Assignment 3 APC 2024

December 2024

1 Introduction

The Stable Marriage Problem (SMP) is a classic problem in algorithmic theory that seeks to find a stable pairing between two equally-sized sets of elements, where each element has preferences over the other set. A pairing is considered stable if there are no two elements that would prefer each other over their current partners[1].

In this analogy, we frame the problem in the context of computational efficiency. The *proposers* are applications, which need to be executed on computing devices. Each application has preferences over devices based on its execution time: the shorter the time, the more favorable the device. The *acceptors* are devices, which rank applications based on their ability to make full use of the device's computational power, measured in terms of FLOPS (Floating Point Operations Per Second). FLOPS is a standard metric for assessing computational performance, representing the number of arithmetic operations a system can perform in one second.

The goal is to find a stable pairing between applications and devices, ensuring there are no instances where an application and a device mutually prefer each other over their assigned partners. In practical terms, this stability ensures that applications are executed efficiently without underusing device capabilities, while devices are optimally loaded.

Through the Gale-Shapley algorithm, a well-known solution to the SMP, we can achieve a stable configuration that satisfies the preferences of both applications and devices. The following are the high-level steps of the algorithm:

- **Initialization:** Each member of the two sets is unpaired. Decide which set proposes (proposer) to the other set (acceptor). In our example, the acceptors are devices, while the proposers are the applications.
- **First round - proposal:** Each application "proposes" to its most preferred device.
- **First round - acceptance:** Each device will match with the most preferred application among the ones that proposed. It is possible that some

[1]Link to a video that explain the general mathematical problem: `https://www.youtube.com/watch?v=Qcv1IqHWAzg`

devices did not receive any proposal and that some applications have been rejected.

- **First round - termination:** If all the devices have been matched by an application, the algorithm ends. Otherwise, we continue with the next round.
- ***k*-th round - proposal:** Each unpaired application proposes to the next most preferred device.
- ***k*-th round - acceptance:** Each device chooses among the applications that proposed in this round (if any); if the device prefers one of the new proposers rather than the proposal that it already had, it dismisses the application already matched.
- ***k*-th round - termination:** If every application is matched, the algorithm ends, otherwise it goes to stage $k + 1$.

2 Overview

We developed a serial application to simulate the Stable Marriage Problem (SMP) using the `la::dense_matrix` data structure to manage preference data. The primary goal of the assignment is to parallelize the two methods that implement the proposal and the acceptance phase using MPI. To reach this goal you can rely on the following assumptions:

- The number of devices is equal to the number of applications, which will be referred to as the *input size*.
- The input size is a multiple of the number of MPI processes.

3 Working example

The application takes as input two textual files containing matrices that describe the preferences of one set over the other. Their file paths are provided through command-line arguments. Each matrix is a square matrix, where each row represents the preferences of one element in the set. For clarity, we will focus on describing the preferences of the acceptors, though the structure is identical for the proposers. In this representation, all proposers are enumerated using their indices. For example, if there are four applications (a_0, a_1, a_2, a_3), they are indexed from 0 to 3, with index 0 corresponding to a_0 and index 3 corresponding to a_3. Each row in the matrix contains the indices of the proposers, representing the preferences of the corresponding acceptor. The order of the values in each row reflects the acceptor's preference ranking. Specifically, the leftmost value corresponds to the index of the proposer most preferred by that acceptor.

In the initial implementation, we provide two input matrices that specify the preferences for four applications and four devices. Table 1 shows the device preferences over the applications, while Table 2 shows the application preferences. In this example, we use these preferences to show how the algorithm works.

Table 1: Device preferences over the applications. The input files store only the 4×4 matrix with the indexes.

Device	**First**	**Second**	**Third**	**Fourth**
d_0	0 (a_0)	1 (a_1)	2 (a_2)	3 (a_3)
d_1	1 (a_1)	0 (a_0)	3 (a_3)	2 (a_2)
d_2	0 (a_0)	3 (a_3)	2 (a_2)	1 (a_1)
d_3	1 (a_1)	0 (a_0)	3 (a_3)	2 (a_2)

Table 2: Application preferences over the devices. The input files store only the 4×4 matrix with the indexes.

Application	**First**	**Second**	**Third**	**Fourth**
a_0	0 (d_0)	1 (d_1)	2 (d_2)	3 (a_3)
a_1	1 (d_1)	0 (d_0)	3 (d_3)	2 (d_2)
a_2	0 (d_0)	2 (d_2)	1 (d_1)	3 (d_3)
a_3	0 (d_0)	1 (d_1)	2 (d_2)	3 (d_3)

The output of the application is the log of the simulation that keeps track of which application "proposes" to which devices, and the current matching. We use three types of logs. For clarity purposes, in this section we use a different color for each type of log. The actual output logs of the application will have the same color.

