Universitat Politècnica de Catalunya

January 14, 2013

Smith-Waterman Alg. Parallelization using OpenMPI and OpenMP

Group 4

Alexandre Fonseca, Anh Thu Vu, Isak Nuhić

Contents

1 Introduction

This document aims to describe the actions we undertook and the choices we made to parallelize the Smith-Waterman algorithm, used in the determination of similar regions between 2 protein sequences.

In section 2 we start by describing the general algorithm and in section 3 we describe its serial implementation. In section 4 we move on to describing how we parallelized it using OpenMPI, OpenMP and both as well as 2 different variations of the parallelization: one without interleaving but column partitioning and another one with interleaving and column partitioning. Then, on section 5 we develop performance models for each of the aforementioned parallelizations and on section 6 we perform parameter estimation, compare the different implementations and compare them to the theoretical performance model, finding the theoretical and empirical optimum parameters. Finally, on section 7 we reflect on our work, comment on the major findings and discuss eventual complications.

2 The Smith-Waterman Algorithm

The Smith-Waterman algorithm is used to determine the similar regions between 2 different protein sequences. It takes in two aminoacid sequences, a and b, and determines the best alignment between the two based on a score computed as follows:

- One symbol from a sequence can be aligned with one symbol from the other sequence or with a blank '-'. Two blanks cannot be aligned.
- A score is given for each alignment of a pair of symbols based on an input score matrix, called PAM. The score can be positive, zero or negative.
- A penalty DELTA is given for an alignment between a symbol and a blank '-'. This DELTA value is provided by the user as the program starts.
- The score of an alignment between two sequences is the sum of the scores corresponding to all the symbols in the sequences. The best alignment is the one with the highest total score.

In order to obtain the alignment giving the best score, a local similarity matrix, h, is constructed. The columns of the matrix h correspond to the second sequence, b, and the rows correspond to the first one, a. The cells in the first row and column are initialized to zero. Then, the value of a cell at (i, j) is the maximum among the three following values:

$$diag = h[i-1][j-1] + score\ for\ the\ pair\ (a_i, b_j) \tag{1}$$

$$down = h[i-1][j] + DELTA \tag{2}$$

$$right = h[i][j-1] + DELTA \tag{3}$$

Note that according to the criteria for alignment, the three cases listed above are the only three possibilities to align two amino acid elements, i.e.: either they are

aligned to each other, or a blank is aligned with one of them. In this way, the value in each cell of the matrix h represents the score of the best alignment between the two sequences up to the elements corresponding to the row and column of the cell.

After computation of h, we determine the cell with the highest value and traceback the path from that maximum value to the initial row/column by retracing the decisions made at each point (if we chose to follow the diagonal cell, the down cell or the right cell) and thus construct the alignment corresponding to the highest score.

3 Serial Implementation

The serial implementation of the algorithm we used was based on code already provided to us but restructured to reflect a cleaner organization and better abstraction.

The main parameters are:

- **N** - The size of the protein arrays. In fact, this is not the actual size of the protein sequences (which is N - 1 since the first element of the array is not part of the sequence but rather contains the size of the sequence). However, this particularity is abstracted in the following sections for simplicity of explanation.
- **AA** - The number of amino acids being considered. This is important for the allocation of the PAM matrix.

The files containing the actual protein sequences are read into memory by converting each character found on their respective streams into a short representing the code for the given amino acid. The PAM matrix is read next by reading the triangular half-matrix provided in the input file and reflecting it to the other half of the PAM matrix (sim) due to the symmetry property. Finally, we obtain the delta value and allocate the h matrix (of size (N, N)).

The next step is to actually compute h following the procedure described in the previous section. This is done with the *computeResult* function which takes as input all the data read from the input files (a, b, sim and $delta$) along with the just created h matrix. After this procedure returns, we have a completely filled h matrix and need only perform the traceback.

The tracing back and printing of the result is performed with the *printResult* function using the complete h matrix. The maximum score is determined by scanning h and is saved in Max, along with its index *(xMax, yMax)*. During this scanning we also create the $xTraceback$ and $yTraceback$ matrices which capture the decisions made at each point (diag, down or right). Then, starting at $(xMax, yMax)$, we trace our way back to the initial row/column by following the decisions captured at $xTraceback$ and $yTraceback$ and print the found alignment.

The reason for the separation of the 2 previously described steps (computation and traceback) is due to us only being interested in parallelizing the actual calculation of h, the most computationally expensive operation.

4 Parallelization

As described on the previous section, we are interested in parallelizing the computation of the h matrix. In this and the following sections, we refer to *processes* as the entities that allow concurrent execution of some task and not with the usual OS meaning. This means that in an OpenMPI context processes might represent different nodes whereas in an OpenMP context they might represent threads.

On a naive attempt, one can try to divide the task of computing the matrix by its rows (or, equivalently, by its columns). However, as there are dependencies between a cell (i, j) with the three cells at its left, top and top-left corner, i.e.: cells at $(i, j-1)$, $(i-1, j)$ and $(i-1, j-1)$ respectively, a process P^k would have to wait for process P^{k-1} to complete his part and send the last row result to P^k. That would mean that in this naive implementation no parallelization whatsoever is achieved.

One solution is to divide the matrix into blocks. Each process will therefore be responsible for a row of blocks. As soon as process P^k finishes the computation of one block, it will send the last row of the block to the process P^{k+1}, who is responsible for the next row of blocks. In this way, we will achieve a pipeline-like parallelization where a process P^k can start its execution as soon as P^{k-1} finishes his first block and P^{k+1} can start when P^k finishes the first block while P^{k-1} is probably starting the third block, etc... In our project, we considered two variations of this approach which we detail below:

1. **Using different blocking sizes at column level**
   The matrix is divided into blocks of N/P rows and B columns, where N is the number of rows and P is the number of processes. Each process will be responsible for only one row of blocks during the whole execution.

2. **Using different levels of interleaving at the row level + blocking sizes at column level**
   The matrix is divided into blocks of $N/(P*I)$ rows and B columns, where I is the level of interleaving. Each row of blocks is then attributed to each process in a round-robin fashion so that each process still computes a total of N/P rows divided over I rows of blocks (abstracting leftovers). In this approach, a process P^k would handle blocks $(k, 0 \text{ to } B)$, $(k+P, 0 \text{ to } B)$, $(k+2P, 0 \text{ to } B)$...

Our main objective was to implement these 2 variations using the OpenMPI framework. As the reader might have noticed, the second variation corresponds to a general case of the first one when $I = 1$, therefore we developed these incrementally using OpenMPI. We then did some extra implementations using OpenMP and mixing OpenMPI and OpenMP. In the following subsections we describe each of the implementations in detail.

The optimum values of B and I are determined in section 6.

4.1 OpenMPI

With OpenMPI, each of the aforementioned processes (usually) resides at a different host/node with its own individual memory. Therefore, we have an increased

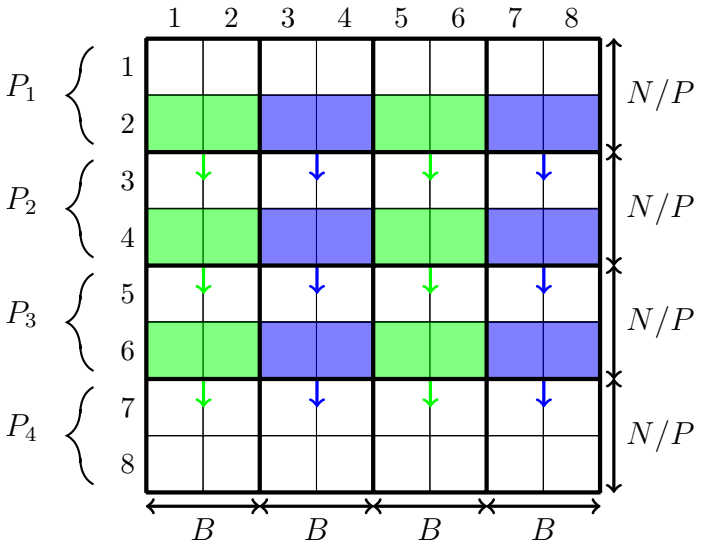

Figure 1: Dependencies between blocks, $N = 8$, $B = 2$ and $P = 4$

overhead of having to distribute the required data to all participating nodes.

4.1.1 Using different blocking sizes at column level

As the program starts, the root node has to get all the required inputs from the user: read the files containing the two amino acid sequences (constructing the a and b arrays), read the PAM file (constructing the sim matrix) and getting the value of the gap penalty ($DELTA$). b, sim and $DELTA$ are then broadcasted to all the nodes (since they are needed by all of them) whereas a is scattered through all the nodes (since node 1 only needs the first N/P rows, node 2 the second set of rows and so on) with the partitions residing in the $localA$ array. Leftover rows due to non-exact divisions are kept at the root node and calculated at the end. Each process also initializes its $localH$ matrix (of size $(|localA|, |b|)$) and sets its first row and first column to zeros just as what happened in the serial case.

Once all required data is available and supporting structures created, each node starts computing the blocks (B columns) on its set of rows sequentially. However, if the node is not the root ($rank > 0$), it must first wait for the last row of the block calculated by the previous node due to the dependencies highlighted on the previous section and shown in Figure 1. Initially, only the root node will be working. However, once it finishes computation of the first block and sends its last row to the next node, two nodes will now be working simultaneously: the root node on its second block and the second node on its first block. After computing the block, each node (with exception of the last one) will send the last row of that block on to the next node. This process is repeated until there are no blocks left. In case N/B doesn't have an exact integer result, the remaining columns are included in the computation of the last block.

Once all blocks are calculated, we perform a gather operation on all $localH$ arrays, filling the h array on the root node. The leftover rows due to non-integer

divisions are then calculated by the root node and when that's done we have our complete h array and need only compute the result, which as described before, is not object of parallelization.

