Performance Evaluation and Modeling of Smith-Waterman Algorithm on HPC Plateform

***Abstract*—Smith-Waterman algorithm is primarily used in DNA and protein sequencing which helps by a local sequence alignment to determine similarities between biomolecule sequences. However the inefficiency in performance of this algorithm limits its applications in the real world. In this perspective, this work presents two fold contribution. It evaluates in detail the performance scalability of smith-waterman algorithm using OpenMP, MPI and a Hybrid (OpenMP + MPI) parallel programming models on Altix-4700 supercomputing platform. This evaluation shows that the hybrid approach performs better than the other simple approaches. Secondly it develops and evaluates a mathematical performance model for the algorithm by targeting a distributed processing system. This mathematical model can be helpful to estimate performance of the algorithm for larger size of sequences aligned by the thread level parallelism, using large set of processors configured as distributed processing nodes.**

I. Introduction

In sequence alignment problems, normally, it is not possible to perfectly match two sequences therefore the most basic sequence analysis task remains to ask if two sequences are related to each other in some arrangement. The Needleman and Wunsch algorithm [1] is normally referred as the first rapid method in the biological literature for determining sequence homology. However the algorithm was kept on improving till Smith-Waterman et al. [2] showed an improved and more sensitive algorithm for local alignment by following the same idea employed by Needleman and Wunsch.

Smith-Waterman algorithm which is based on dynamic programming provides a high sensitivity but it makes the algorithm more computationally intense with such data dependencies that restrict it to easily scale for parallel implimentations [3] [4] . The time complexity of this algorithm for comparing two sequences is O(mn), where m and n are the lengths of the two sequences being compared. Although this computational complexity may not seem threatening, the growth in the genetic bio-sequence database is exponential. Thus, the complexity that concerns the real world applications is really O(Pmn), where P represents the exponential growth of the size in genetic databases. This exponential expansion of the genetic sequence databases motivates this research.

Our goal is to develop parallel implimentation methodologies that can best utilize the available computational power provided in a multiprocessor platform. As an evaluation platform, we used Altix-4700 multiprocessor supercomputing machine [5] to achieve highly-efficient genetic sequence searching. SGI Altix is a shared memory machine, with a cc-NUMA architecture (Cache Coherent Non- Uniform Memory Access). As the architecture of the Altix system is a single image shared memory that well suits for the OpenMP programming model but in order to evaluate our MPI and hybrid models implementations, we consider the platform as a set of distributed nodes while each node contains multiple (single for MPI only Implimentations) shared memory processors. The distributed nodes are accessed using MPI based interface while the processors within a node used OpenMP interface. The mathematical model is developed for MPI based layout while considering that each node has processing power like a single processor.

Next in Section-I-A, we will briefly introduce the Smith-Waterman (SW) Algorithm which is followed by some description of related work for accelerating the algorithm. In section-III and section-IV, we will present the parallele implimentations and evaluations of smith-waterman algorithm for MPI, openMP and Hybrid methodologies. Section-V will present and evaluate the MPI based mathematical model of the SW-Algorithm.

A. The Smith-Waterman Algorithm

Smith-Waterman algorithm, based on the dynamic computational matrix technique, is used to compute the optimal local alignment of two sequences. The procedure consists of following three steps:

1) Fill in the dynamic programming matrix.
2) Find the maximal value (score) in the matrix.
3) Trace back the path that leads to the maximal score to find the optimal local alignment.

Let S = s1, s2, - - -, sm of length m and T = t1, t2, - - -, tn of length n be the two sequences for comparison. The basic,Smith-Waterman algorithm for calculating the best score between the two sequences S and T is to construct a $(m+1)\times(n+1)$ matrix DM, named the dynamic alignment matrix or similarity matrix. DM is indexed by i and j, where each index is for one sequence. In particular, a cell DM_{ij} can be built iteratively or recursively using the following relations.

For

$$0 \leqslant j \leqslant k, 0 \leqslant m \leqslant n$$

$$DM_{j0} = DM_{0k} = 0$$

For

$$1 \leqslant j \leqslant m, 1 \leqslant k \leqslant n$$

$$DM_{jk} = MAX\{0, DM_{j-1,k-1} + Sc(S_j, S_k),$$

$$DM_{j-1,k} - Cost_{gap}, DM_{j,k-1} - Cost_{gap}\}$$

The gap cost in above is the penalty for inserting a gap character -. This could be s_i aligned to a gap (-) or t_j aligned to a gap (-). The value of scoring matrix Sc(si, tj) is for scoring a match or a mismatch. Applying the equation, all values of the matrix can be obtained. Note that this step takes O(mn). The second step in the procedure is to find the maximal value in the matrix. Once the maximal value is found, the third step can be taken, which traces back from the maximal value until a 0 value is reached. Focus of this work is on parallelization methodologies for the first step beacause other two steps are either less computational expensive or strictly sequential.