The first log signals that an application has "proposed" to match with a device. For example, the following log line states that application a_2 proposed to device d_0. The first column states the rank of the process that logged the line.

```
P0 - Proposer 2 propose to 0
```

The second log type signals that a proposer prefers the current match with respect to the other devices that have not rejected it, yet. For example, the following log line states that application a_2 prefers to be paired with device d_3.

```
P0 - Proposer 2 is fine with 3
```

Finally, the third log type states a match between a device and an application. For example, the following log line states that device d_2 has matched application a_3. To improve clarity, it will also report the application previously matched.

```
P0 - Acceptor 2 has accepted 3 (before was 3)
```

We use the number of applications as special value to state that a device has no matching applications. For example, the following log line states that device d_2 was not matched by an application before the current match.

```
P0 - Acceptor 2 has accepted 3 (before was 4)
```

3.1 Initialization

In the starting round all applications and devices are not matched.

3.2 First round

3.2.1 First round - proposal

In the first round no applications are paired. Thus, every application proposes to its most preferred device, which is defined by the first column of Table 2. For this reason the log generated by this phase is the following:

```
P0 - Proposer 0 propose to 0
P0 - Proposer 1 propose to 1
P0 - Proposer 2 propose to 0
P0 - Proposer 3 propose to 0
```

In particular, all applications except a_1 propose to d_0, whereas a_1 proposes to device d_1.

3.2.2 First round - acceptance

In this phase each device selects, among the proposing applications, its most preferred one, according to Table 1. In this case, d_0 can choose among three alternatives: a_0, a_2, and a_3. If we consider the row associated with d_0 in Table 1 (i.e., the first one), we notice that it prefers a_0 over a_2 and a_3. For this reason it will match with a_0, rejecting all the others. The only other device that received a proposal is d_1, which has received a proposal from a_1, which it accepts since it is the only one. Then, this phase produces the following log:

```
P0 - Acceptor 0 has accepted 0 (before was 4)
P0 - Acceptor 1 has accepted 1 (before was 4)
P0 - Acceptor 2 has accepted 4 (before was 4)
P0 - Acceptor 3 has accepted 4 (before was 4)
```

3.2.3 First round - termination

Since we have unmatched devices and applications, we need to perform another round of the algorithm.

3.3 Second round

3.3.1 Second round - proposal

At this point of the evolution of the system, a_0 and a_1 are matched with their most preferred device. For this reason they do not make any proposal to the other devices, generating the following log lines:

```
P0 - Proposer 0 is fine with 0
P0 - Proposer 1 is fine with 1
```

On the other hand, a_2 and a_3 will use their preferences—stated in Table 2—to find their next preferred device, by moving one step to the right of their preference row. For this reason, a_2 proposes to d_2, while a_3 proposes to d_1, generating the following log lines:

```
P0 - Proposer 2 propose to 2
P0 - Proposer 3 propose to 1
```

3.3.2 Second round - acceptance

In this round only d_1 and d_2 received a proposal from any application. Device d_1 is currently paired with a_1, which it prefers over a_3 according to the preferences reported in Table 1. For this reason d_1 rejects the proposal of a_3. On the other hand, d_2 accepts the proposal of a_2 since it is currently unpaired. To summarize, the current matches at the end of the second round are the following:

```
P0 - Acceptor 0 has accepted 0 (before was 0)
P0 - Acceptor 1 has accepted 1 (before was 1)
P0 - Acceptor 2 has accepted 2 (before was 4)
P0 - Acceptor 3 has accepted 4 (before was 4)
```

3.3.3 Second round - termination

Since both a_3 and d_3 are currently unpaired, we need to proceed to the third round.

3.4 Third round

3.4.1 Third round - proposal

At this point of the simulation a_0, a_1, and a_2 are already matched with the devices they prefer the most, among the ones that did not reject them. Only a_3 makes a proposal to the next preferred device according to Table 2. Moving one step on the right of its row, it will make a proposal to d_2. Thus, this phase is summarized by the following log lines:

```
P0 - Proposer 0 is fine with 0
P0 - Proposer 1 is fine with 1
P0 - Proposer 2 is fine with 2


P0 - Proposer 3 propose to 2
```

3.4.2 Third round - acceptance

At this point d_2 has to choose between its current match a_2 and a_3. According to Table 1, d_2 accepts the proposal of a_3, rejecting a_2. All the other matches remain the same. The following logs summarize the current situation:

```
P0 - Acceptor 0 has accepted 0 (before was 0)
P0 - Acceptor 1 has accepted 1 (before was 1)
P0 - Acceptor 2 has accepted 3 (before was 2)
P0 - Acceptor 3 has accepted 4 (before was 4)
```

3.4.3 Third round - termination

Since both a_2 and d_3 are currently unpaired, we need to proceed to the fourth round.