4.1.2 Using different levels of interleaving at the row level + blocking sizes at column level

With interleaving, a node no longer handles a continuous set of rows but rather discrete groups of rows distributed over the entire M matrix. For instance, the first node may handle rows 1-4, 9-12, 16-20. The previous separation of rows in several blocks also continues to be applied but since we now have a non-contiguous row-space, we will obtain a greater number of blocks, i.e, if on the previous case we had 4 blocks of 8 rows, we may now have 16 blocks of 2 rows. In Figure 2, we show the case where each discrete set of rows contains exactly one row. Therefore, node 1 computes all of the blocks on row 1 sending that row to the next node (since it's a dependency of the computation of the next row). It then proceeds on to computing row 5.

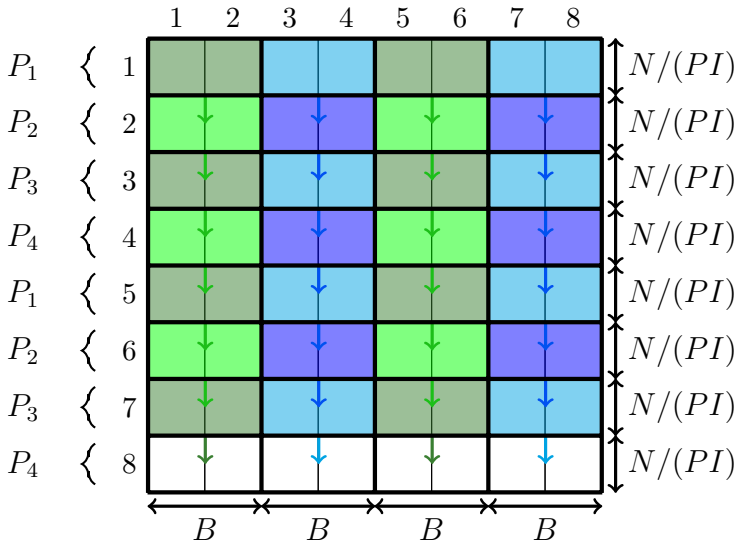

Figure 2: Dependencies between blocks, $N = 8$, $B = 2$, $P = 4$ and $I = 2$

On the previous example, since the number of horizontal blocks equals the number of processors, the maximum parallelization is easily achieved with no specific particularity. If the number of horizontal blocks is less than the number of processors, the maximum parallelization is given by the former. Otherwise, the maximum parallelization is given by the number of processors. In this last case, however, we have to be careful if using blocking sends. Since the dependencies are only at the row level (not at the column level since an entire row is handled by a specific node), we have to choose carefully how to organize the computation of the M matrix to maximize node cooperation. If, as before, we let each node compute an entire row of blocks before passing to the next one, and referring to Figure 3, once the last node finishes computation of its first block and wants to send it to the next node (actually, the first node due to the round-robin attribution), it will have to wait until the

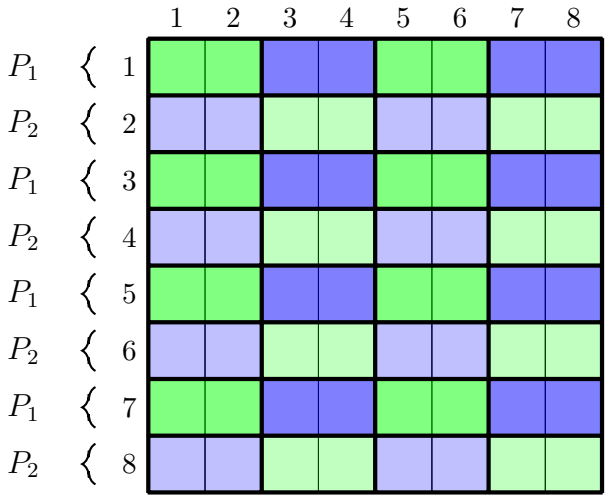

Figure 3: Multiple blocks per node per cycle, $N = 8$, $B = 2$, $P = 2$ and $I = 4$ Alternating colors represent consecutive cycles, with shades showing processing required at each node.

first node has finished computing its entire row and specifically ask for that data. This is clearly undesirable. To prevent this, we reorganize the computation of M as shown in Figure 4, creating partitions with at most P columns of blocks. Notice that there are no inter-node dependencies between each partition and, therefore, everything works correctly.

The process P_i repeats the following 3 things until it has finished all the blocks in one of its rows of blocks (given the partition discussed before). If there is another row of blocks to calculate in the current partition, it starts to calculate that row and so on until all its rows of blocks in the current partition are calculated. When that happens, the node moves on to the next partition (if any exists).

1. Process P_i receives the last row of a block from the previous process.
2. Process P_i performs the calculation of its block.
3. When process P_i has finished it sends the last row of the block it calculated to the next process. Now if the process has the last rank and if there are rows of blocks left to be calculated in the current partition, it will send the last row to the process with the first rank. If there is nothing left to calculate it will not send anything.

This implementation can actually do both discussed versions up until now - with or without interleaving (i.e, considering $I = 1$). As before, the root node is responsible for reading input data and distributing it to the other nodes. This is done in the same fashion as in the previous implementation with a slight twist: the elements of sequence A which have to be scattered to each node are no longer adjacent to one another. Unfortunately, *MPI_Scatterv* is unable to handle scattering of multiple discrete groups per node so we had to be creative. After reading sequence A, the root node calculates the rows for which each node is responsible for and

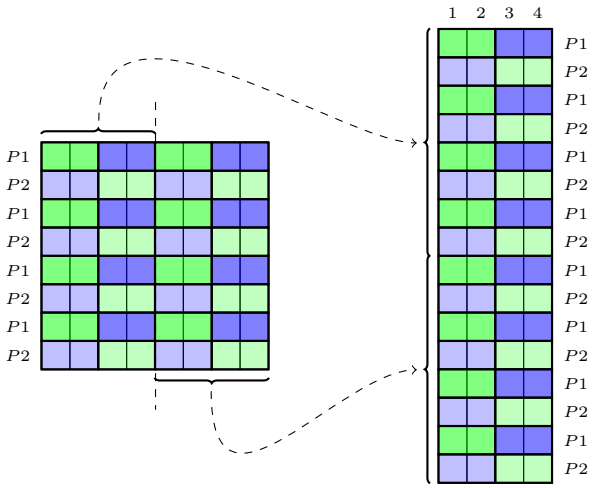

Figure 4: Rearrangement of M to facilitate the understanding of the computation with maximum parallelization.
Alternating colors represent consecutive cycles, with shades showing processing required at each node.

reorders the sequence so that those rows handled by the same node are adjacent to one another. This reordered sequence is then scattered normally with *MPI_Scatterv*. This way, we are able to distribute all the required data related to sequence A with a single scatter operation instead of having to do I operations.

Another particularity is that the $localH$ matrix at each node also contains non-sequential data and each node must track the positions it reads/writes to carefully. In fact, whereas, on the previous implementation, only the first row of $localH$ contained a value sent by the previous node, in this implementation there are multiple such rows including in the middle of the $localH$ matrix (one for each row of blocks). Therefore, before gathering the $localH$ matrices into a h matrix on the root node, those aforementioned rows have to be removed to prevent the introduction of duplicates on the final matrix. In addition, after the gather, h will have the same order as the reordered A sequence which is not the order we are looking for. To obtain the correct result, we have to apply the reverse ordering operation that we applied to A at the beginning. Once this is done, however, the program goes on to computing the final result as before.

4.2 OpenMP

In OpenMP, the processes are not nodes located in different machines with independent memories but rather threads sharing a common memory space. As such, the actual computation of h is very similar to the OpenMPI interleaving case only we now don't have to broadcast, scatter or gather any data since we can just access it from the common memory.

The actual implementation of the parallelization relies on the *task* paradigm. In

this paradigm, we have a pool of threads waiting to perform work. Each "piece" of work is known as a task and is assigned to one of the waiting threads automatically. In our implementation each of these tasks handles the computation of a different block of size (BY, BX) (BY is equal to N/PI on the OpenMPI case and was used in its stead due to simplicity of implementation).

Because of the dependencies between blocks already discussed, we needed a way to determine when we can actually start the next task (this should only happen when the blocks/tasks it depends on have already completed). To this end, we use a *block_status* matrix of size $(N/BY, N/BX)$. Each cell of the matrix can hold one of 3 values:

- **VOID** - The block represented by this cell is still a blank slate with no thread operating on it.
- **CREATED** - A thread to operate on this block was already created and so its computation is underway.
- **DONE** - The block represented by this cell is completely calculated and therefore, its last row and rightmost column are accessible to the adjacent blocks.

We start by creating the threads through the use of a *parallel* pragma. We then order one of those threads to start computing the initial block (using the *single* pragma and the *computeBlock* functions). The remaining threads will enter a waiting state by reaching the barrier at the end of the *single* pragma. After finishing the computation of a block, we mark the respective cell of the *block_status* matrix as DONE and decide whether or not to start new tasks: one for the block to the right and another for the block below. In the case we are checking whether to create the task regarding the block to the right, we check if the block above that one is marked as DONE. If that is the case, we perform an atomic check-and-set (using the *critical* pragma) of the status of the block to the right to the state CREATED. If it was already in such a state, then the task had already been created by a different task (the one responsible for the block above the one we were trying to start) and as such we don't do anything. Otherwise, we create the said task. The process works in the exact same way with the block below.

As tasks are created, the threads that were in a waiting state at the barrier, start computing them and eventually all tasks are completed and all threads have reached the barrier. When this happens, h is totally computed and we can proceed to the computation and printing of the result.

