Parallel Computing and Algorithms

Project Design Document

Team Members:

- Franky (1110339128)
- Jirka Marsik (1110339127)

Our goal will be to implement both the column-based and the block-cyclic approach to parallel dynamic programming and to verify the results claimed in the "Parallel Design Pattern for Computational Biology and Scientific Computing Applications " paper on our cluster.

The implementation will be able to solve any of the dynamic programming problems demonstrated in the paper. Distributed computing shall be achieved using the MPI library. To facilitate rapid development, we have chosen Python as the programming language of our implementation. To communicate via MPI, we will use the MPI4PY bindings which closely follow the object-based design of the C++ API.

The entire SPMD program will be parameterized by the following values:

- dimensions of the dynamic programming matrix (*m*, the number of columns and *n*, the number of rows)
- *p*, the number of processors (externally specified by the MPI runner)
- *d*, division parameter (see below)
- *h*, block height (the number of rows each processor will compute before sending the results to the next processor)
- input loading function (a function which loads any necessary input data)
- cell computation function (a function which returns the value to be written to a given cell given the current state of the DP matrix)
- output writing function (a function which prints the result of the computation)

Algorithm design

Before we can do any computations, we first call the input loading function which returns some piece of data necessary to compute cells in the DP matrix. For example, in the Smith-Waterman algorithm, this piece of data would be the pair of sequences we wish to align. This data piece is then broadcast to all participating processors. Then, in every processor, we allocate space for the DP matrix, a 2D-array of dimensions *n* x *m*. The array will be implemented using the NumPy array objects which can be directly passed to the underlying C MPI functions.

Next, we divide all the columns of the DP matrix into *p*d* equally long sections. Processor *i* will be responsible for computing all the values in every *p*-th section starting with section *i*. This is to balance the load between the processors, since in a majority of DP tasks, the complexity of computing the value in cell (*i*,*j*) grows with *i* and/or *j*. Setting the division parameter *d* to 1 results in the column-based method, which is unsuitable to most DP tasks due to balancing issues. By increasing *d*, we can improve load balancing but we also start adding more overhead.

Now we have *p*d* sections each containing *w=m/(p*d)* columns. Next, we cut the DP matrix alongside the horizontal direction so that we end up with blocks of *h* rows. Combining both of these cuts, we end up with rectangles of height *h* and width *w* (or *w+1* when *m* is not divisible by *p*d*).

Finally, we can start the computation itself. Every processor has a set of rectangular blocks to compute. The blocks are processed left-to-right (the sections of columns are processed one by one) and top-to-bottom (within a section of columns, we first process the upper blocks). The processing of a block starts by issuing a blocking call to MPI_Recv where we wait to receive the computed values in the blocks having the same rows but lower-numbered columns. After we have received the data, we now have all the computed values of the quadrant to the left and above our current block in our memory. We then process our block line-by-line and for each cell, we call the cell computation function, supplying it with the coordinates of the cell we want to compute, the DP matrix and the piece of input data we loaded and broadcast at launch. After we have computed every cell in the block, we issue a non-blocking asynchronous Send (since the DP matrix is single-write, we do not have to worry about overwriting the values before they get sent) and continue processing the next block while the current finished block is being sent.

After the lower-right block of the matrix has been computed, the current processor now has the entire computed DP matrix in memory and can call the output writing function on it, which extracts the necessary data and prints the result of the computation.

Plans and goals

We plan to implement the algorithm as described above and measure its performance on one or two DP tasks for varying values of the *p, d* and *h* parameters given some input of reasonable size. Once all the projects have been submitted and graded, we will publish the Git repository online on Github, so other people can also reproduce the paper's claims without having to implement the algorithm again.

Tools and technologies used

- Python as the programming language
- MPICH2 as the underlying library for message passing
- MPI4PY as the Python bindings to MPI
- NumPy as the implementation of efficient (for sending over MPI) arrays in Python
- Git for source control