3.5 Fourth round

3.5.1 Fourth round - proposal

Also in this case, we have only one unmatched application (a_2), which, according to Table 2, will now propose to d_1. Even if this is its third option, it is the best among those that did not reject a_2. This situation is summarized by the following log lines:

```
P0 - Proposer 0 is fine with 0
P0 - Proposer 1 is fine with 1


P0 - Proposer 2 propose to 1


P0 - Proposer 3 is fine with 2
```

3.5.2 Fourth round - acceptance

While in the third round (see Section 3.4.2) the application that made the proposal was able to replace the current match, in this case d_1 prefers its current match (a_1) with respect to a_2, according to Table 1. This means that d_1 rejects the proposal and the matches are the same as in the previous round:

```
P0 - Acceptor 0 has accepted 0 (before was 0)
P0 - Acceptor 1 has accepted 1 (before was 1)
P0 - Acceptor 2 has accepted 3 (before was 3)
P0 - Acceptor 3 has accepted 4 (before was 4)
```

3.5.3 Fourth round - termination

Since we still have a_2 and d_3 that are unpaired, we need to proceed to the fifth round.

3.6 Fifth round

3.6.1 Fifth round - proposal

In this round a_2 proposes to the next most preferred device (d_3) that did not reject it, which is also its last option in Table 2. The other applications are still matched with their best option, so they do not make any new proposal. This situation is described by the following log lines:

```
P0 - Proposer 0 is fine with 0
P0 - Proposer 1 is fine with 1


P0 - Proposer 2 propose to 3


P0 - Proposer 3 is fine with 2
```

3.6.2 Fifth round - acceptance

Device d_3 is still unpaired to any application, thus it will accept the proposal of a_2, even it it is its less favorite one according to Table 1.

3.6.3 Fifth round - termination

Since all the applications and devices are being paired, we reached a stable matching solution and the algorithm ends.

4 Application description

The source code of the application is composed of the following files:

> `main.cpp` drives the computation. At first rank 0 reads the input preferences file to fill the application and devices preferences. Then it broadcast the input matrices to all the other processes.

`dense_matrix.hpp/.cpp` describes a dense matrix represented by a `std::vector`. It is the same as the one seen in class, with the difference that it stores unsigned integers instead of double precision floating points.

`logger.hpp/.cpp` utility function to print the proposal and matches.

`simulator.hpp/.cpp` the class the implements the algorithm that compute the stable matches.

Inside the `Simulator` class the method that computes the proposals is named `compute_proposal`. While the method that updates the matches, according to the proposals, is named `update_matches`. **The goal of this assignment is to split the computation done by these two methods among the available MPI processes. You are allowed to change only the serial implementation of these two methods, by making them parallel.**

These two methods are the only ones that generate the log described in Section 3, which use the MPI rank of the process to decorate the log message. We use this log to check the validity of the application and how you parallelized the computation. In particular, the simulation outcome should not depend on the number of processes and the computation must be splitted among the MPI processes. For example, if we use 4 MPI processes to compute the proposal described in Section 3.2.1, and each process compute a single proposal, it can generate the following log lines:

```
P0 - Proposer 0 propose to 0
P1 - Proposer 1 propose to 1
P2 - Proposer 2 propose to 0
P3 - Proposer 3 propose to 0
```

4.1 Compute the proposal matrix

The signature of the method is the following:

```
la::dense_matrix Simulator::compute_proposal()
```

The output of the method is a dense matrix (`proposal`) of size $N \times N$, where N is the number of both acceptors and proposers. Each row of matrix `proposal` represents an acceptor and each column represents a proposer. Each value stored in the matrix is either 0 or 1. If the element with index (3,2) has value 1 (i.e., `proposal(3,2)` is equal to 1), it means that the proposer with index 2 proposes to the acceptor with index 3; otherwise (i.e., `proposal(3,2)` is equal to 0), proposer 2 does not propose to acceptor 3.

The next best preference for each proposer is captured by a matrix (`proposer_status`) of size $N \times 1$, where each row represents a proposer. The value of each element of column 0 is the index of the most preferred acceptor that has not yet rejected the propose. For example, if the value of `proposer_status(2,0)` is equal to 1, then d_2 (the element at the second column) in Table 2) is the next acceptor to which proposer 2 will propose.

