Multi-Paradigm Real-Life Assignment

Virginia Niculescu
Department of Computer Science
Faculty of Mathematics and Computer Science, "Babeş-Bolyai" University
Cluj-Napoca, Romania
0000000299810139

I. Introduction

Real life problems may attract students' interest in solving problems even if they may also incur mechanisms with high degree of complexity as those from parallel and distributed programming.

"Parallel and Distributed Programming" is a mandatory course in the curricula of the third-year undergraduate students from the Computer Science specialization of our faculty. The course classes contain lecture hours, but also laboratories where students must do assignments that allow them to put into practice the principles, concepts and mechanisms theoretically presented. The course uses Java and C++ languages as demonstrators of the general programming concepts and mechanisms. For the proposed project the required execution language is Java, based on previous years insights' into the students' preference.

The presented assignment is a complex project that covers as objectives many of the previously presented topics:

- client-server,
- thread-pools,
- producer-consumer pattern,
- conditional synchronization,
- balanced distribution,
- data race,
- fine-grain vs. coarse-grain synchronization,
- synchronized methods vs. locks,
- signalization of the end of an operation/computation in multi-threaded and client-server environments,
- the impact on the performance of right splitting of the resources (number of threads).

Details about the assignment could be found at the address: //https://www.cs.ubbcluj.ro/∼vniculescu/ppd/project

II. The problem

Problem text: *An international programming contest proposes several assignments to very many participants from different countries. For each assignment some points could be gained and finally the competitors that gain more points will be the winners. There are several ethical rules that should be respected; if a competitor doesn't respect one of these rules, then he/she is eliminated from the competition; in this case a negative number of points is registered for the corresponding assignment. At the end of the competition, the results for each assignment are sent (by each country client) to a central server that stores them in a specific directory. There are several prizes (no_prizes) and in order to determine the winners, a list ordered descending based on the total points is needed. The server is responsible for delivering this list as soon as possible. Performance analysis should be conducted based on the resources' allocation for different tasks, on synchronization granularity, and on the size of the blocks sent by the clients.*

A. Macro-level Design

Two main components should be considered:

1) Client-Server component
   - Each client is associated to a country and sends the files of the corresponding assignments results.
   - The server receives files and stores them into directories that correspond to each assignment, and concurrently, executes the second component.
2) Classification list construction component
   - This is done at the server side using multiple execution threads.

1) Client-Server component:

- The server uses a thread pool executor that uses t threads.
- The connection should be based on Java sockets.
- A client sends a file split in blocks of predefined size.

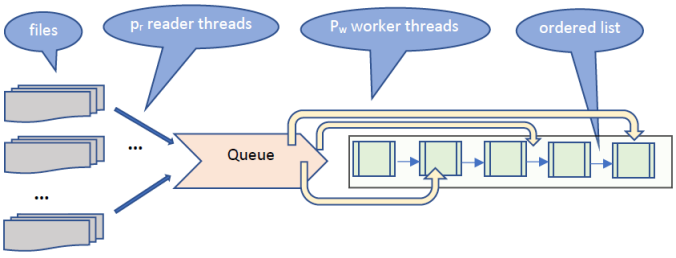

Fig. 1. Visual description of the ordered list construction.

2) Ordered list construction component:

- The *readers* are responsible for reading pairs from the files and add them into a queue Q that has a maximum capacity.
- The *producer-consumer* pattern should be used! Readers are the producers and the workers are the consumers.
- A *worker* takes a pair from Q and adds it into the list L (initially this list should be created as an empty list and finally it will contain the output ordered list).

The p threads that could be used for the list creation are split into two groups: *readers* (p_r) and *workers*(p_w) such that: $p = p_r + p_w$.
For adding a pair into the list the synchronization is based on:

- Variant W_A: mutual exclusion by blocking the entire list,
- Variant W_B: mutual exclusion at the node level.

