the depth first search for its correctness, which makes it inherently sequential and challenging to parallelize.

One solution to implementing a multi-core NDFS algorithm is presented by Laarman et al. [2]. They show how information from the nested search phase can be shared between concurrent threads executing an NDFS search, reducing the amount of duplicate work done by different threads. This paper will be the basis for your MC-NDFS implementation in Java.

The goal of the assignment is to familiarize yourself with Java’s concurrency constructs, to learn to reason about correctness of a concurrent program, to gain some experience in designing and implementing concurrent code, and to discover some common trade-offs between synchronization and performance.

Prerequisites
First, decide whether you want to do the assignment alone or with someone else (you are encouraged to work in pairs). Then, you need to register for the assignment. You can do this by sending an email to c.j.h.jacobs@vu.nl. Include your name, VU-net-ID and student-number, and those of your partner if applicable. Make sure to indicate whether you are a Master or Bachelor student and whether you already have a DAS4 account. We’ll supply one if you don’t already have one.

On the course website at http://www.cs.vu.nl/~tcs/cm/ you can find the papers referenced by this document and a framework for your implementation.

You will be using the standard compute nodes of the DAS4 cluster to test and evaluate your solution. Access to the clusters requires a DAS4 account, see above.

During development you can also use your own computer, but the implementation that you submit should compile and run on DAS4. In order to develop on your own computer, make sure it meets the following requirements:

• A Java JDK version 1.7 or higher is installed. You need a JDK (Java Development Kit), a JRE (Java Runtime Environment) is not sufficient.
The program ant is available to build your implementation.

Multiple cores are available for testing.

Also keep in mind that different operating systems can have different performance characteristics, so be sure to double-check your intermediate results on a DAS4 node. Final measurements for the evaluation part of the assignment should always be done on a DAS4 node.

Note that this can be a very challenging assignment. Please do not hesitate to ask questions via email. It is also possible to make an appointment.

Details of the assignment

Background

Finding cycles in a graph plays an important role when verifying software through model-checking. The goal in model-checking is to ensure that a program never violates certain temporal properties. Think for example of a safety property (e.g. a response can only follow after a request) or a liveness property (e.g. after a request, a response always eventually follows). Often it is sufficient to prove these properties for an abstract model of the program.

One model-checking approach is to generate a state-space of the model under investigation: a graph in which nodes represent states (memory snapshots), and edges represent transitions (statements) of the model. The graph contains all possible states which can be reached by executing the model. Similarly, one can create a state-space for the negation of the temporal property being checked: a graph in which nodes represent partial counter-examples (executions of a model), and edges represent propositions (checks on a model state) for the property. This second graph represents all executions, for any model, which would violate the property. Combining both state-spaces allows one to search for counter-examples in a specific model.

The graphs you will be using in this assignment represent such a product state-space of both the model and a negated property. Depending on the temporal property being verified, one either wants to show that certain states are unreachable (a safety property), or are not on a cycle (a liveness property). These states of interest are called accepting states. Reaching such a state, or finding a cycle through one, means that this specific execution is accepted as a counter-example. The program violates the property if a counter-example can be found.

For the assignment we will be looking at counter-examples to liveness properties. We are only interested in searching for accepting states on a cycle, which in turn can be found using a nested depth-first search (NDFS) algorithm. Your implementation of a multi-core NDFS algorithm will be based on the work of Laarman et al. [2]. For some additional information on NDFS you can read the papers of Courcoubetis et al. [1] and Schwoon et al. [3].

If you don’t understand the above, don’t worry. In the end, the assignment is about graphs with states that are either accepting or not, and the detection of cycles containing accepting states in that graph.
Overview
After studying the assignment and referenced paper [2] you should have a general understanding of the problems involved in parallelizing an NDFS algorithm. The paper contains three algorithms: Alg. 1 is the sequential algorithm, Alg. 2 is the multi-core algorithm without extensions, and Alg. 3 is the multi-core algorithm with some extensions. A framework is provided, which already contains an implementation of Alg. 1.