II. Related Work

The Smith Waterman algorithm for sequence alignment is one of the main bioinformatics algorithms therefore large number of efforts have been made to accelerate this algorithm using both homogeneous and heterogeneous accelerator architectures and as well using application specific accelerators.

The work presented in [6] used a Distributed Shared Memory (DSM) system. The work is focused on more algorithmic strategies to improve the performance of SW Algorithm however the performance evaluation is done on eight machine cluster which creates a shared memory abstraction that parallel processes of an application can access. The work used wavefront method and work load was assigned in a column basis.In contrast to our MPI and SMP based approach, synchronization was achieved with locks, condition variables barriers using JIAJIA (a cache coherence protocol).

A white paper from Altera [7] describes an implimentation of the SW-Algorithm based on the XD1000 reconfigurable supercomputing platform. The implimentation presents a multistage PE (processing element) design that consists of 384-PE systolic array working at 66.7 MHz. Performance of FPGA based implimentation is compared to a 2.2-GHz AMD64 Opteron host processor of the XD1000 platform. The evaluation presented shows a range of speed-ups (2x to 250x) with small gains for very small sequences and excellent speed-up for large sequences.

Another FPGA based Smith-Waterman accelerator is presented in [8]. This work implements a basic processing module for computing the score of a single cell of the SW matrix. Then this module is instentiated as a grid and the entire SW matrix is computed at the speed of field propagation through the FPGA circuit. The evaluation of this implimentation shows a acceleration up to 160 folds compared to a pure software implementation running on the same FPGA with an Altera Nios II softprocessor.

[9] presents acceleration of smith-waterman algorithm using Graphical Processing Unit (GPU [10]). This work claims an efficient code implimentation with performance improvement up to 3.5x over the previous implimentations of SW algorithm using GPUs reaching 70% of theoratical peak performance of the GPU.

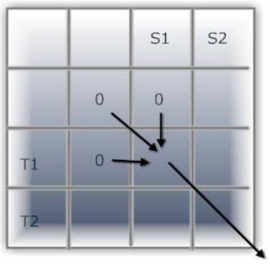


Fig. 1. Smith-Waterman dependencies

III. OpenMP, MPI & Hybrid Programming Methodology

A. OpenMP

OpenMP is a programming model for shared memory architectures. It is also an Application Program Interface (API) that could be used to explicitly direct multi-threaded, shared memory parallelism for a set of processing nodes. It comprised of three primary API components “Compiler Directives”, “Runtime Library Routines” and “Environment Variables”. This model can deal with fine grain memory sharing with relatively very small overhead as compared to MPI based model. In our SW-Algorithm implimentation we experimented the OpenMP based parallelism using various sizes of blocks for filling dynamic programming matrix in parallel by different processors. Performance evaluation section presents the sizes of the block dimensions used for parallel programming of the dynamic matrix.

B. MPI

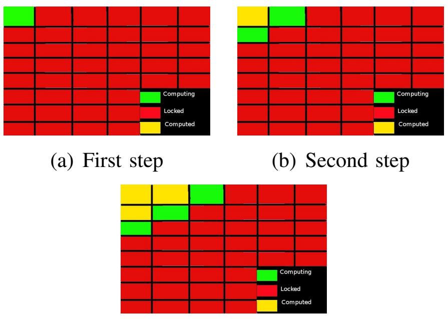
MPI (Message Passing Interface) is another programming model, focused on distributed processing systems. MPI also works as a specification for the developers and users of message passing libraries. It helps to write a message passing programs for system with processing nodes configured for message passing interfaces. The experimental system (Altix-4700) used in our work is a shared memory architecture however the MPI interface is supported on this machine which enables us to observe the effects of MPI based implimentation for SW-Algorithm as presented in the performance evaluation section.

C. Hybrid (OpenMP+MPI)

In parallel implementation, the positive slope diagonal entries as shown in the Figure-2 can be computed simultaneously. The final edit distance between the two strings appears in the bottom right table entry. While calculating the similarity matrix, the score of any matrix element always depends on the score of three other elements (Figure-1):

- The up-left neighbor element
- The left neighbor
- The up neighbor

Therefore, the calculation sequence of the similarity matrix will be as shown in Figure-1. It begins from the top-left element to bottom-right element according to the direction as



(a) First step (b) Second step

(c) Third step

Fig. 2. Matrix filling methodology

shown by the arrow. Through observation of the similarity matrix calculation process, we found that for each thread, every element on an anti-diagonal line marked with the same number could be calculated simultaneously, with the standing for the elements that could be calculated at the same time.For example, in the first cycle, only one element could be calculated.In the second cycle, two elements could be calculated. In the third cycle, three elements could be calculated, etc., and this feature implies that the algorithm has a very good potential for parallelism.

