HPC Laboratory Assignment
Lab 3: Parallel Smith–Waterman with MPI

J.R Herrero and M.A. Senar

Spring 2023

Contents

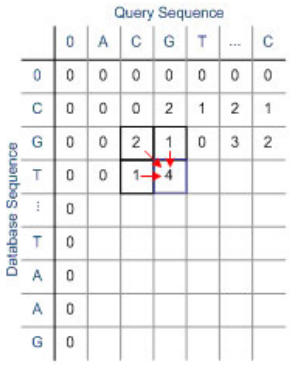
Chapter 1

Parallel Smith–Waterman with MPI

The goal of this assignment is to learn how to parallelize the creation of the scoring matrix in a Smith–Waterman algorithm for local sequence alignment. You will need to apply what you have learned about parallel programming with MPI; about having the data distributed while still collaborating by exchanging segment boundaries; the need for *ghost points*; and the need for *blocking* in one dimension. We are going to work with small problems. Therefore, we cannot expect to obtain any speedup. On the contrary, the overheads of the parallelization will slow down the execution. However, for much larger problems which you may deal with in the future, this can boost the performance.

```
for (i = 0; i <= dim1; i++)
  h[i][0] = 0;
for (j = 0; j <= dim2; j++)
  h[0][j] = 0;

for (i = 1; i <= dim1; i++)
  for (j = 1; j <= dim2; j++)
    {
      diag  = h[i - 1][j - 1] + sim[ s1[i] ][ s2[j] ];
      down  = h[i - 1][j    ] + DELTA;
      right = h[i    ][j - 1] + DELTA;
      max = MAX3 (diag, down, right);
      h[i][j] = ...
    }
```



In order to work on this assignment you need to extract the files from file `lab3.tar.gz` from ∼hpc0/sessions by uncompressing it into your home directory in `boada`. Unpack the files with the following command line: `"tar -zxvf /scratch/nas/1/hpc0/sessions/lab3.tar.gz"`[1] from your home directory in `boada`.

We provide a Makefile so compilation can be done via the `make` command. All the codes have to be compiled with `mpicc` and run in the `execution` queue by launching the appropriate `submit-xxx-mpi.sh` file. Remember that the interactive node in `boada` is heavily loaded. So please avoid performing interactive tests or limit them to very short sequence lengths with less that 4 `MPI` processes.

Invocation of the program without parameters or wrong parameters will provide help about its usage. As an example:

```
> SW sequences/query1.dat sequences/target1.dat data.score -1 3560

> mpirun -np 4 SW_mpi sequences/query1.dat sequences/target1.dat data.score -1 91 3560
```

[1] You can also work in your own computer if you install an `MPI` distribution such as `OpenMPI` or `MPICH` and change the scripts to call the appropriate version of `mpirun`.

At this point you should have enough understanding of all the concepts and enough experience in the lab to apply your knowledge to parallelize the sequential code that we provide. Even when we cannot expect speedups working on small problems, this assignment will show you have to tune your code so that you optimize the speedup obtained.

In order to simplify the assignment, you will have to work on a simpler problem, with no traceback, and with a block size and number of processes dividing the sequence lengths evenly. Later, if you feel like handling the complete problem, making it robust, complete and efficient, do it once you have managed to solve the simplified problem. Nevertheless, in this document we provide some guidelines.

1.1 Wavefront parallelism

Due to the dependencies we can only exploit *wavefront parallelism*. To avoid serialization we will need to apply *blocking* in the columns. Let's assume that each process is given `I` rows and a blocking factor `B`.

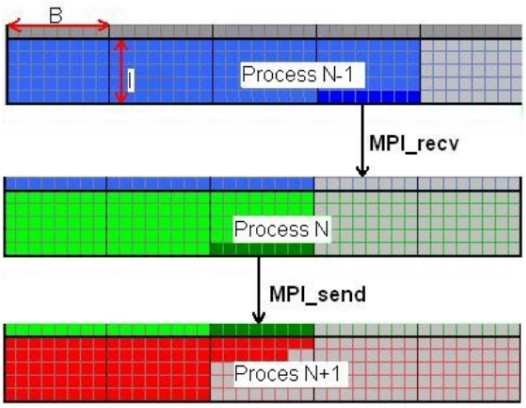


Figure 1.1: Wavefront Parallelism with ghost points.

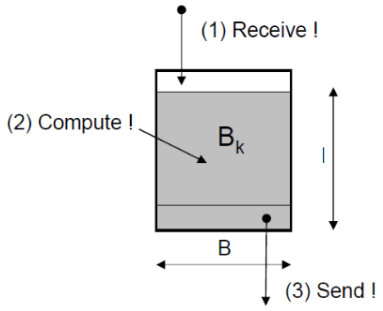


Figure 1.2: Block detail: Communications and one initial row for ghost points.

Note that in this case we only need one extra row for ghost points since all the data comes from previous rows and columns.