The first part of the assignment, for all students, is to design and implement the naive version of the MC-NDFS algorithm (Alg. 2 in the paper). This version should use a very naive form of synchronization only; for example, you could use a global lock on your data structure(s). Do not do anything more complicated than this, because it will be used as a baseline to compare your improved version(s) to in the evaluation. You must implement the algorithm exactly as described in the paper (don’t miss anything out); make sure you read the paper carefully. Master students, in addition, are also required to implement Alg. 3 (with allred and early cycle detection enabled).

The second part of the assignment is to design and implement one or more improved versions. Bachelor students are required to implement a single improvement. Master students should implement four improvements, all of them based on their Alg. 3 implementation.

Under no circumstances should any of these try to improve on the algorithm itself. The improvements should rather come from for instance more clever ways to access the shared data, or other ways to improve the efficiency of the concurrent versions.

Each student should write a design document that discusses the challenges and solutions they use when implementing these different versions. This document should describe both the naive version (that will act as baseline) and the improved version(s) that you will implement.

Finally, the performance of the different versions must be evaluated, with a focus on scalability.

Design
Your design document should describe how you translate the MC-NDFS algorithm to an actual Java implementation. In addition, the design document should discuss versions that improve on this naive version. Some examples of questions we think you will need to consider (about both your naive and improved versions) are:

- What data needs to be shared, and what doesn’t?
- Which data structures are you going to use? What are the advantages and disadvantages of these?
- How do you keep track of visited states?
- How are you going to prevent threads from searching the same area of the graph?
- When and where do threads need to synchronize? The paper states that lines of pseudo-code are executed atomically. How does this influence your implementation?
• Based on the algorithm, how well do you expect your implementation to perform? What kind of graphs would the algorithm perform well in? In which would it perform less so? What about your naive version? And your improved version?

• How will you terminate the search and your program?

This list is not meant to be exhaustive, but your design should at least cover all of these areas. There is a good reason that we ask each question, and you should make sure to think carefully about each one! Your design should indicate a good understanding of the problem, and show that you have a sound plan for implementing the algorithm in Java. We want you to really think through your implementation before you actually start programming.

Implementation

Your implementation will extend the skeleton framework provided with the assignment. The framework contains two copies of the sequential NDFS algorithm (Alg 1): one to keep as a reference (nddfs), the other to modify into your own MC-NDFS algorithm (mcndfs naïve). You can extend the framework in a similar fashion when implementing your own version(s), e.g. (mcndfs ò <optimization>).

Some tools you might want to use to debug and/or measure your implementations are "jvisualvm", "jconsole", "kill" and "top". "Jvisualvm" can be used to profile your code and is part of the Java 1.6 and later Sun/Oracle JDK’s. "Jconsole’ allows you to monitor your java processes for performance and resource consumption. The "kill -3 <pid>" command can be used to send a QUIT signal to the java process with process id <pid> and print its stack trace. "Top" shows you the processes running on the system.

You can test your implementation either locally or on the DAS4 cluster. Running jobs on the DAS4 can be managed with the prun utility. You first have to make the prun command available, using the module command:

module load prun

It is probably best to put this in your .bashrc script.

The following commands can then for instance be used:

prun -np 1 <executable>

Checking the current node reservations can be done with:

preserve -llist

The skeleton provides a script "bin/ndfs" which runs your application either locally or on a DAS4 node. For more information on the DAS4, see http://www.cs.vu.nl/das4/. It is a shell-script, so you can examine it to see what it does.

Evaluation

After the implementation phase, you should write an evaluation of the performance of your versions with a focus on scalability. Support your findings with measurements and present them using tables or graphs. We want you to show a thorough understanding of why you obtain a certain result. Make sure you support your reasoning with measurements.
We expect you to adjust the parameters of the input files to get meaningful results. Discuss your obtained results in relation to the input models that you use.

Do your measurements live up to what you expected in the design phase? It is not an issue if an optimization turns out to be slower. However, we expect you to explain this. We want to emphasize that being able to understand the performance (or lack thereof) of your improvements is more important than actually achieving good performance.

