ASSIGNMENT 2: Concurrent PageRank

CSC3021 Concurrent Programming 2018-2019

Submit by 5:59 pm on Monday 19th November 2018

In this assignment you will create a concurrent implementation of the PageRank code that you have written for the first assignment in order to utilise multiple processor cores and accelerate the computation.

Questions 1-4 in this assignment depend on solving earlier questions. The amount of programming required is not much when measured by number of lines; but it may take time to take everything in. Starting this assignment early is strongly advised!

Question 1: Analysis

Consider the implications of concurrent execution for each of the sparse matrix formats, i.e., synchronisation requirements, shared variables and race conditions. Assume that each thread computes a subset of the iterations of the outer loop in `iterate()`, i.e., each thread processes a subset of the edges.

Briefly answer the following questions (do not implement anything yet):

- For each of the CSR, CSC and COO formats, describe the instructions and the set of variables and array elements where data races may appear (i.e., list the conflicting pairs of instructions and the memory locations they operate on).
- For each of the formats, discuss how data races may be avoided, e.g., using critical sections, hardware atomics, etc. Explain how you would use these constructs, e.g., what operations would need to be executed atomically? What code goes into the critical sections?
- On the balance, what graph format would you use in a concurrent program and why?
- For each of the graph formats, measure the duration of the part that we aim to make parallel (the `iterate` method) and the sequential part (the remainder) of the PageRank program created in Assignment 1 using the orkut graph. Using Amdahl's Law, what speedup may be achieved on 4 threads? How much on 8 threads?

Question 2: First Implementation

You will distribute the computation performed in the `iterate` method over multiple threads using barrier synchronisation. This implies that each thread should execute a subset of the iterations of the outer loop of the `iterate` method. Before and after starting the iterate method, all threads synchronise using a barrier. Barriers are a multi-thread (more than 2) rendez-vous

construct. Consult the Java Platform Documentation on barriers for more information and a usage example.[1]

Implement a concurrent version of the power iteration step for the **CSC** sparse matrix format. Distribute the iterations of the outer loop in the iterate method **equally** over all threads (each thread processes a consecutive sequence of n/T vertices, where n is the total number of vertices and T the number of threads). You will need to take care with rounding, i.e., n may not be a multiple of T.

A sketch of how the concurrent program should be structured is given below. Validate the correctness of your code by cross-checking the residual error for each iteration with the sequential code you submitted for Assignment 1.

Record the execution time for your concurrent solution when processing the orkut_undir graph and when using a varying number of threads. Select 1, 2, 3, … T+2 where T is the number of physical processing cores (including hyperthreads) that your computer has. Report the number of processor cores and the hyperthreading arrangement on the computer that you used for these measurements. Were you able to achieve the speedup predicted by Amdahl's Law?

Main thread (id=0)

```
create threads for id > 0;
while( true ) {
    barrier.await();
    if( flag == true )
        break;
    matrix.iterate(id*(n/T),(id+1)*(n/T));
    barrier.await();
    normalise y; // as in skeleton code
    delta = residual error;
    copy y to x;
    if( delta < tol )
        flag = true;
}
join threads with id > 0;
```

Other threads (id>0)

```
while( true ) {
    barrier.await();
    if( flag == true )
        break;
    matrix.iterate(id*(n/T),(id+1)*(n/T));
    barrier.await();
}
```

Question 3: Measuring Workload Imbalance

In order to best utilise the processor cores, each thread should execute for the same amount of time. The time taken by each thread is determined by the

[1] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html

number of vertices and edges that they process. Measure the time taken by each thread during a power iteration step using System.nanoTime(). During a separate execution, record the number of edges and number of distinct source and destination vertices processed by each thread (a thread may process more or less distinct sources compared to destinations!). These numbers roughly correspond to the number of floating-point additions and multiplications performed, as well as the amount of data accessed.

Analyse to what extent the number of vertices and edges processed explain the time taken by each thread. Report your measurements using charts (e.g., plot the time taken for threads to execute against the number of edges or vertices in a scatter plot) and summarise your observations.