III. Methodology and Analysis

The project is very complex and it should be developed in stages. Because of this, two approaches could be considered:

1) Inform the students from the beginning about the entire goal of the application, and then split its development into stages.
2) Start first with the problem that creates the ordered list from a given set of input files (that are stored in the shared memory), and then, add the distributed part based on the client-server application.

The first approach has the advantage of rising the students' interest by proposing real-life applications, but this may increase the failure concern due to the estimated complexity. The second is a more agile approach that could lead the students to the final results following a more gentle path, and in the same time, allowing better focus on different concurrency mechanisms. (This was the approach that we followed.)

On both approaches the students are asked first to developed the component that creates the ordered list from some input files that contain the pairs of type $(id, points)$. At this stage the *producer-consumer* pattern is discussed and analysed in more details, and on another side, the implications of the *fine-grain synchronization* over *coarse-grain synchronization*.
The general *producer-consumer* pattern and its implementation were previously presented at the lectures, so the students just have to identify the fact that this pattern fits to the problem *readers=producers, workers=consumers*), and put it to work. The workers should add new nodes into the ordered list, update the points of some nodes, or delete some nodes. These operations could imply *data race*, and so, the implementation imposes access based on mutual exclusion; this can be done for the entire list, which allows a very simple implementation. Still, for large volumes of data, the performance could be much better if this synchronization is defined at the node level. But the synchronization at the node level is not so simple to understand and implement and a detailed analysis of this solution should be provided to the students from the beginning (concrete examples, scenarios based analysis, recommendations as using head and tail empty nodes, etc.). The problem represents an illustration of the need to use such a *fine grain synchronization*, even if implies a much complex solution compared to the *coarse grain synchronization*.

The readers should read pairs from files. The files are organized into $no\_assignment$ directories, and there are p_r threads that should fulfill this task. It could be assumed that each directory contains a similar number of pairs distributed into $no\_countries$ files stored in each directory. A *balanced distribution* of the work should be done and this mainly depends on the relation between $no\_assignment$ and p_r.

Another interesting problem, which is not so obvious at the first analysis, and was proved to be difficult for the students, is given by the necessity to signalize the end of the computation. For the workers, an empty queue doesn't necessarily mean that the work is finished; they should be informed when no other data is going to be added to the queue. It is important for the students to understand that this is happening only when all the readers finish their job. *Poison Pill* pattern was discussed in this context. On the other hand, atomic variables could be used, and so, *low-level synchronization* was used, too.

The solution should be tested against the correctness but also performance. The correctness testing is suggested to be done by implementing a sequential solution, and then, the two outputs of the sequential and of the parallel solutions are compared. The students are informed that this method just increase the level of confidence relatively to correctness.
Different values for p, p_r, p_w are considered for the performance analysis, and the execution times are compared with the sequential time, too.

After this first component is well understood and analyzed, the students are asked to develop a client-server application in which several clients send files to the server that stored them in specific directories based on their names. For this operation the students are advised to define clients that send chunks (binary blocks) of files. The possible sizes of these blocks should be discussed, and performance analysis could be done for different values. The problem of *signalizing the end of the computation* appears also here, but in the context of the distributed memory environment, in which the solution should be based on messages. A comparison between distributed and shared memory cases could be done and the differences are emphasized.

Finally, the two components are put together and tested in several scenarios. The scenarios are defined based on different values for the total number of threads used by the server, and how are these distributed for the different tasks (t, p_r, r_w), based on size of the blocks on which the files are split by the clients in order to send them, and all these scenarios were tested for the both variants W_A and W_B. The testing activity is very important for this project since it depends on many parameters that may have impact on the performance.

Final analysis. Overall, the problem emphasizes that there are many factors on which the performance depends, and these factors are correlated. The analysis of the possible optimization should be done first at the component level and then for the entire application.

Variations: A significant variation is to eliminate the *reader* threads, and to impose the clients, instead of files to send result pairs (not only one, but a bunch of pairs) which are put into the queue (Q) directly by the server's executor tasks. The advantage of this approach is that the threads at the server side could be used more efficiently. If both variants are implemented than they could be compared from the performance point of view.
Another variation is to implement the list using only CAS operations.