4.3 OpenMPI + OpenMP

Combining OpenMPI and OpenMP, we now have two types of processes: nodes which compute blocks concurrently and threads which compute partitions of each of those blocks concurrently (or rather - subblocks).

Overall this works in much the same way as the OpenMPI interleaving implementation. However, now the computation of each block spawns a set of threads

which calculate subblocks in the same way as the threads in the OpenMP calculated each block.

5 Performance model

5.1 OpenMPI

In this section, we attempt to model the execution time of our OpenMPI based parallelizations, taking into account not only the computation time but also the communication time. To that end, we consider a cluster with P nodes and sequence sizes of N so that the computation matrix has size N^2.

During the initial subsections, we consider a simple linear/ring array. On subsubsection 5.1.3 we analyze the particularities of considering other topologies.

For simplicity reasons, we assume that nodes are somewhat synchronized in their executions, therefore if 4 nodes are working concurrently, performing the same amount of work which takes t time, they'll be able to obtain 4 results in t time.

5.1.1 Using different blocking sizes at column level

When considering the division of the matrix in blocks of size $(N/P, B)$, the dependencies of the problem (shown in Figure 1), make it so that its behaviour when parallelized is somewhat similar to a pipeline execution in the sense that we can identify the following 3 distinct phases:

1. **Growing Phase** - This phase corresponds to the latency period of a normal pipeline, where the system isn't running at full capacity yet. The amount of blocks being processed in parallel increases by one for each cycle until we reach the maximum parallelism of $Y_{max} = min(N/B, P)$ blocks in which case we enter the second phase. This means that, the total number of cycles spent in this phase is given by:

$$c_{grow} = Y_{max} - 1 = min\left(\frac{N}{B}, P\right) - 1 \tag{4}$$

2. **Steady Phase** - During this phase, our system is able to execute Y_{max} blocks simultaneously. Examining the structure of the problem, it is obvious that this is the maximum amount of parallelism we can get in terms of block processing. If the blocks are squares (i.e, $N/P = B$), we will only spend one cycle in this phase where all P nodes will be working simultaneously (since $N/B = P$). However, if the blocks aren't squares we will spend several cycles in this phase. If $N/P > B$, all nodes will be working simultaneously (Figure 5) while if $N/P < B$ at most N/B nodes will be working simultaneously. (Figure 6).

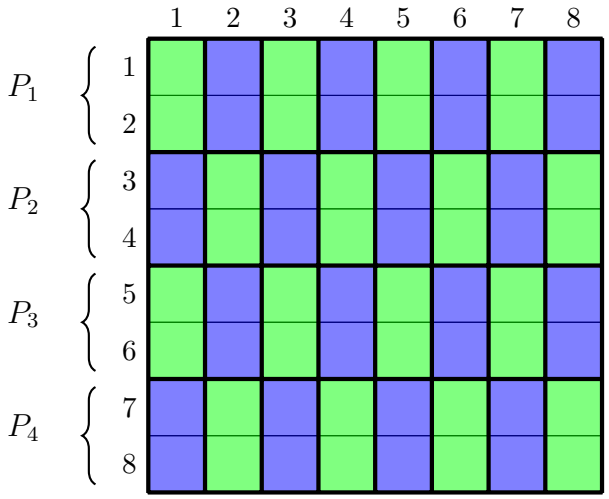

Figure 5: Example of computation with vertical rectangle blocks. Alternative colors represent alternative cycles.

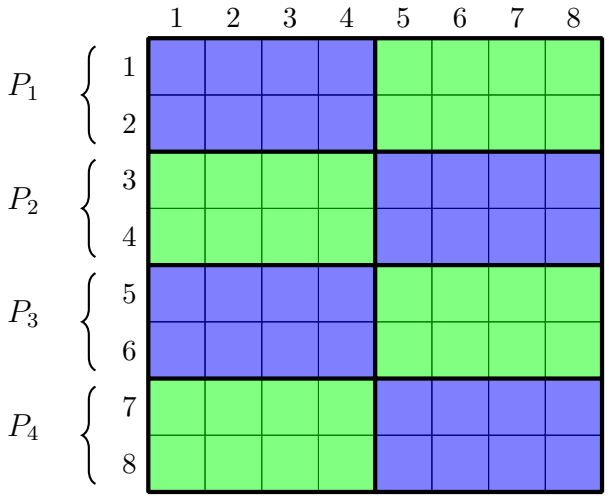

Figure 6: Example of computation with horizontal rectangle blocks. Alternative colors represent alternative cycles.

In the general case, the total number of cycles spent in this phase is given by:

$$\begin{aligned} c_{steady} &= \frac{b_{total} - b_{grow} - b_{shrink}}{Y_{max}} \\ &= \frac{PN/B - 2\sum_{i=1}^{Y_{max}-1} i}{Y_{max}} \\ &= \left(\frac{PN}{B} - 2 * \frac{Y_{max}-1}{2} * Y_{max}\right) * \frac{1}{Y_{max}} \\ &= \frac{PN}{BY_{max}} - Y_{max} + 1 \end{aligned} \tag{5}$$

3. **Shrinking Phase** - This phase is symmetrical to the growing phase and as such, the total number of cycles spent in this phase is also given by Equation 4.

The concept of cycle mentioned in the description of each phase represents (in the specific case of this modeling) the period of time required to compute an entire block. Given Equation 4 and Equation 5, the total number of cycles for the computation of M is given by:

$$c_{total} = \frac{PN}{BY_{max}} - Y_{max} + 1 + 2 * (Y_{max} - 1) \tag{6}$$

$$= \frac{PN}{BY_{max}} + Y_{max} - 1 \tag{7}$$

Figure 7 shows an example with the distribution of phases (by colour) and the various cycles in each phase (by opacity).

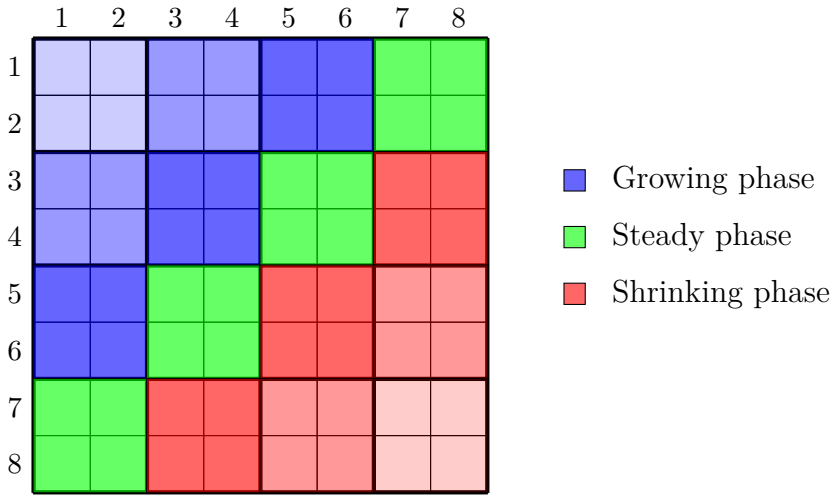

Figure 7: Different phases, $N = 8$, $B = 2$ and $P = 4$

Using just OpenMPI, we will process each block sequentially at each node. This means that in order to be able to compute a block, each node will have to perform a total of $N/P*B$ computations (one for each entry). Therefore, the total computation time per block (and, thus per cycle) is:

$$t_{comp\_block} = \left(\frac{N}{P} * B\right) * t_c \tag{8}$$

Where t_c represents the time needed to calculate a single $M[i][j]$.

Regarding communications, initially we have to scatter N elements of sequence A (the one represented vertically, representing rows of M) through P processes (N/P entries per process) from the root process (One-to-all scatter). According to [1, ch.4.4], we are able to obtain a scatter time for linear arrays/rings of:

$$t_s * log(P) + t_w * \left(\frac{N}{P}\right) * (P - 1) \tag{9}$$

We also need to broadcast the entire B sequence (since each process will make computations over the entire set of columns of M), delta, the PAM matrix and the

sizes of the sequences. For simplicity, we will abstract these last 3 broadcasts as the broadcasted data is either very small (sizes and delta) or is fixed and could just as well be stored at each node (and is also small if we consider large sequences). Therefore, we are left with the broadcast of B which according to [1, ch.4.1] for linear arrays/rings takes the following time:

$$(t_s + t_w * N) * log(P) \tag{10}$$

All things considered, the initial communication time can be approximated to:

$$\begin{aligned} t_{comm\_initial} = 2 * t_s * log(P) + t_w * \left(\frac{N}{P}\right) * (P-1) + \\ t_w * N * log(P) \end{aligned} \tag{11}$$

After computing a block (after a cycle), we need to send the bottom row of the block to the next process. This is a simple send of an array of size B and, as such, should take:

$$t_{comm\_block} = t_s + B * t_w \tag{12}$$

This sending of the block to the next node should occur after computing a block in every node but the last. However, as an approximation, we ignore this detail on the intermediate block computations and just consider it on the last computation where the last node will be the only one making a computation (therefore, this message will be sent non-concurrently $c_{total} - 1$ times).