Question 4: Addressing Workload Imbalance

If you balance the time taken by each thread, the total execution time will decrease. To this end, you need to balance the work assigned to each thread. Your measurements in response to Question 3 should give you insight how to improve the workload balance by modifying how the iterations of the outer loop in the iterate method are distributed over the threads. You can assign a different number of iterations to each thread in such a way that each thread takes about the same amount of time to complete execution. For instance, if you observe that execution time grows with an increasing number of edges processed, you may want to redistribute the loop iterations in such a way that each thread executes the same number of edges. You may achieve this by calculating the ranges of iterations processed by each thread in such a way that each range of iterations covers the same number of edges as opposed to the same number of vertices.

Implement this idea in your code and evaluate how much load imbalance is improved, and how that affects overall execution time. Use the same setup as before.

You may want to pre-calculate the number of iterations processed by each thread before starting the first power iteration step by analysing the graph.

Question 5: Fine-Tuning Concurrency

While concurrency helps to increase speed, you have seen in Question 3 and 4 that it is important to fine-tune the program. The goal of this question is to further increase the performance impact of concurrent execution. We will list a number of suggestions on how you can achieve this, but really any valid change to the program is acceptable as long as the correctness of the program is maintained.

Marking Method

Marks will be awarded in part on execution speed (8 marks) and in part on the basis of the ideas that you have pursued (4 marks). Some ideas you try may not result in the performance you expected, so it is important that you report on the things you tried regardless of the result and that you submit the corresponding code to your repository.

We will assess the performance of your code using the server at http://hvan.eeecs.qub.ac.uk/domjudge using the contest '1819_prcon'. There will be two problems you can submit your code to: the problem 'test' will evaluate your code using a number of small graphs and with different numbers of threads. The problem 'competition' will evaluate your code using an undisclosed graph. It will record the execution time of your program. The server will also perform some correctness tests. All submissions that pass the correctness test will be listed on the scoreboard, showing the relative position of your code in the class. This will show you how well you are doing.

The domjudge server will call your program with 5 arguments: three arguments describing the COO, CSR and CSC files, the number of threads that your program should use and the name of the output file where your program will write the PageRank values. The argument list may thus look like:
C:\> java –Xmx8g PageRank orkut_undir.coo orkut_undir.csr orkut_undir.csc 8 output.txt

The server has 8 physical CPU cores. Your program will be called with the number of threads set to 8. However, your code should be correct for any reasonable number of threads.

If your solution runs at the same speed as a **reference concurrent** implementation (similar to what you would obtain in Question 4), then you obtain 40% for this component. If it executes more slowly, you will receive less than 40%. If the performance of your code meets or exceeds the performance of an **undisclosed well-optimised concurrent implementation**, then you will receive 100% of marks for this component. Other correct solutions will receive marks distributed in between these extremes. If your solution fails a correctness test, or when inspection of the code reveals concurrency errors (e.g., data races, synchronisation errors, deadlock potential), then your code will be marked independently and based on judgement.

You are allowed to submit new solutions as frequently as you want; however only the final submission will be taken into account for marking purposes.

Potential Performance Optimisations

Some performance optimisations that are not too hard to pursue are listed below. This list is not exhaustive, you are free to let your imagination run wild! You are free to request additional guidance on these ideas.

A. Besides the iterate loop, you can also introduce concurrent execution in the loops in the methods calculateOutDegree, sum and normdiff. You can estimate potential speedup of parallelising these loops using Amdahl's Law.

B. You have experimented with single and double precision floating point precision in assignment 1. Can you leverage this to speed up your code? Can you find a way to mix both precisions to even greater benefit?

C. You may have found in Assignment 1 that the CSR format provides higher performance than the CSC format. However, data races make it

hard to use the CSR format. Data races can be avoided, however, when the graph is partitioned during a pre-processing step in such a way that all edges that point to the same destination vertex appear in the same partition. Each thread now processes one partition and all edges pointing to any particular vertex are processed by the same thread, similar as in the CSC. You can implement this by creating one CSR object per partition. You assign a subset of the vertices to each CSR object using the same rule as in your response to Question 4. If a range of vertices, say 100-199, is assigned to CSR object 1, then it means that all edges that point to a vertex in the range 100-199 will be stored in this CSR object. Other edges will be distributed to other CSR objects, but together all CSR objects store each edge exactly once. Building the CSR objects consists of just two passes over the full CSR: one to count the number of edges per partition/object so that you know how much memory to allocate, and one to build the new object. Once you have the objects, you change the code such that each thread iterates over its own object. Partitioning the CSR has some implications on the average degree of vertices in a partition, which you can read about in the literature.[2]