We provide three C source codes. You can start the parallelization with MPI from scratch using file `SW_mpi_scratch.c`, which is basically a replica of the sequential code. However, you might be interested in taking advantage of the parallel scheleton which we prepared for you. In such case, you only have to modify `SW_mpi.c`, filling in some gaps, while `SW.c` is provided for reference: to check that results are identical and to calculate the speedup of your parallelization. You can execute the programs with no parameters and they will print usage indications in the standard output. In the programs provided `size` is the length of the sequences we want to analyze. Additionally, in `SW_mpi.c` the parameter `BS`

is the blocking factor to apply blocking in the columns. We can use `h[i][j]` to access element `(i,j)` in the scoring matrix, and `h[i]` to refer to the beginning of row `i`; similarly with `xTraceback` and `yTraceback`. Note however, that in `SW_mpi.c` all the code which corresponds to the traceback has been disabled since we want to focus on the parallelization of the creation of the scoring matrix. In order to check if the parallelization worked correctly we will use the information about the maximum value found in the scoring matrix and it's position. To do so, Also, the last position computed for the scoring matrix (bottom-right corner) is provided. Thus, we will compare the values reported by the sequential code:

```
Max=161 xMax=36 yMax=37
Last element in matrix: h[36][38]=160
```

against those reported by the parallel code:

```
...
rank:3  Max=161 xMax=9 (global xMax=36) yMax=37
Last element in matrix: rank=3  (local index)   (global index)
                                h[9][38] =      h[36][38] = 160
...
Rank 3 has maximum value of 161
```

Note that the parallel code distributes the scoring matrix by rows. Thus, processes display both the *local index* and the *global index* of the row where the maximum was found. The global index (`xMax`) should match the one obtained with the sequential code. Each process works on whole rows. Therefore, in this case there is no local index in the columns and `yMax` is directly used.

The parallel code will also display information about the problem size given to each process, because debugging parallel codes is non-trivial.

1.2 Guidelines

Except for the 1^{st} item, here is a checklist which can help you work towards a functional parallel implementation if you start from scratch. However, if you just fill in the missing parts of the scheleton in file `SW_mpi.c`, these items have already been applied. Nevertheless, the checklist can help you to understand the goal of each part.

1. Perform an initial inspection of the parallel scheleton provided in file `SW_mpi.c`. Note that it has comments which aim at helping you to understand what needs to be done at each point. Note also, that there are differentiated parts for the *master* process and the *workers*. Remember though, that collective communication must be done by all processes.

2. Apply blocking in the columns to prepare the code for wavefront parallelism. Note that the block size `BS` is a parameter to the program provided from the command line. You need to split the loop used to traverse the columns into two loops (*strip mining*) and then apply *loop interchange*.

   ```
   #define min(a,b) (((a)<(b)) ? (a) : (b))
   ...
   for (int jj = 1; jj <= dim2; jj += BS){
       ...
       for (i = 1; i <= dim1; i++)  /* dim1 TO BE REPLACED by number of LOCAL rows */
         for (j = jj; j < min(jj+BS,dim2+1); j++) {
            ...
   ```

   Do this before adding any MPI primitive and make sure that the result continues to be the same. Since this is still sequential code test it for several values of BS using a single process.

3. Compute the number of rows `I` that each process needs to compute and which are those rows so that the data and associated work are well distributed. (See function `getRowCount` in the previous Lab2 and in the matrix multiply example in `MPI` in the course slides.)

4. Set a process to be the special *Master* process. Remember that process 0 always exist. Have the master be the only one which reads the input files and then broadcast or distribute the information to the *worker* processes.

5. You can start with a code where all processes store the whole matrix but work only on their part. However, ideally each process other than the master should just store a part of the scoring matrix (as in the previous Lab2 and in the matrix multiplication example parallelized with `MPI` in the slides). Note that if each process allocates space for the whole scoring matrix and traceback matrices you will soon run out of memory and will be quite limited in the problem size that you can solve. Beware that the execution will abort if the memory allocation fails. Check that **each process allocates enough space for the arrays it needs** before receiving data via an `MPI` primitive and before trying to use them.

6. Does the scoring matrix need to be initially broadcast or scattered from the master to the workers? The answer comes from observing if each process is going to update or overwrite their part of that matrix.

7. Compute the creation of the scoring matrix in parallel communicating the block boundaries between neighbor processes (review the course slides, the first problem within the In-Term Exam in May 2022 or alternatively in June 2018 Final Exam, and the codes you completed in Lab2).

8. An initial rather simple check of the correctness of the parallelization can be done by checking if the maximum score obtained is the same as in the original sequential code. (Remember that `MPI_Reduce` has a predefined `MPI_MAX` operator.)

9. Once you think your initial code is correct check if the results are correct using larger sequences and number of processes. However, please **do not execute this interactively**. Instead, launch your execution to the queue. For instance, `sbatch submit-mpi.sh 256 15000 12`. As stated above, don't expect to see speedups for this problem dimensions. The parallelization overheads are too large.

10. We need to know which was the process that found the maximum score, and in which position `(i,j)` that process found it. You can do this:

    (a) Either using `MPI_MAXLOC` in the reduction (see example in ∼hpc0/Examples/MPI/maxloc.c), plus adding a subsequent point-to-point communication between that process and the master to communicate that pair of values `(i,j)`.