Computation begins from the top-left element to bottom-right element. Mixed mode programming OpenMP [11] and MPI [12] provide a more efficient and scalable parallelization strategy than pure MPI or OpenMP based designs. By utilizing a mixed mode programming model we should be able to take advantage of the benefits of both models. For example a mixed mode program may allow us to make use of the explicit control data placement policies of MPI with the finer grain parallelism of OpenMP. The majority of mixed mode applications involve a hierarchical model, MPI parallelization occurring at the top level, and OpenMP parallelization occurring below. For example, Figure-2 shows OpenMP based parallelization at one node and Figure-3 below shows a Hybrid Model for 2D grid which has been divided geometrically between MPI processes. These sub-arrays have then been further divided between OpenMP threads. This model closely maps to the architecture of an SMP cluster (NODE), the MPI parallelism occurring between the SMP boxes and the OpenMP parallelization within the boxes.

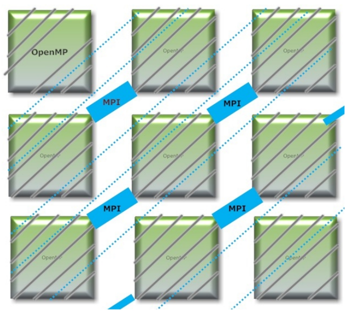


Fig. 3. OpenMP and MPI Mix Mode programming methodology

IV. Performance Analysis

In following we have experimented and compared Smith Waterman parallel algorithm using OpenMP and MPI programming models with the hybrid model.

A. OpenMP Based SW-Algorithm

B. Initial parallelization

The initial OpenMP implementation parallelized the diagonals using a *brute force* approach to compute every single element of each diagonal in parallel. This approach provided some improvement over the sequential version (see Tables I and II).

Input Size	Sequential
10000	≈ 1.5s
20000	≈ 5.9s
30000	≈ 13.3s

TABLE I
Elapsed loop time – sequential version

Input Size	# Threads	OpenMP
10000	12	1.08s
	24	1.45s
	32	1.61s
20000	12	2.59s
	24	2.60s
	32	3.22s
30000	12	5.14s
	24	5.41s
	32	5.28s

TABLE II
Elapsed loop time – initial OpenMP version

However, as it can be observed in II, there is little or no improvement when running this code using more threads. One of the reasons for the lack of improvement may be precisely the fact that each task is doing very little work, so the overhead of using OpenMP isn't worth it.

C. Parallelization using submatrices

In order to improve the parallelization even further, and take advantage of the available hardware, a second OpenMP version was implemented. Instead of each task computing a single element of the matrix, in this version we split all the work into smaller submatrices (or *chunks*).

As it can be observed in Table III, using submatrices of size 2x2 already provided some improvement over the initial parallelization (equivalent to a submatrix of 1x1), so we executed it using a wide range of submatrix sizes in order to evaluate which one was the optimal size.

Figures 4, 5 and 6 show the speedup for a different number of threads using different input sizes. Also, note that each line of these figures represents a different submatrix size, ranging from 1 to 1000.

Input Size	# Threads	OpenMP
10000	12	0.69s
	24	0.89s
20000	12	1.83s
	24	1.98s
30000	12	3.92s
	24	3.73s

TABLE III
ELAPSED LOOP TIME – OPENMP VERSION, SUBMATRIX SIZE = 2X2

One of the first noticeable differences between these figures is that the input size has a big impact on the achieved speedup. While for relatively small inputs like 10000 it is stabilizes at around 5 at most, and even declines for 24 or more threads, for larger inputs it grows much faster and is able to get a speedup of up to 16. It should also be noted that it is hard to get any improvement if the number of threads is small (4 or less).

Overall, 250x250 seems to be the optimal submatrix size for the selected input sizes.

