ASSIGNMENT 3

CSC3021 Concurrent Programming 2023-2024

Submit by 5:59 pm on Friday 15th December 2023

Total 100 marks. Counts towards 40% of marks on CSC3021.

In this assignment you will create two concurrent implementations of connected components problem on undirected graphs. The goal of this assignment is to utilise multiple processor cores and accelerate the computation. You will also provide explanations of why these algorithms are correct using the theory of concurrent programming.

Question 1: Analysis (using PageRank) [10 marks]

The edgemap method is the main place in the code where time is spent. According to Amdahl's Law, this is where you should focus your attention to obtain a speedup. We can distribute the work performed by the edgemap method over multiple threads using "DOALL" parallelisation: as the method applies a "relax" method to all edges, each thread calls the "relax" method for a subset of the edges. To this end, the set of edges is partitioned with one partition assigned to each thread, which will call the "relax" method on all edges in the partition assigned to it.

Briefly answer the following questions using a sequential implementation of the program:

A. Measure the duration of the edgemap method (which will become the parallel part) and the remainder of the program (which will remain sequential) using the **CSC format**. Perform the measurements on the PageRank problem using the orkut graph. Using Amdahl's Law, calculate what speedup may be achieved on 4 threads? What speedup may be achieved on 1,000 (one thousand) threads?

B. Assume that the iterations of the outermost loop in the edgemap method will be distributed over multiple threads. Describe the race conditions that will occur when using the CSR, CSC and COO representations, if any. Hint: Processing an edge implies reading and writing values associated to the endpoints of the edge (source and destination vertices). Check the relax method in PageRank.java to identify what read and write operations are executed for source vertices and for destination vertices. Will these reads and writes result in race conditions for each of CSR, CSC, and COO?

C. Based on your analysis to answer the above questions, and what you have learned in the first assignment, what graph format would you use in a concurrent program and why?

Question 2: Parallel Edgemap (PageRank and Connected Components) [10 marks]

Implement a concurrent version of the power iteration step using the **CSC** sparse matrix format. Modify the program to:

1. Create a configurable number of threads. The number of threads that should be created is passed in as an argument to the program's main method.
2. Create a version of the edgemap method for the CSC format with signature public void ranged_edgemap(Relax relax, int from, int to), where the outer loop will iterate only over the range from "from" to "to" ("to" not inclusive).
3. Distribute the iterations of the outer loop of the edgemap method over the threads by determining the start and end vertex IDs for each thread. Keep in mind that the number of vertices may not be a multiple of the number of threads. For instance, if there are 17 vertices and 3 threads, then a first thread should iterate over the vertices 0,…,5, the second thread should iterate over 6,…,11 and the final thread should iterate over 12,…,16.
4. Run all threads concurrently and make them call the newly defined edgemap method on their designated vertices.
5. Wait for all threads to complete by joining them.

To help you with these changes, an abstract class ParallelContext has been defined in the provided source code[1] with the purposes of capturing the creation of threads and edge subsets in a way that is independent of the algorithm (PageRank/ConnectedComponents) and graph data structure. The ParallelContext is set by the Driver at program startup and used by the PageRank and ConnectedComponents classes to iterate the edges of the graph. A ParallelContextSingleThread is provided that merely calls edgemap on the appropriate graph, as per assignment 1. A place holder is provided for ParallelContextSimple where you can extend the edgemap method to take the additional steps outlined above. All changes can be implemented within the ParallelContextSimple class, except for the selection of the context in the driver. However, you should not feel obliged to use this class and may implement concurrency anyway you see most appropriate, in any class.

Validate the correctness of your code by cross-checking the residual error for each iteration with the sequential code, e.g., using the CSR implementation provided in the feedback of assignment 1. A validation program will also be provided.

Domjudge: Problem "Q2", using data structure "ICHOOSE" (you may map this to any of CSR, CSC or COO, as you see fit) and algorithms "PR" and "CC".

Writeup: Report the execution time for your concurrent solution on the **PageRank** problem when processing the orkut_undir graph and when using a

[1] git@hpdc-gitlab.eeecs.qub.ac.uk:hvandierendonck/csc3021_assignments_2324.git

varying number of threads. Select 1, 2, 3, … T+2 where T is the number of physical processing cores (including hyperthreads) of the computer that you are using for these measurements. Plot these measurements in a chart. Report the number of processor cores and hyperthreading arrangement on the computer that you are using. Discuss the achieved speedup you have obtained for PageRank and compare to the speedup predicted by Amdahl's Law.