Each proposer operates in a independent way with respect to the others. It starts by check whether in the current round it has been paired with a device. In this case, it will not propose to any other device. Otherwise, it will propose to the most preferred device left, according to matrix `proposer_status`, by setting to 1 the corresponding value in matrix `proposal`. To keep track of the proposal, it will also increment by 1 the corresponding value in the next best preference matrix `proposer_status`.

To parallelize each execution of `compute_proposal()` we can split the computation of the preferences of each proposer among processes. Since each proposer operate in an independent way, we can assign an iteration of the outermost loop in the method to different processes. For example, if we have 4 proposers and 2 MPI processes, each MPI process can compute the proposals of 2 proposers. Then, we need to combine together the updates performed by each process on matrix `proposal` and on matrix `proposer_status`. This means that at the end of each method call you have to collect on the two matrices across all processes.

4.2 Compute the next matches

The signature of the method is the following:

```
void Simulator::update_matches(const la::dense_matrix& proposal) const
```

The input of the method is the proposal matrix, which is the output produced by function `compute_proposal()` described in Section 4.1. The goal of `update_matches` is to update the matrix which describes the current matches (`matches`). Matrix `matches` is of size $N \times 1$, where N is the number of acceptors. Each row represents an acceptor and stores the index of the matching proposer. For example, if the value of `matches(2,0)` is 1, it means that the acceptor with index 2 (d_2) is paired with the proposer with index 1 (a_1). We use the value N to represent the case where an acceptor is not paired with any proposer—for example, if $N = 4$ and the value of `matches(0,0)` is 4, then proposer 0 is not matched with any acceptor.

For each acceptor `i`, the value of `matches(i,0)` is computed in the following manner: the current match for `i` (if any), which is stored in matrix `matches` (i.e., the value of `matches(i,0)`) is compared with the most preferred proposer that made a proposal to `i` (if any). To perform the comparison, it uses matrix `prefences_acceptor`, which represents Table 1, to select the best proposer. If the device prefers the best proposer to the current match, then we update `matches(i,0)`.

In a similar manner as for function `compute_proposal()`, we can parallelize the computation performed by `update_matches()` by splitting among processes the computation of the values of the elements of `matches` (i.e., the outermost `for` loop performed by `update_matches()`)—that is, each process executes a piece of the outermost loop. Then, we need to combine together the parts of matrix `matches` computed by each process. This means that at the end of each method call you have to collect on the two matrices across all processes.

5 Compiling and executing the application

From within the Virtual Machine, you can generate the application executable by compiling the source files:

```
$ cd /path/to/assignment3_initial
$ mkdir -p build
$ cd build
$ mpicxx -o main ../src/*.cpp
```

In the example above, the path `/path/to/assignment3_initial` is a placeholder for the folder that you used to unzip the source files.

In the initial implementation you can find two different sets of input, with two and four devices/applications, with the reference output generated with a single process. Once you have compiled the application you can run it using `mpiexec`. For example, you can use the following commands to generate the output for four devices/applications:

```
$ cd /path/to/assignment3_initial
$ mpiexec -np 1 ./build/main ./input/apps4.txt ./input/devices4.txt
```

You can find the reference output in the related file: `output/example4.txt`. When you use more than one process, the order of the log lines and the rank of the process that produced the log line might change.

6 Delivery Instructions

The assignment is not mandatory. If the solution is implemented correctly, it can lead to a +1 point in the final grade.

Please, follow these instructions for the delivery:

- Download the zipped folder `assignment3_initial.zip` from WeBeep.
- Unzip the `assignment3_initial.zip` file.
- Change the name of the **unzipped** folder in "`YourCodicePersona`" (e.g., "`10699999`"). **It is important to do this before re-zipping the project with your solution.**
- Implement your code within the provided files.
- Test the code on the provided input file using a different number of processes.
- Zip the folder containing the entire project, making sure that the resulting file name is "`YourCodicePersona.zip`" (e.g., "`10699999.zip`").
- Upload on WeBeep $\longrightarrow$ Assignments $\longrightarrow$ Assignment3.

Attention: The assignment is personal; no group nor team work is allowed. In case of plagiarism, you will be assessed 2 penalty points (i.e., a -2 on the total score of your exam).

If you have questions, post them on the WeBeep Assignments forum. **Note that we will not provide feedback in the last 24 hours before the deadline.**

Code submission opens on 28th of December at 08:00 and closes on 4th of January at 18:00 (Rome time).

If something changes in the source code we provided, an announcement will notify you. So, keep an eye on WeBeep these days.