Finally, the calculated entries of M should be returned to the root process for the determination of the best alignment. This can be accomplished with a All-to-one gather of N/P entire rows of the M matrix which takes the same time as the scatter:

$$t_{comm\_final} = t_s * log(P) + t_w * \left(\frac{N^2}{P}\right) * (P-1) \tag{13}$$

Combining everything and keeping in mind Equation 7, the total time required for distributing, computing and gathering M is given by:

$$\begin{aligned} t_{total} &= t_{comm\_initial} + t_{comm\_block} * (c_{total} - 1) + t_{comp\_block} * c_{total} + t_{comm\_final} \\ &= 3 * t_s * log(P) + t_w * \left(\frac{N^2}{P} + \frac{N}{P}\right) * (P-1) + \\ &\quad t_w * N * log(P) + \\ &\quad \left[\left(\frac{N}{P} * B\right) t_c\right] * \left(\frac{PN}{BY_{max}} + Y_{max} - 1\right) + \\ &\quad (t_s + B * t_w) * \left(\frac{PN}{BY_{max}} + Y_{max} - 2\right) \end{aligned} \tag{14}$$

Notice that when considering $Y_{max} = min(N/B, P)$, the parts between brackets referring to Y_{max} have the exact same value whether they are calculated with $Y_{max} =$

N/B or $Y_{max} = P$. Therefore, from this point on we can simply substitute Y_{max} for P:

$$t_{total} = 3 * t_s * log(P) + t_w * \left(\frac{N^2}{P} + \frac{N}{P}\right) * (P-1) + \\ t_w * N * log(P) + \\ \left[\left(\frac{N}{P} * B\right) t_c\right] * \left(\frac{N}{B} + P - 1\right) + (t_s + B * t_w) * \left(\frac{N}{B} + P - 2\right) \quad (15)$$

If we want to determine the best time we can get with fixed P and N, we have to determine the optimum value of B. To achieve this, we simply have to find the derivative of t_{total} in order of B (removing the constant subtractions for simplification):

$$\frac{dt_{total}(B)}{dB} = t_c N - \frac{t_s N}{B^2} + t_w P \quad (16)$$

And then equal it to 0 and solve the equation:

$$(t_c N - \frac{t_s N}{B_{opt}^2} + t_w P) = 0 \Leftrightarrow \\ B_{opt}^2 (t_c N + t_w P) = t_s N \Leftrightarrow \\ B_{opt} = \sqrt{\frac{t_s N}{t_c N + t_w P}} \quad (17)$$

5.1.2 Using different levels of interleaving at the row level + blocking sizes at column level

Considering interleaving at the row level in conjunction with blocks at the column level, nodes are no longer assigned contiguous areas of M. Being I the degree of interleaving, the number of contiguous rows assigned to each node is given by $N/(P * I)$, for a total of $P * I * N/B$ blocks of size $(N/(P * I), B)$. Figure 2 shows an example block distribution and dependencies based on the example of the previous section.

If $I = 1$ or $P >= N/B$, this case is not so different from the previous implementation considering a total of $P' = P * I$ different nodes doing the computation over N/B blocks per node. However, If $I > 1$ and $P < N/B$ as is the case of Figure 3 where $I = 2$ and $2 < 4$, it happens that as we navigate towards the steady stage, each node has to start computing multiple blocks per cycle considering the previous analysis of computation via diagonals. This happens because with this implementation, the maximum parallelism Y_{max}, doesn't correspond to the number of rows or the number of columns of M as occurred on the previous implementation. However in these situations, we can "rearrange" the computation of M, dividing it in $N/(BP)$ vertical partitions of blocks with each part containing at most P columns of blocks. Since each block only requires information from the previous vertical block from another node (information about the left block comes from the same node), we can rearrange M by putting all the aforementioned partitions on top of

each other. As Figure 4 shows, this has approximately the same behaviour as the previous implementation with a total of $P' = PIN/(BP) = IN/B$ nodes, P blocks per node.

Notice that given the 2 cases exposed on the previous paragraph and the comparison made with the previous implementation, we can construct the performance model for this implementation based on the previous one just considering the new "number of nodes" ($P' = I * X_{max}$ with $X_{max} = max(N/B, P)$) and new "number of blocks per node" (Y_{max}).

As such, some of the values remain the same between the 2 implementations. Such is the case of $Y_{max} = min(N/B, P)$ and, consequently, the growing and shrinking phase work:

$$c_{grow} = c_{shrink} = Y_{max} - 1$$

The steady state suffers some changes since the total number of blocks is no longer given by $P*(N/B)$ as considered in Equation 5 but is now given by $P'*Y_{max}$ $= I * X_{max} * Y_{max}$:

$$\begin{aligned} c_{steady} &= \frac{b_{total} - b_{grow} - b_{shrink}}{Y_{max}} \\ &= \frac{IX_{max}Y_{max} - 2\sum_{i=1}^{Y_{max}-1} i}{Y_{max}} \\ &= \left(IX_{max}Y_{max} - 2 * \frac{Y_{max} - 1}{2} * Y_{max} \right) * \frac{1}{Y_{max}} \\ &= IX_{max} - Y_{max} + 1 \end{aligned} \tag{18}$$

For a total of:

$$\begin{aligned} c_{total} &= 2(Y_{max} - 1) + IX_{max} - Y_{max} + 1 = \\ &= IX_{max} + Y_{max} - 1 \end{aligned} \tag{19}$$

Since each block is now smaller than without interleaving (since it has only $N/(PI)$ rows), computation time is now given by:

$$t_{comp\_block} = \left(\frac{N}{PI} * B \right) * t_c \tag{20}$$

The initial scattering/broadcasting and final gathering don't change since each real node will continue to handle the same number of rows, they'll just be scattered throughout M and not contiguous. Since we also continue to send just the last row of a block to the next node, the time to do this step remains unaltered as well.

Therefore, combining everything, we get:

$$\begin{aligned} t_{total} &= t_{comm\_initial} + t_{comm\_block} * (c_{total} - 1) + t_{comp\_block} * c_{total} + t_{comm\_final} \\ &= 3 * t_s * log(P) + t_w * \left(\frac{N^2}{P} + \frac{N}{P}\right) * (P-1) + \\ & \quad t_w * N * log(P) + \\ & \quad \left[\left(\frac{N}{PI} * B\right) t_c\right] * (IX_{max} + Y_{max} - 1) + \\ & \quad (t_s + B * t_w) * (IX_{max} + Y_{max} - 2) \end{aligned} \tag{21}$$

Which corresponds to the following branched function:

$$t_{total}(B, I) = \begin{cases} \frac{Bt_cN(-1+\frac{IN}{B}+P)}{IP} + (-2 + \frac{IN}{B} + P)(t_s + Bt_w) + \\ t_wNlog(P) + \\ 3t_slog(P)t_w\left(\frac{N^2}{P} + \frac{N}{P}\right)(P-1) & \text{, if } B <= N/P \\ \\ \frac{Bt_cN(-1+\frac{N}{B}+IP)}{IP} + (-2 + \frac{N}{B} + IP)(t_s + Bt_w) + \\ t_wNlog(P) + \\ 3t_slog(P)t_w\left(\frac{N^2}{P} + \frac{N}{P}\right)(P-1) & \text{, if } B > N/P \end{cases}$$

To find the optimum values of B and I we must calculate the partial derivatives of t_{total} with respect to B and I, equal them to 0 and solve the system. However, in the case of multi-dimensional functions such as this one, this is not always easy to do because there can be a great many deal of solutions and it is very difficult to determine a-priori through analytical methods not only their formula but also which ones make sense given the domain's boundaries (for example, $1 \leq B \leq N$ and $1 \leq I \leq N/P$ and B, I integers). Fortunately, those same boundaries make it easy to obtain the minimum value numerically which we will do in section 5. In any case, we leave in the following paragraphs the systems of interest.

Starting with $B <= N/P$, the partial derivatives of t_{total} are [1] [2]:

$$\frac{dt_{total}}{dB} = \frac{t_cN}{I} - \frac{INt_s}{B^2} + P * t_w \tag{22}$$

$$\frac{dt_{total}}{dI} = \frac{Nt_s}{B} - \frac{Bt_cN}{I^2} + N * t_w \tag{23}$$

And solving for 0[3]:

$$\begin{cases} \frac{dt_{total}}{dB} = 0 \\ \frac{dt_{total}}{dI} = 0 \end{cases} \Leftrightarrow \begin{cases} \frac{t_cN}{I} - \frac{INt_s}{B^2} + Pt_w = 0 \\ \frac{Nt_s}{B} - \frac{Bt_cN}{I^2} + Nt_w = 0 \end{cases} \Leftrightarrow \begin{cases} B = \sqrt{\frac{I^2t_sN}{t_cN + IPt_w}} \\ I = \sqrt{\frac{B^2t_cN}{Nt_s + Nt_wB}} \end{cases}$$

[1] Wolfram Alpha - Partial derivative of t_{total} in order to B

[2] Wolfram Alpha - Partial derivative of t_{total} in order to I

[3] Wolfram Alpha - Solutions for system of partial derivatives equal to 0

With $B > N/P$, the partial derivatives of t_{total} are [4] [5]

$$\frac{dt_{total}}{dB} = t_c N - \frac{N t_s}{B^2} + IPt_w \tag{24}$$

$$\frac{dt_{total}}{dI} = P(Bt_w + t_s) - \frac{t_c N^2}{I^2 P} \tag{25}$$

And solving for 0[6]:

$$\begin{cases} \frac{dt_{total}}{dB} = 0 \\ \frac{dt_{total}}{dI} = 0 \end{cases} \Leftrightarrow \begin{cases} t_c N - \frac{N t_s}{B^2} + IPt_w = 0 \\ P(Bt_w + t_s) - \frac{t_c N^2}{I^2 P} = 0 \end{cases} \Leftrightarrow \begin{cases} B = \sqrt{\frac{t_s N}{t_c N + IPt_w}} \\ I = \frac{N}{P}\sqrt{\frac{t_c}{t_s + t_w B}} \end{cases}$$

5.1.3 Different topologies

Considering different topologies such as 2D meshes or hypercubes the only thing that changes is the initial and final communication times, namely, the scattering, gathering and broadcasting operations.

According to [1, ch4.4], the scatter and gathering operations can be performed in the time shown in Equation 9 and Equation 13 in linear arrays, rings, hypercubes and 2D meshes. For any d-meshes, we obtain similar equations by making slight changes to just the t_s part.