Question 3: Disjoint Set Algorithm [15 marks]

Implement Jayanti and Tarjan's randomised concurrent disjoint set algorithm. You can build on top of the Relax interface and the associated methods you previously developed in the second assignment for visiting all edges of the graph using multiple threads. Your Relax class will have as data member a reference to the relevant data structure to store the parent of each vertex. Initially, a new set is created for each vertex (method makeSet). The relax method will record that the source and destination of this edge belong to the same set by calling the method union.

The code in the repository contains a partially defined class "DisjointSetCC" where you can add your code.

Domjudge: Problem "Q3", using data structure "ICHOOSE" and algorithm "DS".

Writeup: Paste the code of your DSSCRelax class in the writeup document using a pretty printer. Explain any design decisions of interest, at least your choice between no path compression, path splitting and path halving, and the motivation for this choice. Report the speedup of your parallel code (parallel iteration of edgemap has been achieved in Question 2 and should require no additional code) as a function of the number of threads using a chart.

Question 4: Pipelined Operation (Disjoint Set) [10 marks]

Implement pipeline operation specifically for the DisjointSetCC algorithm using the producer-consumer template where the producer thread reads edges from the file and the consumer threads call edgemap on those edges. The graph data structure need never be stored in memory as the DisjointSetCC algorithm makes only one pass using edgemap. Create a file "SparseMatrixPipelined" where the producer/consumer threads are set up in the edgemap method and select it in the main driver method. To simplify your code, use the sequential ParallelContext and implement all threading code in the edgemap function. You may use code from the practical material to implement the producer/consumer mechanism.

Implement your code such that a block of edges is passed as one unit between producer and consumer (minimum of one edge, but typically better performance if multiple edges are passed in a block). Typically a fixed-size block should be chosen (e.g., 128 edges) but the final block may have a different size. Note also that the buffer size is a configurable parameter that impacts performance (see practical material "6 Monitor Performance").

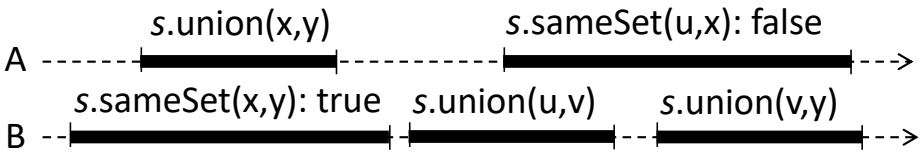
Domjudge: Problem "Q4", using data structure "ICHOOSE" and algorithm "DS".

Writeup: Explain which file format you read in (CSC, CSR or COO) and why. Record the speedup of your solution using a chart giving attention to ~~the number of threads used~~, the block size and the buffer size, and discuss your findings.

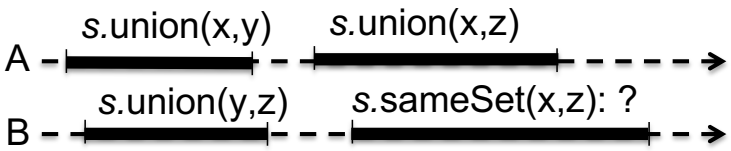
Question 5: Linearisability and JMM [15 marks]

The following diagrams show the execution timelines of threads and the methods executed against a concurrent data structure. "*s*" is a disjoint set data structure where each element is initially placed in a singleton set.

1. Is the following execution linearizable? Is it sequentially consistent? If yes, show a consistent execution order; if not, argue why such an order does not exist.



2. In the following execution, the response for B : s.sameSet(x,z) is omitted. Which outcomes out of "false" and "true" would be possible for a linearizable execution? Justify your answers.



3. Consider the Jayanti and Tarjan's concurrent randomised disjoint set data structure (slides 12_concurrentdisjointset.pptx, slide 6). Assuming a concurrent execution of "union(x,y)" with index(x) < index(y) and "find(x)", explain how the Java Memory Model ensures that the value of parent(y) is visible to the find method. Identify relevant load and store operations, synchronisation operations and happens-before relationships between them.