D. It has been demonstrated[3] that when the graph is partitioned, e.g., as in the case above, then the processing time reduces because a higher fraction of the data is found in the on-chip CPU caches. As you increase the number of partitions (i.e., each thread processes a more than one partition), execution time reduces further, up to some point. Extend your solution to C to create multiple partitions per thread (e.g., 8 partitions per thread) and observe the speedup.

E. The techniques described in cases C and D may also be applied to the COO format. In this case you can improve the effectiveness of the CPU caches by rearranging the edges within a partition. The COO matrix format allows you to store the edges in any order without affecting the functionality of the code. By using this freedom, you can ensure that accesses to the same memory location follow each other more closely in time, which improves the effectiveness of CPU caches. While it is hard to determine a good order to store the edges in, traversing the edges in the order of a space filling curve[4] is generally a good heuristic.[5] All that is needed is to pre-process the COO edge list, sorting the edges in the order that a space-filling curve would traverse them. The COO format with Hilbert sorting is known to be more effective than the CSR format for the PageRank problem.

F. Load balance can be further improved by recognising that not only the number of edges in a partition, but also the number of distinct

---

[2] https://doi.org/10.1145/3079079.3079097 section 1-3 and 4.3

[3] http://ieeexplore.ieee.org/document/8025292/

[4] https://en.wikipedia.org/wiki/Hilbert_curve and https://en.wikipedia.org/wiki/Z-order_curve

[5] http://www.frankmcsherry.org/graph/scalability/cost/2015/02/04/COST2.html

destinations affects the execution time. Load balance can be improved by relabelling the vertices in the graph (giving them new sequence numbers) in such a way that each partition of the graph has a similar fraction of the high-degree vertices and a similar fraction of the low-degree vertices. This can be achieved using a list scheduling algorithm (which is similar to bin packing). An implementation of this algorithm is described in the literature.[6]

G. Many other data structures exist to store sparse matrices. The supplementary document on PageRank cites a paper on "SPARSEKIT" which describes several more formats. You can find this paper online.[7] Moreover, compression of the graph is helpful because it is such a large amount of data that needs to be transferred between the main memory and the CPU for each iteration of the power method. The arrays holding PageRank values are really small in comparison. A simple idea is to use "short" integers where possible, but this may not always be possible. An algorithm for compressing the description of the graph is described in the literature.[8]

H. There is a fair amount of repeated computation. For instance, for a vertex with out-degree 10, the ratio of its current PageRank value over its out-degree is calculated 10 times per iteration of the power method. Each of these 10 calculations produces the same result. Speed can be improved by reducing the number of computations performed. This can be achieved by storing pre-calculated ratios in an array and retrieving them during the computation. This optimisation is not related to concurrency. However, it does affect load balance. Why?

Submission

Your submission will involve providing access to a **git repository** which contains regular commits of your work (you may use the same repository as for the first assignment), submission of at least one version of your code to the **domjudge** server and a written report submitted to **TurnItIn**.

Clearly label the code that you use to answer each of the questions, e.g., by creating directories named "Q1", "Q2", "Q3", "Q4" and "Q5" in the git repo or by including "Q1" etc in the filenames.

The write-up will be a PDF document containing the answers to Questions 1-6 and will be submitted through TurnItIn. It will be at most 4 sides of an A4 page. The minimum font size should be 11 point and margins must be at least 2cm in all directions. Only a standard 'Arial' or other 'Sans Serif' font should be used. Excess pages may be ignored during marking.

---

[6] https://arxiv.org/abs/1806.06576

[7] https://ntrs.nasa.gov/search.jsp?R=19910023551

[8] https://people.csail.mit.edu/jshun/ligra+.pdf

Guidance: Marking scheme:

Q1: 4 marks, Q2: 6 marks, Q3: 3 marks, Q4: 4 marks, Q5: 12 marks. Peer Review: 1 mark.
This assignment contributes to 30% of the marks for CSC3021.