As for the one-to-all broadcasting operations, according to [1, ch4.1] we can perform them in the following times for each of these topologies:

- **Ring/Line**

$$t = (t_s + t_w m) log(n) \tag{26}$$

- **2D Mesh/Torus**

$$t = 2 * (t_s + t_w m) * log(\sqrt{n}) \tag{27}$$

- **d-hypercubes**

$$t = d * (t_s + t_w m) \tag{28}$$

Where m is the size of the message being broadcasted.

5.2 OpenMP

As discussed in subsection 4.2, the implementation of the parallelization with OpenMP is very similar to the OpenMPI one minus the communication times (due

[4] Wolfram Alpha - Partial derivative of t_{total} in order to I (2^{nd} case)
[5] Wolfram Alpha - Partial derivative of t_{total} in order to B (2^{nd} case)
[6] Wolfram Alpha - Solutions for system of partial derivatives equal to 0 (2^{nd} case)

to the shared-memory). Therefore, the total time needed for a OpenMP computation can be given by:

$$\left[\left(\frac{N}{PI} * B\right) t_c\right] * (IX_{max} + Y_{max} - 1) \tag{29}$$

With P now representing the total number of threads available for computation and all other variables keeping the same values as in the OpenMPI case.

5.3 OpenMPI + OpenMP

Once again, the similarity between this implementation and the OpenMPI and OpenMP ones was already mentioned in a previous section (subsection 4.3). In a nutshell, it represents an extension of the OpenMPI implementation with an OpenMP parallelization at the sub-block level. This results in the existence of two distinct sets of variables: one for the blocks and another for the sub-blocks (represented with an apostrophe e.g B').

Each block has size $(\frac{N}{PI}, B)$. Therefore, each sub-block will have size $(\frac{N}{PIP'I'}, B')$. Considering this and the calculations performed on the previous sections, we know that the computation time per block is then given by:

$$t_{comp-block} = \left[\left(\frac{N}{PIP'I'} * B'\right) t_c\right] * (I'X'_{max} + Y'_{max} - 1) \tag{30}$$

$$\begin{aligned} t_{total} &= t_{comm\_initial} + t_{comm\_block} * (c_{total} - 1) + t_{comp\_block} * c_{total} + t_{comm\_final} \\ &= 3 * t_s * log(P) + t_w * \left(\frac{N^2}{P} + \frac{N}{P}\right) * (P - 1) + \\ &\quad t_w * N * log(P) + \\ &\quad \left[\left(\frac{N}{PIP'I'} * B'\right)\right] * (I'X'_{max} + Y'_{max} - 1) * (IX_{max} + Y_{max} - 1) + \\ &\quad (t_s + B * t_w) * (IX_{max} + Y_{max} - 2) \end{aligned} \tag{31}$$

6 Evaluation

In this section, we create the bridge between the theoretical performance models created on the previous section and the empirical data collected using the BSC GPU cluster - MinoTauro.

6.1 t_s and t_w estimation

In order to obtain estimates for the real value of t_s and t_w, we created a program that performed a total of 20000 MPI sends of an array of a well-defined size, measured the time taken for each send and averaged it over all sent messages. We ran this program with different array lengths from 1000 to 9000 and obtained the results that can be seen in Figure 8. The obtained results are quite strange. Where we expected a linear-alignment of the data points we can approximately see 3 different linear alignments: one in the 1000-3000 interval, another on the 4000-6000 interval and a third one on the 7000-9000 interval. This remained constant between multiple observations at different times so we do not think it is load related and assume it has to do with optimizations and buffering mechanisms implemented on the Minotauro system.

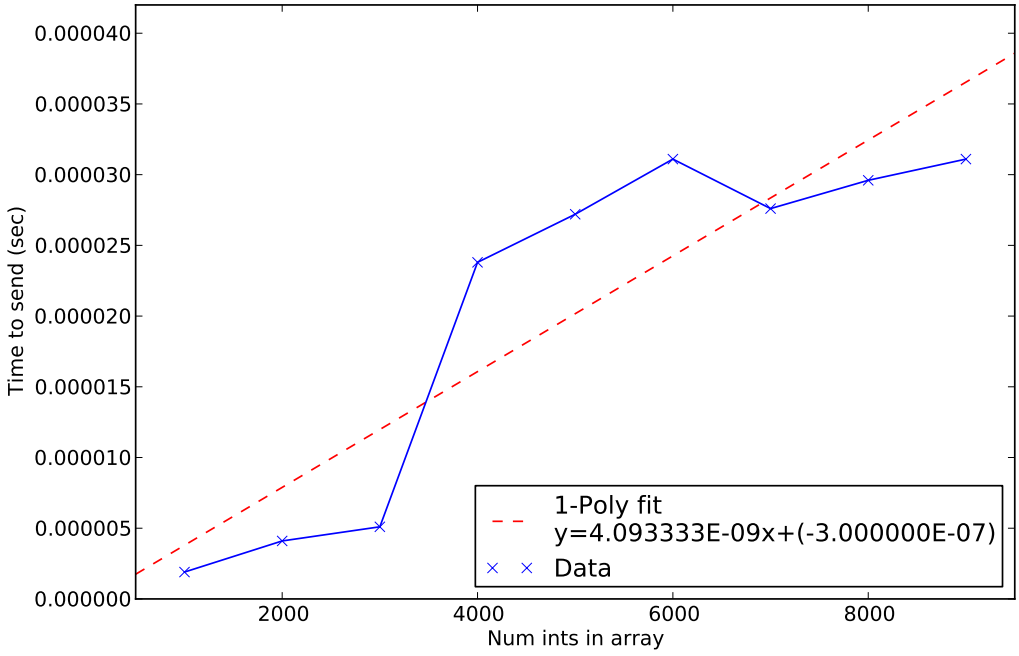

Figure 8: t_s and t_w parameter estimation obtained from MPI sends of int arrays of size 1000 to 9000 with an increment of 1000.

We could try to use the parameters obtained by the linear fit seen in red but it has a negative t_s value which doesn't make sense. Therefore, we will take the pessimistic approach and use the parameters obtained from the 4000-6000 interval

shown in Figure 9:

$$t_s = 9.116667 * 10^{-6} \text{ seconds}$$
$$t_w = 3.65 * 10^{-9} \text{ seconds}$$

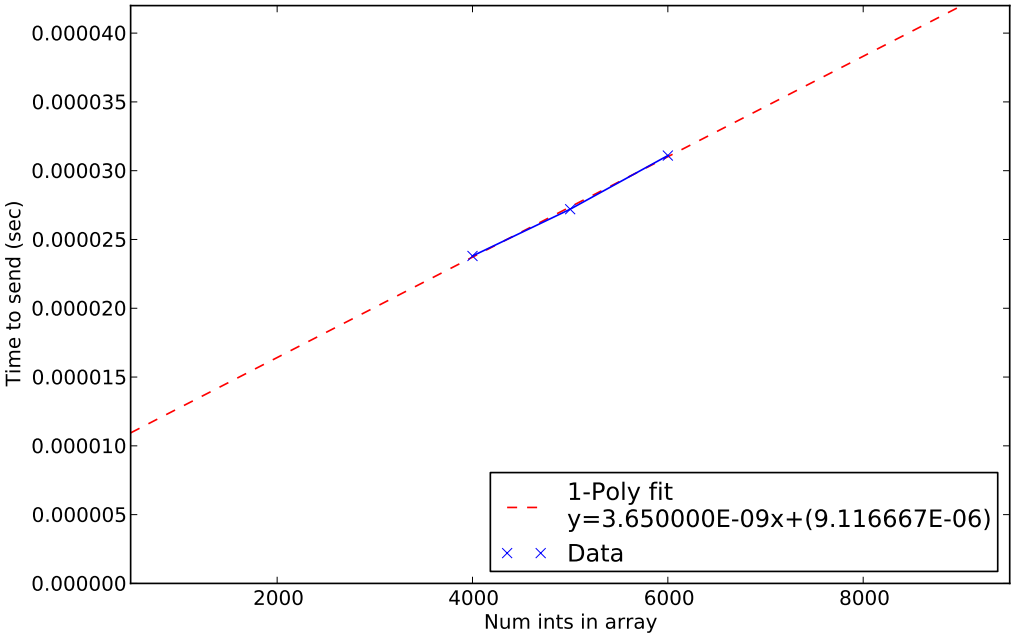

Figure 9: t_s and t_w parameter estimation obtained from MPI sends of int arrays of size 4000 to 6000 with an increment of 1000.

6.2 t_c estimation

To estimate the time it takes to calculate a cell of the M matrix (the t_c time therefore), we ran *sw-serial* with a sequence size of 10000 elements. We did a total of 5 executions, took the average value and obtained:

$$t_c = \frac{1 \text{ s} + 819719166 \text{ ns}}{10000^2} = 1.819719166 * 10^{-8} \text{ seconds}$$

6.3 Theoretical expectations

Armed with our estimations for the various times, we are now in a position to exercise our models with an example load. In this subsection, we're going to consider an execution over $P = 5$ nodes, $N = 10000$ rows and columns for M and the aforementioned times.

6.3.1 Using blocking sizes at the row level

The graph for the total estimated time considering the parameters described above can be seen in Figure 10.