Dos and don’ts

- allocate a separate Graph object for each thread; it does not consume much memory since it only contains an implicit description of how to generate the actual graph. Traversal of the graph (by calling the post() function) can then be done independently by each thread. Note that a State from one Graph object can meaningfully be compared to a State object from another Graph object, if both Graph objects are created from the same Promela input. So, don’t share a Graph object between threads.

- don’t try to pre-compute the full state-graph, since the result of that may not fit in memory.

- don’t use Thread.stop() to terminate a thread, since it is unsafe. Have a look at ExecutorService, or ExecutorCompletionService, or Thread.interrupt().

- Test your implementation regularly, and compare the result with that of the provided sequential implementation! Incorrect programs will be penalised heavily.

Accessing DAS4 from home

If you want to work from home and run on DAS4, you may notice that DAS4 (and actually, most of the VU network) is not directly accessible from home. You will have to go through the VU stepstone machine, which you can reach using either ssh or putty, using your VUnet id. Say you have a directory "NDFS" on your local machine. You can then first copy it to ssh.data.vu.nl:

```
scp -r NDFS <your VUnet id>@ssh.data.vu.nl:
```

(or something similar with pscp if you use putty on Windows). Note the ':' at the end of the command! This probably asks for your VUnet id password.

Then, login on ssh.data.vu.nl, again using your VUnet id.

```
ssh <your VUnet id>@ssh.data.vu.nl
```

This may again require your VUnet id password. You can now copy the "NDFS" directory to DAS4 using scp:

```
scp -r NDFS <your DAS4 account>@fs0.das4.cs.vu.nl:
```

This probably asks for your DAS4 account password.

Then, login on DAS4 (from the ssh stepstone machine):

```
ssh <your DAS4 account>@fs0.das4.cs.vu.nl
```
Assessment

Most important in determining your grade are the quality of your design, code and evaluation. The design should describe the challenges in the assignment, and your strategies to solve them. The code should be easy to understand and documented where necessary. The evaluation should give a clear picture of the performance of your solutions and explain the results. Use a clear coding style for your solution and concise language in your documentation.

If you want, your design and first version will be evaluated three weeks after the start of the assignment; this feedback may help you improve your solution. We encourage you to make use of this evaluation. Depending on the quality of your work, it may also add up to a point to your grade.

Your solution and evaluation will need to be submitted at the end of the assignment. The design counts for 30% of your grade, the implementation for 40%, and the evaluation for 30%.

After submission, feedback and grades will be provided by email. It will also be possible to make an appointment afterwards, to discuss your solution in more detail.

Submission

Submissions should be sent to c.j.h.jacobs@vu.nl. The submission mail should contain your VU-net-ID and student-number, and those of your partner if applicable. It also should contain three attachments: your design document (in PDF format), your evaluation document (again in PDF format), and a tar.gz file containing your java source files (the distribution contains a bin/mk-submit.sh script to create such a file).

Shortlist for bachelor students

- Decide if you want to work alone or paired and register.
- Study the assignment and referenced paper.
- Design and implement Alg2.
- Optionally submit the design document and your code before the first deadline (see below).
- Design and implement a single improvement.
- Evaluate your implementation.
- Submit the design, final implementation and evaluation before the second deadline (see below).

Shortlist for master students

- Decide if you want to work alone or paired and register.
- Study the assignment and referenced paper.
• Design and implement Alg\(\tilde{2}\) and Alg\(\tilde{3}\).

• Optionally submit the design document and your code before the first deadline (see below).

• Design and implement four improvements.

• Evaluate your implementation.

• Submit the design, final implementation and evaluation before the second deadline (see below).

Deadlines

For this assignment we use strict deadlines. After each deadline you will receive feedback. The deadlines are as follows:

• Design document and initial implementation: Sunday November 19, 23:59. This deadline is optional, but if you want feedback at this stage, you need to adhere to it.


References