Question 6: Competition – Connected Components [40 marks]

The goal of this question is to further increase the performance impact of the connected components problem. You will need to select first what algorithm you further want to fine-tune (ConnectedComponents vs DisjointSetCC). Both are viable approaches. We will list a number of suggestions on how you can improve their performance, but really any valid change to the program is acceptable as long as the correctness of the program is maintained. 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! And don't forget about Amdahl's Law, nor the importance of keeping the disjoint set trees shallow.

Domjudge: Problem “Q6-test” and “competition”, using data structure “ICHOOSE” and algorithm “OPT” (you may map OPT onto CC or DS as you see fit).

Writeup: Explain which algorithm you have focused on and why. Describe any performance optimisations you have pursued and explain how well they worked out. 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. You will be marked separately on the effort and ambition of your attempts (15 marks available) and on the obtained performance (25 marks available). The performance of the code you developed for question 5 will be determined by submitting it to the DOMjudge server.

Competition marking: An execution time meeting the time achieved by a minimal reference solution will account for 40% of marks on Question 5. A time comparable to or below that of a well-optimised reference solution will achieve 100% of the marks on Question 5. It is expected that multiple students will exceed the latter performance.

The student achieving the fastest execution time recorded on DOMjudge with a correct solution will win the Concurrent Programming Competition, and receive a £100 Amazon voucher at graduation time.

Submission

Your submission consists of three components:

1. A writeup containing the requested discussion for each of the questions submitted through Canvas. Please be conscious about the length of your responses; adding incorrect or irrelevant content to an otherwise correct response will not result in full marks.
   Start the document with a **statement of how you have used Generative AI**. You are allowed to use such, **except for trying to elicit direct answers or generate code**.
2. Your Java code submitted through a private git repository on gitlab2.eeecs.qub.ac.uk. You will give at least developer access to this repository to h.vandierendonck@qub.ac.uk and yun.wu@qub.ac.uk (not to any of the demonstrators). Your repository should contain code for each of the questions in different classes, with selection of the relevant code determined in the driver.
3. Your Java code in response to the competition uploaded to DOMjudge at domjudge.hpdc.eeecs.qub.ac.uk under the competition ‘2324_A3’. **You should submit your solution to every question to the corresponding problem**. You submit a ZIP file containing all relevant Java source code files (.java extension) where Driver.java is at the root of the directory tree contained in the ZIP file. Upload any piece of code you have for each of the problems, even if not fully correct.

Potential Performance Optimisations Question 5:

You are welcome to request additional guidance on these ideas.

A. **[P/C] Workload balance:** Because of the frequent synchronisation on a barrier, the parallel code is sensitive to workload balance, i.e., each thread performing the same amount of work. Previously it was proposed that each thread should process the same number of vertices. To determine better load balancing strategies, measure how long each thread performs useful work, and how many vertices and edges are processed during this period. Draw scatter plots to determine whether the threads should process the same number of vertices or edges. Then adjust your code accordingly.
Some observations have been made in the literature, e.g., [1] section 3.3; [2]; and [3] section 5.

B. **[P] Pipelined Input of the Graph:** Parallelise the input of the graph data (the readFile method) using pipeline parallelism. The rationale is that reading the input data (BufferedReader.readline) and parsing the data (String.split, Integer.parseInt) take a fair amount of time. However, the loop they execute in is not fully parallel, it only admits pipeline parallelism. Split the loop that reads and parses the input across 2 or 3 threads using pipeline parallelism. You can find implementations of parallel queues in the practical material (part 6). Experiment with a few different buffer sizes for the queues. Measure the performance of your code and discuss your result.

C. **[P/C] Parallel input:** Use multiple threads to read in the graph. Files are in principle read sequentially, however, you can create multiple points of access to the file.[2] You can use a similar approach as in the iterate method where you distribute the reading of the lines of the file and its parsing to multiple threads. Two difficulties arise:

    a. A file is a sequence of bytes, and so you will determine the start and end positions for each thread at byte granularity, while you require those positions to coincide with the start/end of a line. As such, you will need to select a possible byte offset to start reading within a thread, then scan forward towards the end of the line

    b. You are lacking information required to construct the sparse matrix data structure, so you have to think before you do anything. Do you know at what position of the arrays a second or third thread should place the values it reads? Or do you create a separate array for each thread? But then, how large does each array need to be? Is it easier to read some data structure files concurrently than others and how will this impact the total execution time?