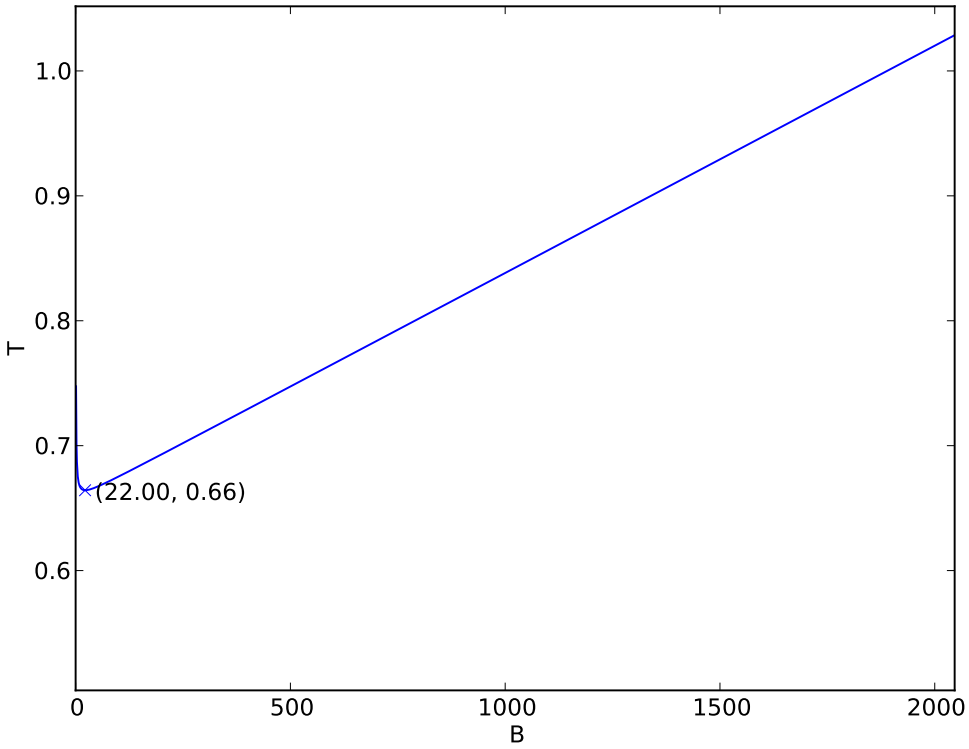

Figure 10: Total time as a function of B

The graphical minimum at $(22, 0.66)$ matches the expected minimum from Equation 17:

$$B_{opt} = \sqrt{\frac{9.116667 * 10^{-6} * 10000}{1.819719166 * 10^{-8} * 10000 + 3.65 * 10^{-9} * 5}} = 22.38175 \simeq 22 \qquad (32)$$

6.3.2 Using interleaving and blocking sizes at the row level

Now taking interleaving into account besides the previous blocking at the row level, things get more interesting as we now have to find a minimum over a 3d surface. Through the analysis of all valid values of B (1 to N) and I (1 to N/P), we determined numerically that the minimum point is once again associated with $B = 22$: $(B = 22, I = 1, T = 0.664306)$ according to the model. If, however, we force interleaving by considering a minimum $I = 2$, we get the following minimum time: $(B = 45, I = 2, T = 0.664341)$ which makes sense since the duplication of the width of a block (B) in conjunction with the halving of the height of a block $(N/(PI))$ results in blocks with the same size and, therefore, approximately the same amount of parallelization.

Figure 11 plots the entire 3d surface over the valid domain while Figure 12 and Figure 13 plot the area around the two specific points we mentioned on the previous paragraph.

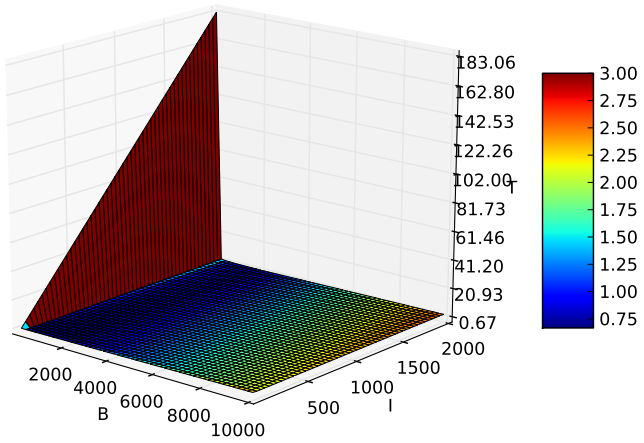

Figure 11: Total time as a function of B and I

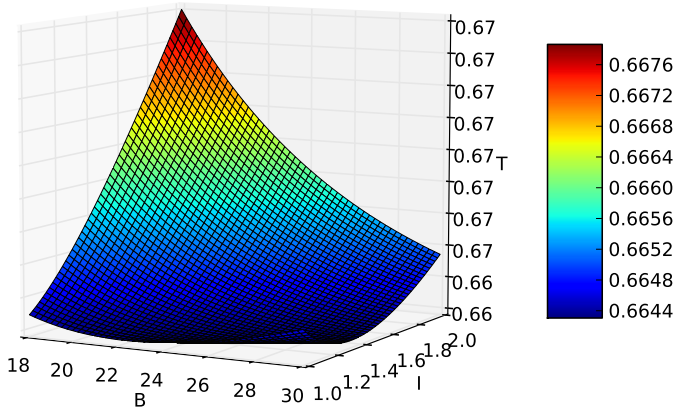

Figure 12: Total time as a function of B and I around the $(22, 1)$ point

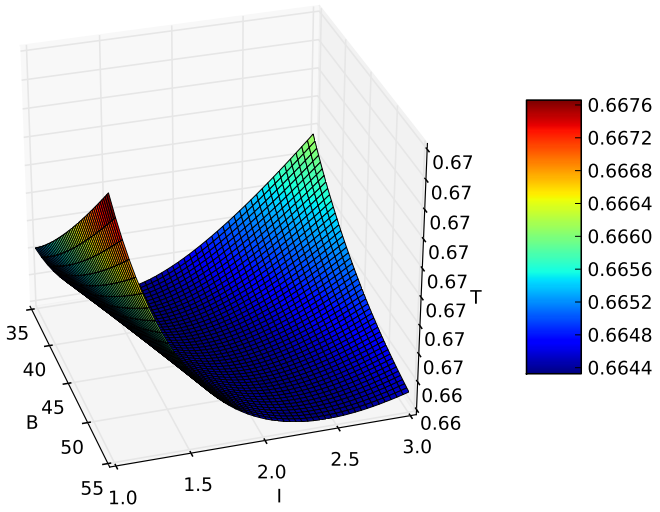

Figure 13: Total time as a function of B and I around the $(45, 2)$ point

It is curious to notice how interleaving starts being considered when we get to higher values of B and the individual block computations of each node start to get relatively heavy. This effect can be seen in Figure 14. However, as Figure 11 clearly shows, too much interleaving is prejudicial, specially at lower values of B where the computation time gets eclipsed by the increased communication times.

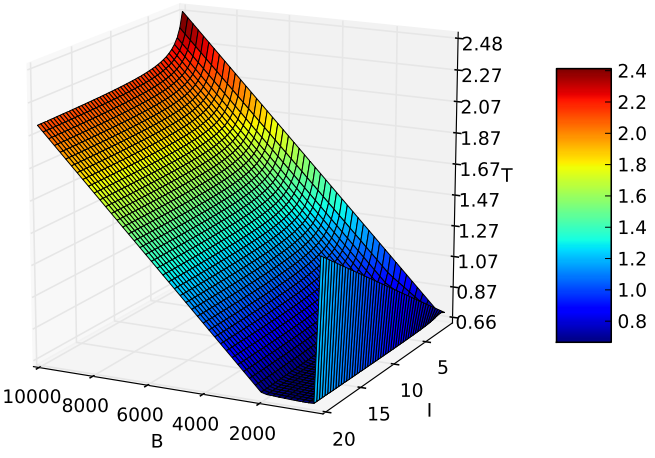

Figure 14: Increased benefit of interleaving at bigger B values

6.4 Finding optimum B and I experimentally

In this section we attempted to obtain experimental optimum values of B and I for both mpi-only implementations. After some experiments, we noticed that the maximum value of N we could test was 10000 with bigger values depleting the available 1.8GB limit of RAM memory on the minotauro nodes. Therefore, on the following experiments we used a N of 10000. As for the number of nodes, we used the same number of nodes used on the theoretical analysis, i.e, $P = 5$.

6.4.1 Using different blocking sizes at column level

To try and determine experimentally the optimum B value for an execution considering only blocking at the column level and no interleaving we executed several runs of our implementation without interleaving with various values of B from 10 to 4000 (with greater values, there isn't much change because our implementation notices there are not enough columns to form a new block and just considers the whole 10000 columns). For each value of B we ran the experiment 5 times and obtained the average of the total execution time of the computation of the matrix. The obtained data can be seen graphically on Figure 15.

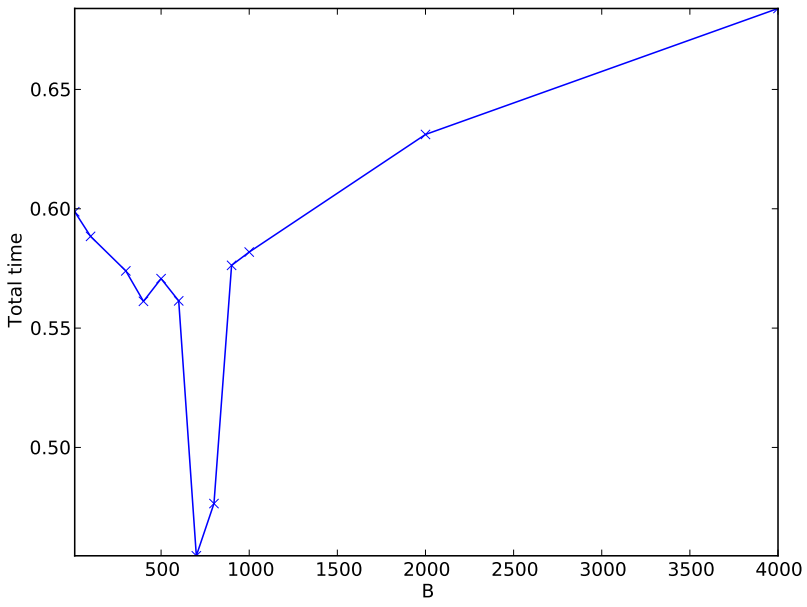

Figure 15: Experimental data for multiple values of B with the mpi-implementation using solely blocking at column level

From the collected data, it seems that the optimum value of B is located somewhere in the vicinity of B = 700. Unfortunately, due to the variability of load conditions, it is very hard to obtain a precise number. In fact, we also came across execution sets where the entire interval from 300-900 had approximately the same

total time. However, what remained constant is that very low values of B and very high values of B give the worst times.