    (b) Or defining a new reduction operation similar to that in ∼hpc0/Examples/MPI/ownreductionop.c

    You can also be interested in studying problem 2 within the In-Term Exam in May 2022.

Once we have checked that the program works correctly for several lengths of the input sequences (or other variations of the other parameters) and blocking factors, we need to evaluate the performance.

1. Once you understand and are sure that the *blocking* mechanism works, you have to explore which are good values for the *blocking factor (BS)* depending on the number of processors used. For that you can submit the `submit-blocksize-mpi.sh` script to explore different values for the *BS* argument. The number of processors is the only argument required by the script: try a few values and comment in your report if there is a substantial difference in the optimal blocking factor. **Please, do not execute this interactively**. Instead, launch your execution to the queue. For instance, for 10 `MPI` processes do `sbatch submit-blocksize-mpi.sh 10`.

2. Once you have selected an optimum value for the blocking factor, analyze the scalability by looking at the two speed–up plots generated when submitting the `submit-strong-mpi.sh` script. Edit that script and **change** the value for `blocking factor` using the optimum value you found in your exploration. Launch your execution with `sbatch submit-strong-mpi.sh`. **Please, do not execute this interactively**.

3. Do you have a Python code implementing Smith-Waterman? Are you curious about its speed. Compare the time of your parallel solution with that of the Python code (just for computing the scoring matrix) given the same inputs.

Optional: We have parallelized the creation of the scoring matrix. However, it'd be good to be able to print the subsequence found. Even if we just want to do that sequentially in the master, we will need several things:

1. Gather the final values for the scoring matrix and the traceback matrices in the master process. (See the examples ~hpc0/sessions/lab2/Jacobi*.c)

1.3 Hints

Data Structures

The code does NOT USE POSITION 0 of both s1 and s2. HOWEVER, when you allocate space for the vectors or transfer them you need to consider that the original code iterates from 1 to dim1 or dim2.

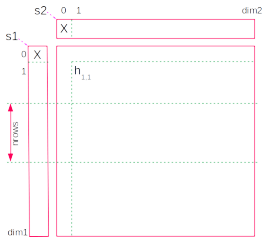


Figure 1.3: Main data structures used in the Smith-Waterman code.

Thus, there is one additional position in both vectors even when it's not used (position with index 0). [In the original code that position was used to store dim1 and dim2. But we changed the code to use dim1 and dim2 which are integers (4 bytes), while s1 and s2 hold shorts (2 bytes), to allow us to work on bigger workloads.]

Broadcast / Scatter

`h` is going to be overwritten so there is no point in sending it from the master.

Sequence 2 needs to be broadcast.

```
MPI_Bcast(s2, dim2+1, MPI_SHORT, 0, MPI_COMM_WORLD);
```

Beware that position 0 in s2 is not used. Maybe you need to transfer dim2+1 positions?

Sequence 1 needs to be scattered. The first position in s1 is not used either. Assuming all processes had to work on `nrows`:

```
MPI_Scatter(s1+1, nrows, MPI_SHORT, s1l+1, nrows, MPI_SHORT, 0, MPI_COMM_WORLD);
```

Obviously, a general code working on any number of rows and processes would require using `MPI_Scatterv`.

Methodology

Extremely Important:
- STUDY and THINK how to apply your knowledge to the target problem.
- TEST EACH CHANGE IN YOUR CODES SEPARATELY before moving on.
- Start with SIMPLE CHANGES and TEST EACH ONE separately.

Enjoy learning!

1.4 Deliverable

After working on this laboratory assignment you will have to deliver **two files**:

1. A report in `PDF` format (other formats will not be accepted) containing the answers to the questions stated at the end of this document;

2. A packed file in either `tar.gz` or `ZIP` format containing all the **source C codes** with the final code. PLEASE, **DO NOT** SEND OBJECT AND EXECUTABLE FILES.

Submissions which do not follow these indications can reach at most 80% of the marks corresponding to this assignment.

Your professor will open the assignment at the moodle aula ESCI website and set the appropriate delivery dates for the delivery. **Only one submission per group** must be done through the moodle aula ESCI website.

Important: In the front cover of the document, please clearly state the name of all components of the group, the username in `boada` of each the two members of the group (username `hpc1XYZ`), your ESCI e-mail addresses, title of the assignment, date, academic course/semester, ... and any other information you consider necessary.

As part of the document, you can include any code fragment, figure or plot you need to support your explanations. If you followed the approach of filling in the scheleton in `SW_mpi.c` then make the effort to explain well, in your own words, each part that you filled in.

In case you need to transfer files from boada to your local machine (laptop or desktop in laboratory room), or vice versa, you can use the secure copy `scp` command. In `scp` first we provide the source file and last the destination. For example, consider you want to copy to your local machine (your laptop, or desktop), a file named `foo.txt` kept inside directory `lab2` in your home directory of boada. And you want to copy it to your local current working directory which is referred to as "`.`". Then, you need to execute **in a shell running in your local machine**:
`"scp hpc1XYZ@boada.ac.upc.edu:lab2/foo.txt ."`