D. **[P] Worklist-based parallel iteration:** As vertices converge more and more over subsequent iterations, at some point we will end up with only a few vertices being updated in subsequent rounds (see [4], figure 4). Rather than visiting all edges, we only need to visit the out-going edges of the vertices that changed in the previous round. We can define a

[2] For instance, using the classes InputStreamReader and BufferedReader.

custom edgemap method that visits only the out-going edges of the vertices that were modified in the previous round. Additionally, we need to determine what vertices are modified during each call to edgemap, as this will determine the vertices that needs to be processed in the next call.

E. **[P] Thrifty label propagation:** The Thrifty Label Propagation algorithm proposes a number of techniques to accelerate convergence and to avoid processing all edges. The algorithm is discussed in a paper and requires worklist-based parallel iteration (point C), see [5].

F. **[C] Processing during input phase:** The disjoint set algorithm only requires a single pass over the graph. The baseline code provided first reads in the graph in memory, then reads the graph back from memory to process it. The data structure is large and does not fit in on-chip caches. Reading it back during edgemap will cause long-latency main memory accesses, which is not ideal for performance. Merging the file reading step and the disjoint set algorithm in one step (without storing the graph in any data structure) will be more efficient.

G. **[C] Path compression:** as per the slides, this affects execution time significantly.

H. [**C] Link order:** Link order can be affected by choosing the random order of vertices through index[.]. It is not strictly necessary to choose a random order, any order of vertices will do (why?). As such, you can consider different attributes of vertices, such as their degree, to determine rank order.

References

[1] Jiawen Sun, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2017. GraphGrind: addressing load imbalance of graph partitioning. In Proceedings of the International Conference on Supercomputing (ICS '17). Association for Computing Machinery, New York, NY, USA, Article 16, 1–10. DOI: https://doi.org/10.1145/3079079.3079097

[2] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: a computation-centric distributed graph processing system. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation(OSDI'16). USENIX Association, USA, 301–316. https://www.usenix.org/system/files/conference/osdi16/osdi16-zhu.pdf

[3] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: distributed graph-parallel computation on natural graphs. In Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI'12). USENIX Association, USA, 17–30.

[4] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '13). Association for Computing Machinery, New York, NY, USA, 135–146. DOI: https://doi.org/10.1145/2442516.2442530

[5] Mohsen Koohi Esfahani, Peter Kilpatrick and Hans Vandierendonck, Thrifty Label Propagation: Fast Connected Components for Skewed-Degree Graphs. In Proceedings of the 2021 IEEE International Conference on Cluster Computing (CLUSTER), pp. 226-237, DOI: https://doi.org/10.1109/Cluster48925.2021.00042

Overview of DOMjudge Problems

Problem	Test	Algorithm	Input	Threads	Graph
Q2	1	PR	ICHOOSE	12	RMAT
	2	CC	ICHOOSE	12	RMAT
	3	PR	ICHOOSE	12	orkut
	4	CC	ICHOOSE	12	orkut
Q3	1	DS	ICHOOSE	1	RMAT
	2	DS	ICHOOSE	12	RMAT
	3	DS	ICHOOSE	1	orkut
	4	DS	ICHOOSE	12	orkut
	5	DS	ICHOOSE	7	RMAT
Q4[3]	1	DS	ICHOOSE	1	RMAT
	2	DS	ICHOOSE	12	RMAT
	3	DS	ICHOOSE	1	orkut
	4	DS	ICHOOSE	12	orkut
Q6-test	1	OPT	ICHOOSE	1	RMAT
	2	OPT	ICHOOSE	12	RMAT
	3	OPT	ICHOOSE	7	orkut
	4	OPT	ICHOOSE	12	orkut
Competition	1	OPT	ICHOOSE	12	orkut
	2	OPT	ICHOOSE	12	orkut
	3	OPT	ICHOOSE	12	orkut

[3] ~~For the pipeline case, the number of threads is intended to specify the number of consumers. It is assumed there is just one producer.~~ A single-producer, single-consumer solution suffices.