The results obtained here seem to contradict the theoretical predictions made on the previous section. This can be explained by the fact that the theoretical predictions are too simplistic, assuming perfect synchronization of every node and constant base performance which doesn't happen on any practical system much less on a complex one such as Minotauro. We can, however, make an interesting experiment of trying to integrate (in a simplified manner) synchronization delays into our theoretical model by considering a greater value for t_s. Instead of $9.116667 * 10^{-6}$ let us consider that $t_s = 9.11 * 10^{-3}$ (for example). The new theoretical prediction can be seen on Figure 16 and appears to be more according to the experimental results.

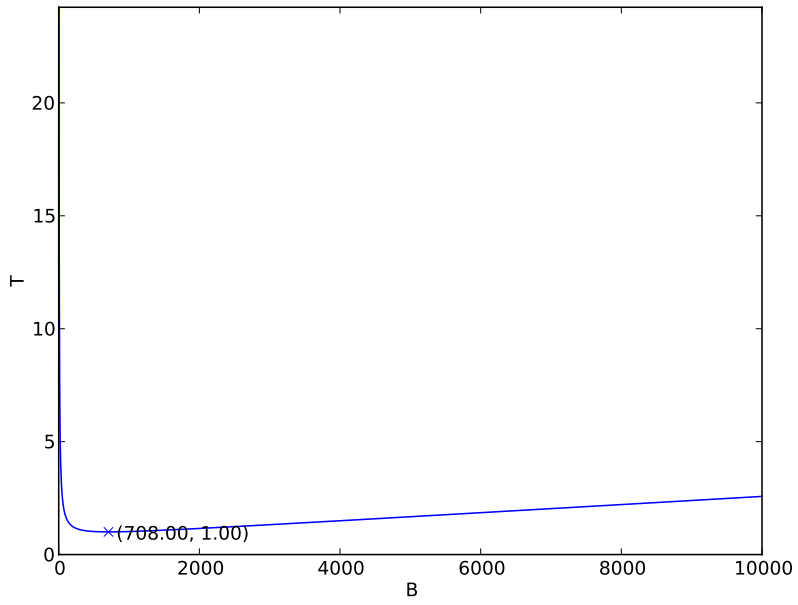

Figure 16: New theoretical optimum value of B with inflated t_s

6.4.2 Using different levels of interleaving at the row level + blocking sizes at column level

To determine optimum B and I when considering interleaving along with blocking at column level, we procceeded similarly to the non-interleaving case but now considered different values of I. Let us start by replicating the theoretical determination of optimum B and I with the inflated value of t_s taking into account node synchronization just as we did on the previous section. This wields optimum values of $B = 708$ and $I = 1$. It seems that, in the conditions of the experiment, interleaving won't introduce any benefit. The collected data can be seen graphically in Figure 17 and appears to support this claim.

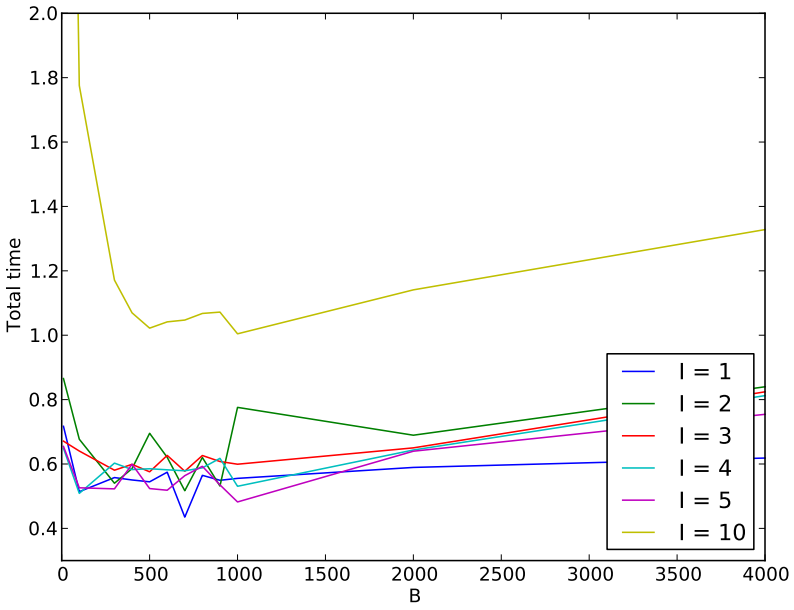

Figure 17: Experimental data for multiple values of B and I with the mpi-implementation with blocking at column level and interleaving

It is interesting to notice how, with the increase of interleaving, the minimum time shifts to ever increasing values of B. This could already be seen in the "diagonal valley" present on Figure 12 and Figure 13. Also worth of mention is the fact that very high interleaving levels (such as $I = 10$) have a great impact on the computation (at least in the conditions under which the experiment was ran).

6.5 Effect of number of nodes in total time

In this section, we do a quick experiment to observe the effects of varying P maintaining everything else fixed. Since in the last sections we determined that interleaving was not very relevant, we will only look at the effects of this on the implementation with no interleaving. Intuitively, one might expect that with very low values of P, we won't obtain as much parallelism as with higher values and, therefore, the total time will be higher. On the other hand, too high values of P result in more communications between borders (just as what happens with more interleaving) and should also wield higher total times.

Keeping in mind our reflections in the previous paragraph, it is to expect that the best execution time will be achieved with a balanced P - one that isn't very low nor very high. Therefore, what we expect is for the graph of total times to show a parabola-like shape pointing up (that is, with a minimum). And, not surprisingly, this was exactly what we got once we ran the experiments and graphed the results (Figure 18).

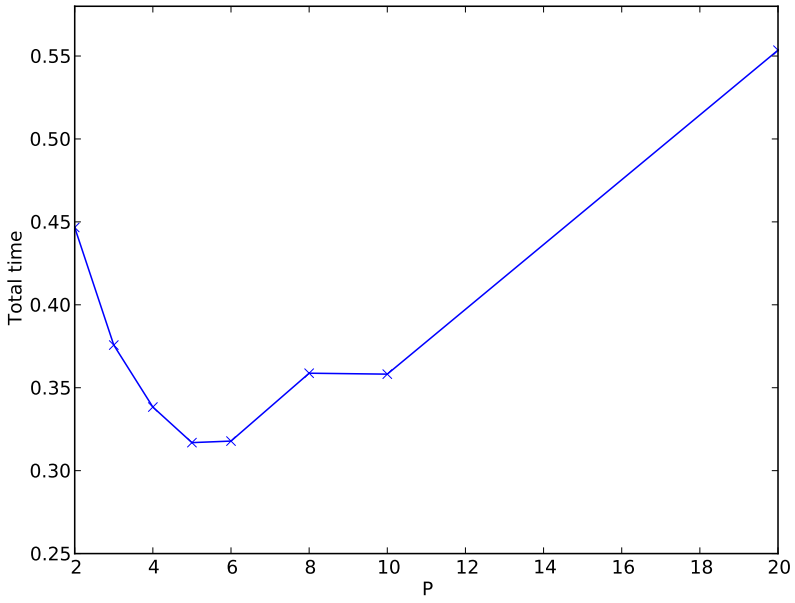

Figure 18: Total time as a function of P for the implementation with no interleaving and fixed $B = 708$ and $N = 10000$

6.6 Comparison between implementations

To have an overview of the performance of the different implementations, we executed them with different values of N, ranging from 50 to 10000. The number of processes (or threads in the case of OpenMP implementation) is set to 5 in all cases.
For all parallel implementations, we attempt to obtain the optimal total time for each value of N by adjusting the other variables, i.e.: B, I, B_x, B_y.

- **OpenMP:**
  For simplicity, we keep $B_x = B_y$. For each value of N, we try with a few (from 6 to 12) different values of $B_x = B_y$. The best result for each N is used in plotting the final graph.

- **MPI without Interleaving:**
  With the result obtained from 6.4.1, we proceed the experiment for this implementation with the optimal value of B for each N, i.e.: $B = 708$ when $N > 2000$ and B ranging from 694 to 707 when $N \leq 2000$.
  However, for $N \leq 1000$, it does not make sense to have $B > 500$ since with our implementation, each process will execute its part as one whole block and no parallelization is achieved. Therefore, for these cases, we try with a few different values of B for each case of N and use the best result in our graph.

- **MPI combined with OpenMP:**
  Showing the results for this implementation is a bit tricky as we have three different variables to adjust:

- B, the block size for the MPI part
- B_x and B_y, the horizontal and vertical block size of the OpenMP part.

To keep things manageable, we use once again the optimal value of B from previous experiments, i.e.: $B = 708$ for $N > 1000$.
In order to determine the optimal value of B_x, B_y given $B = 708$, we ran a few tests with the OpenMP implementation given $N = 708$ and varying B_x, B_y. For simplicity, we still keep $B_x = B_y$ as in previous experiments. The graph below shows the result obtained.

