COMPUTATION STRUCTURES

Radix sort

Project 2 - Parallel programming

Maxime MEURISSE (20161278)
Valentin VERMEYLEN (20162864)

University of Liège
Academic year 2018-2019

1 Presentation

The goal of this project is to implement the "radix sort" sorting algorithm (for positive integer arrays) using parallel programming mechanisms.

To this end, several files have been created:

- `array.c`: a personal library containing functions useful for manipulating arrays, and more specifically the implementation of two-dimensional arrays in one-dimensional arrays;
- `communication.c`: a personal library containing a series of functions allowing easier manipulation of parallel programming mechanisms with System V;
- `main.c`: the main file of the project. It retrieves the data entered by the user and contains the implementation of the sorting algorithm as well as all the parallel programming mechanisms.

The project was compiled on the student machines of the Montefiore Institute thanks to the following command: `gcc main.c array.c communication.c --pedantic -Wall -Wextra -Wmissing-prototypes -lm -o main`.

The algorithm can then be called via the command: `./main (base) (size) (numbers)`. For example, the command `./main 10 4 54 23 6 12` will sort the array `[54, 23, 6, 12]` of size `4` whose elements are in base `10`.

2 Implementation

2.1 Parallel programming mechanisms

2.1.1 Shared memory

Several pieces of information have been placed in the shared memory:

- `numbers[]`: it is the main array containing the numbers to sort. This one is placed in the shared memory because all the worker must read and write in it;
- `temp[]`: this is the two-dimensional temporary array (but we decided to concatenate it in a one-dimensional array) to store the elements for sorting. It is placed in shared memory because all workers must write and read in it;
- `sorted`: this variable is used to tell each worker whether the array is sorted or not. It is placed in shared memory because all workers must be able to read it and the master must be able to modify it. Note that, if it's modified by the master, the workers do not have access to it thanks to semaphore waiting.

The shared memory is presented in the listing 1.

```
1 shared shm_numbers[N]
```

```
2 shared shm_temp[base][N] // initialized at -1 everywhere
3 shared shm_sorted = 0
```

Listing 1 – Shared memory.

2.1.2 Semaphores

In order to ensure mutual exclusion, several semaphores are used:

- `worker`: this semaphore is used to check that all the workers have finished their work (first the writing phase of the main array to the temporary array, then the writing phase of the temporary array to the main array) before the master continues his execution;
- `master`: this is a set of 2 semaphores. These semaphores are used so that workers wait for the master to complete its calculations before re-writing the contents of the temporary array to the main array. They also ensure that we do not try to read from something that is currently being updated.

These semaphores are presented in the listing 2.

```
1 semaphore sem_worker = 0
2 semaphore sem_master = {0, 0}
```

Listing 2 – Semaphores.

2.1.3 Message queues

We decided to use two message queues, presented in the listing 3, to exchange messages between the master process and the workers. Each process will fill it with its `digit + 1` as `mtype`, to ensure the message will reach the right recipient, and an array containing the digit used and the position to start writing in the main array, or the number of numbers on a given line in the temporary array depending on the sender (namely master and worker).

To summarize and make it clearer, here is the content of each queue:

- worker to master: `(id + 1, [digit, numbers])`;
- master to worker: `(id + 1, [pos])`.

Remark We used `id + 1` as `mtype` must be positive and we didn't want to use 0 for worker 0 by fear of unexpected results.

```
1 chan msgq_1 // worker to master
2 chan msgq_2 // master to worker
```

Listing 3 – Message queues.

2.1.4 Processes

The main process `main`, presented in the listing 4, is in charge of getting the arguments passed as parameters to the program, creating the IPC elements, creating the different processes that will sort the array and finally displaying the sorted array.

```
get the user parameters
create the shared memory, semaphores and message queues
compute max and iter
create the workers
create the master
clean memory and display result
```

Listing 4 – Main process.

The program, in addition to this main process, uses 2 different processes:

- `worker`: there are as many worker as the base. Each worker first deals with placing some numbers in the temporary array, and secondly re-writing those numbers in the final array. Each of these operations involves communication with the master process;
- `master`: there is only one master process. The first task is to receive the information of each worker about the number of numbers in each line of the temporary array. It then takes care of communicating to each worker the position at which he can begin to rewrite in the final array. Once that is done, it resets the temporary array to -1 everywhere.