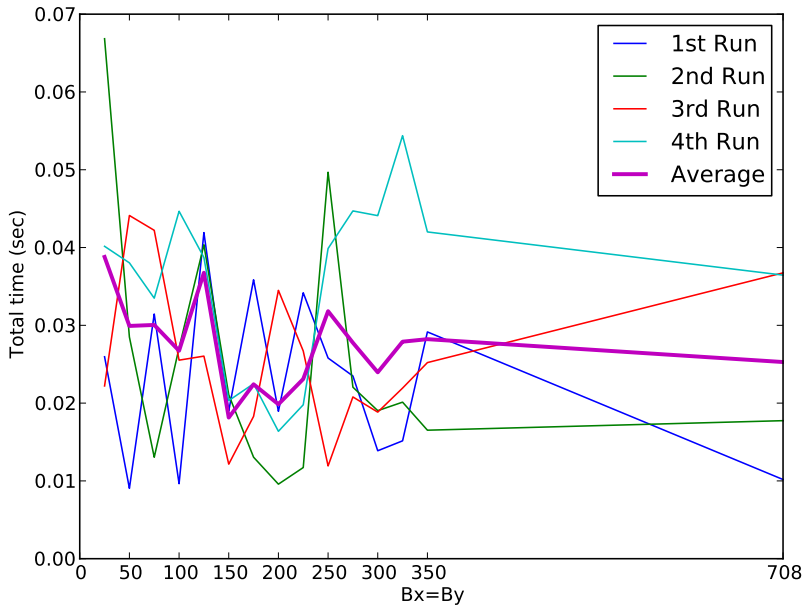

Figure 19: Experimental data for OpenMP implementation with different values of B_x= B_y, given $N = 708$.

From the graph, we can hardly tell which is the optimal value of B_x, B_y. However, $Bx = B_y = 150$ seems slightly better than other cases so we decided to proceed the experiments for $N > 1000$ with this value.
For each case where $N \leq 1000$, we ran some executions while varying B and $B_x = B_y$ (3 different values of B_x or B_y for each B and around 6 different values of B for each N). The best result obtained for each value of N is included in the final graph.

- **MPI with Interleaving:**
  Using the optimal values obtained from 6.4.2 but ensuring there is interleaving, we obtain $B = 708$ and $I = 2$ and use those values in in our executions with $N > 1000$. For other values of N, we run a few tests with $I = 2$ while varying B. The best result for each value of N is used in the final graph.

With the settings as described, we obtain the following graph showing the relative performance of different implementations.

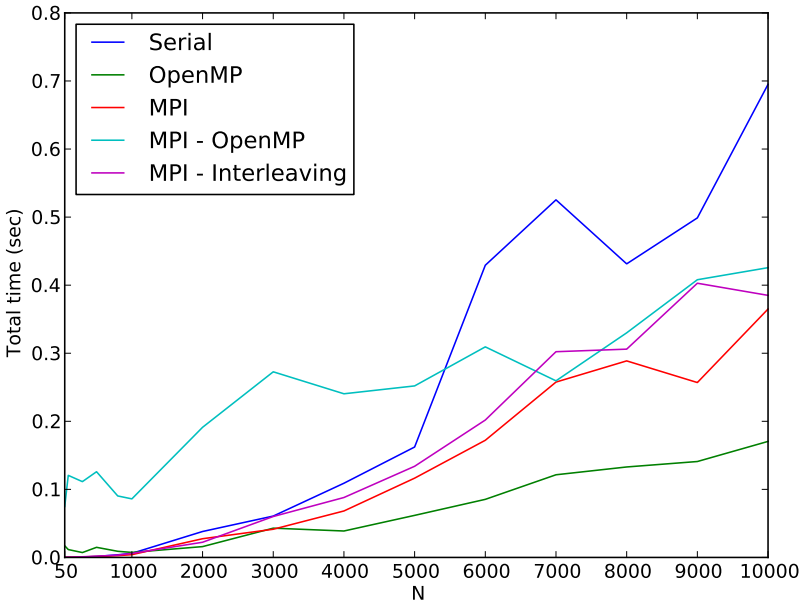

Figure 20: Experimental data for different implementations with different N

From the graph, we observe some interesting points:

- For small N, little improvement is gained from parallelizing the program, indeed parallelizing it can even worsen the performance (i.e.: $N < 1000$). It is expected as there are communication cost and overhead in the parallel versions. The improvement increases as N increases.
- Among the different parallel implementations, the OpenMP one has the best performance as it does not introduce communication cost.
- The MPI with interleaving actually performs worse than the non-interleaving one.
- The implementation with combination of MPI and OpenMP performs worse than the pure MPI version. This actually agrees with the fact that for $N < 1000$, the OpenMP implementation gives a negative improvement compared to the Serial one due to the overhead it introduces. For this implementation, the overhead is even repeated for each block and cause a big gap between the plot for MPI version and its plot as can be seen from the graph. However, should we use a bigger N or a smaller P, it is expected that this implementation will become considerably faster than the others.

7 Conclusion

All implemented code appears to be working correctly and gives the same results as that generated by the original code. This is motivating in that it shows that we correctly understood how to parallelize programs using both OpenMPI and OpenMP, the two main parallelization frameworks studied in this course. The greatest difficulties we found on the multiple implementations was dealing with certain details such as non-exact integer divisions that left rows/columns out and calculations of displacements, specially on the interleaved version.

As for the theoretical models and consequent comparison with the experimental results, the obtained results agreed more or less with our expectations: the simplified construction of the performance model that we employed on this report and that was lectured on the classes fails to take into account synchronization issues even more so in complex systems such as Minotauro. However, after artificially inflating t_s to a higher value to simulate the consideratin of these issues, the theoretical models came quite close to what was seen in practice.

The very nature of the Minotauro system as a multi-node networked system employing virtualization at the node level and the variable load to which it was subjected at different times made the collection of accurate results rather difficult. The variability was too big to be able to draw any precise conclusions as to the experimental optimum values of B and I. The fact that we were limited to 1.8GB of RAM memory in each node also didn't help as we were unable to experiment with bigger values of N that might have resulted in more meaningful results. In addition, as was seen in subsection 6.1, the communication cost doesn't vary (approximately) linearly with the size of the message. While we cannot be certain as to why this occurs, we can only assume that buffers and optimizations at the network level were responsible for this phenomenon which introduces yet another source of noise when compared with the theoretical expectations.

As for our main findings, through our experiments we came to the conclusion that if we have the ability to choose both B and I, interleaving isn't very useful to solving the problem and may actually be prejudicial. The optimum experimental values seem to be located around the $(B, I) = (708, 1)$ point. However, if for any reason we are forced to use big values of B, interleaving may indeed help us reduce the total computation time.

References

[1] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. Introduction to Parallel Computing (2nd Edition). Addison Wesley, 2 edition, January 2003.

Appendix A Compiling and running

Compiling any of our implementations is as simple as using the provided Makefile:

- make sw-original
- make sw-serial
- make sw-mpi
- make sw-mpi-interleaved
- make sw-omp
- make sw-mpi-omp

Our source codes also include debug code which can be activated with the DEBUG=1 flag like so:

make DEBUG=1 sw-mpi

For time-related statistics, there's also the TIMER=1 flag which will report the total time spent and broken down by the main sections.

make TIMER=1 sw-mpi

As for running the programs after compilation, it's a simple matter of running one of the following instructions, substituting the variables between square brackets by the intended values.

- ./sw-serial [N]
- ./sw-mpi [N] [B]
- ./sw-mpi-interleaved [N] [B] [I]
- ./sw-omp [BX] [BY]
- ./sw-mpi-omp [N] [B] [BX] [BY]

Appendix B Used scripts

In order to speed up the collection of data and the obtainment of results, we used several scripts which we included with the report:

- **2dgraph.py**, **2dgraph-multiple.py**
  These scripts generate graphs based on data following the format that can be seen on the many examples in the *graph-data* folder.

  To execute, run 'python 2dgraph.py graph-data/some-graph-data'.

  Dependencies: Python, NumPy, SciPy, Matplotlib

- **experiment-maker.py**
  This script generates experiment scripts and folders according to the variables defined at the beginning of the file and based on the *experiment-template.sh* file.

  To execute, edit the desired variables at the beginning of the file and then run 'python experiment-maker.py'

  Dependencies: Python

- **submit-all-jobs.sh**
  This script searches for all .sh files in the current folder and subfolders and submits it for processing in Minotauro. It is copied to every experiment folder upon running the experiment-maker script. It should be run on the Minotauro server.

  Dependencies: Bash

- **sync_exp_mino_to_local.sh**
  This script synchronizes the entire experiments folder located at minotauro with the local experiments folder. It should be run locally.

  Dependencies: Bash

- **time-interleave-mpi.py**
  This script uses the formulas calculated during the theoretical performance model section to generate graphs and find minimums using numerical methods.

  To execute, edit the desired variables at the beginning of the file and then run 'python time-interleave-mpi.py [B] [I] [plot]'.

  B and I should be replaced by the desired values of the respective variables or by \? to analyze all valid ones. If 'plot' is included when both B and I are \?, then a 3d surface graph will be generated. Otherwise, numerical minimum analysis will be run. With specific values of B or I plot is always run and if both values are set, the correspondent time is directly outputted.

  Dependencies: Python, NumPy, Matplotlib

- **timesanalyzer.py**
  This script is meant to be used with the outputs generated by our implementations compiled with the TIMER=1 flag and what it does is calculate the average of each different measured time over several iterations. This allows the use of a single output file for multiple runs of an experiment with the same arguments.

  To execute, run 'python timesanalyzer.py experiments/.../someoutput.out'

  Dependencies: Python

Appendix C Additional files

We also included all collected outputs and data used to generate the graphs. These can be seen, respectively, in the *experiments* and *graph-data* folders. Due to

the big size of these files and the great amount of data they contain, we decided we would obtain a cleaner report by just showing the graphical representation of the data and provide the files with the report if the reader desires a more detailed analysis.