These processes are presented in the listings 5 and 6.

```
worker(id, N, base) {
    search = true

    begin = (N / base) * id
    end = (N / base) * (id + 1) - 1

    divisor = 1

    if id == base - 1
        end = N - 1

    if N < base
        if id >= N
            search = false

        begin = id
        end = id

    while shm_sorted == 0
        if search
            pos = begin

            for i = begin to end
```

```
                num = shm_numbers[i]
                digit = (num / divisor) % base

                shm_temp[digit][pos] = num
                pos++

        divisor *= base;

        signal(sem_worker)
        wait(sem_master[0])

        get "nb", the number of numbers on the id_th line of "shm_temp"

        msgq_1!(id, nb)
        msgq_2?(id, write_pos)

        write back in "shm_numbers" at position "write_pos" the
            numbers on the id_th line of "shm_temp"

        signal(sem_worker);
        wait(sem_master[1]);

    signal(sem_worker);
}
```

Listing 5 – Process `worker`.

```
master(base, iter, N) {
    for i = 0 to iter
        for j = 0 to base
            wait(sem_worker, 0)

        for j = 0 to base
            signal(sem_master[0])

        for j = 0 to base
            msgq_1?(j + 1, msg)

        compute where worker "j" must write back in "shm_numbers" and
            put it in "msg[j]"

        for j = 0 to base
            msgq_2!(j, msg[j])

        for j = 0 to base
            wait(sem_worker)

        if i == iter - 1
            shm_sorted = 1

        reset "shm_temp" to -1 everywhere

        for j = 0 to base
            signal(sem_master[1])

```

```
28      for j = 0 to base
29          wait(sem_worker)
30  }
```

Listing 6 – Process `master`.

2.2 Sorting procedure

The sorting of the array is done in several stages with the elements presented previously.

1. First, after creating all the processes, each worker takes care of reading a part of the array containing the numbers to sort. Each worker reads a different part (if the base is greater than the size of the array, some workers should not do anything during this step). There is therefore no conflict between the workers during the reading. The bounds in which they read are `begin` and `end` in the listing 5.
2. Secondly, and immediately after reading, each worker places his read items in the temporary array according to the digit (therefore the iteration) concerned. Each worker writes the element to the line corresponding to its `id` and in the column corresponding to the position of the element read in the array. It is therefore impossible for many workers to write a number in the same place in the temporary array and we do not need semaphore to restrict access to the array. Each number is correctly sorted in that array. After this step, a semaphore is used to wait for each worker to finish writing before proceeding to the next step. The master then indicates that each worker can proceed to the next step with another semaphore (`sem_master[0]`).
3. Thirdly, each worker calculates the number of elements on the line of the temporary array he is dealing with (that is, the number of elements different from -1 on the `id`$^{\text{th}}$ line of the temporary array). Each worker sends this number of elements to the master via the first message queue with its `id + 1` as `mtype`.
4. Fourthly, depending on the number of elements on a worker's line, the master calculates from which position in the main array the worker must write back its elements. The master sends each worker his rewrite position via the second message queue. A semaphore is used to wait for each worker to finish writing his items before moving on.
5. Finally, depending on the current iteration, either the temporary array is reset and it moves to the next iteration, or it was the last iteration and the array is sorted. In this case, the workers are told via the sorted variable that the sort is complete. A semaphore (`sem_master[1]`) is used to avoid the workers reading `sorted` before it is updated if needed.

Remark The last set of semaphore waits is used so that each worker has finished with all memory elements before the master deletes the semaphores, message queues

and shared memory segments. That avoids a "semops: invalid arguments" as a worker waits for a semaphore that does not exist anymore.

2.3 Schematic summary of the communications

The communication between the different processes is carried out thanks to the semaphores and the message queue described above. This communication is shown schematically in the figure 1.

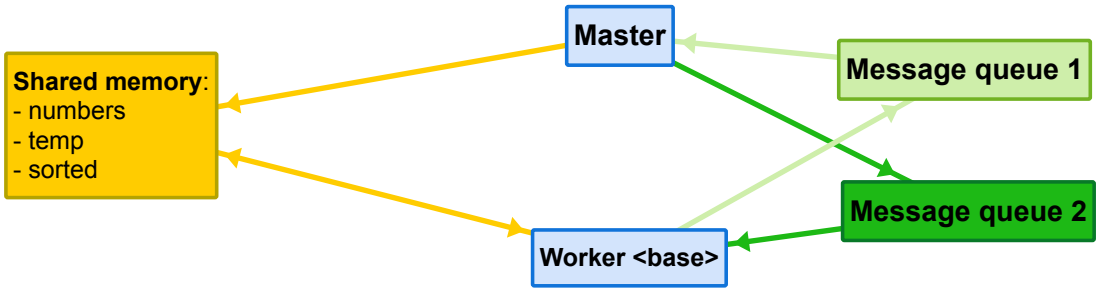

Figure 1 – Schematic representation of the communication between the different processes.

3 Conclusion

In conclusion, we can say that a parallel implementation is more difficult to implement than a sequential implementation, but drastically more effective.

Indeed, in order to manage the synchronization between the processes and the sharing of the memory elements (which could be shared between several distorted machines), more advanced techniques are required and a particular attention must be paid in order not to have conflicts.

However, these parallel programming techniques make it possible to perform a greater number of operations in a much shorter time, and thus to obtain results that could not be obtained via a sequential implementation.

Parallel programming is generally used for calculations much larger than those of this project, but it has already allowed us to observe all the effectiveness of parallelism on sorts of arrays containing a large amount of elements.