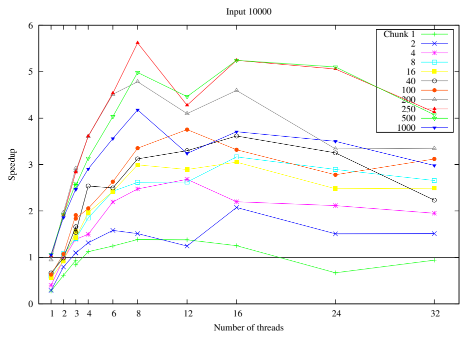


Fig. 4. OpenMP: speedup (input size 10000)

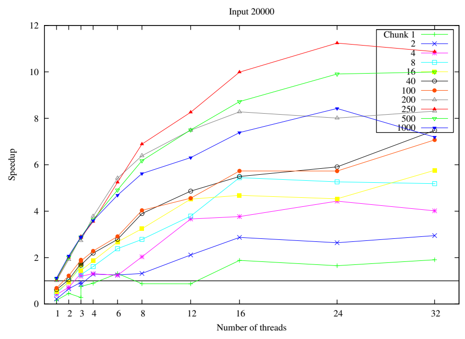


Fig. 5. OpenMP: speedup (input size 20000)

D. MPI Based SW-Algorithm

In order to evaluate the MPI version, we measured the computation time using different values of B and I (Submatrix Dimensions).

In a first experiment (Figure 7), the I was set to 2000 and B changed from 250 to 4000. Gap penalty was set to -2. As can be observed on the graph, the best times are achieved for

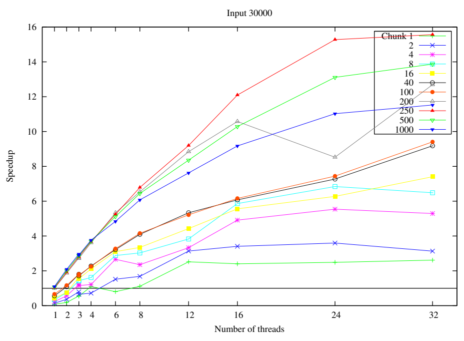


Fig. 6. OpenMP: speedup (input size 30000)

the B equal to I, that is, the computation blocks are squares. In all other cases computation time increases.

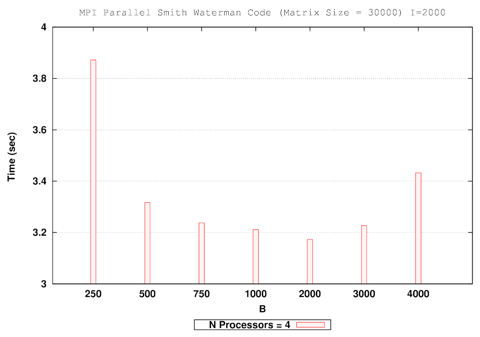


Fig. 7. MPI: execution time changing the value of B

In a second experiment (Figure 8), the value of B was set to 2000, and it was I that changed from 250 to 4000. In this case it seems that there is no parabolic behavior, as parameter I is increasing so computation time is also increasing. We are still not completely sure why this is happening.

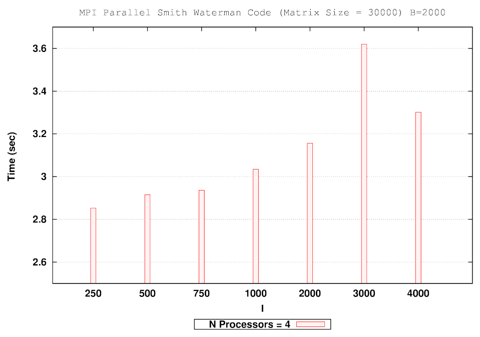


Fig. 8. MPI: execution time changing the value of I

Finally, we also evaluated how the execution times improved using different square block sizes. The bar chart in Figure 9 shows that over X-axis we have number of processor (uP) and Block Size as inputs. Over Y-axis we have time (t in seconds) consumed by each block. Its clearly seen that maximum time is taken when we are running program in sequential mode that is with 1 number of processor. Minimum time is consumed when we are using 32 number of processor except for 5000x5000 block size, with this we are getting marginally good time with 16 uP. There is not much difference in time for blocks 250x250, 500x500 and 1000x1000 while applying different number of processors over them.

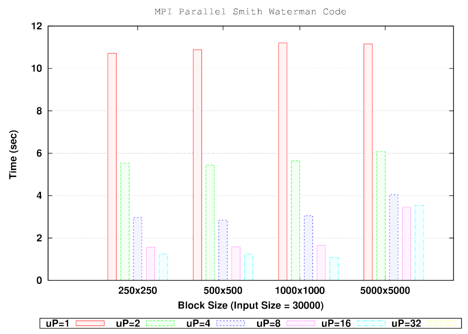


Fig. 9. MPI: execution time using different number of processors

E. Hybrid (MPI+OpenMP) Based SW-Algorithm

For the hybrid MPI & OpenMP version, we simply used the base MPI code, but parallelizing the inner loop of each block using OpenMP, as described in III.

Figures 10, 11 and 12 show the elapsed times of various configurations. Each point represents the time it takes to execute the code with the specified MPI block size and OpenMP chunk size within each MPI block. For instance, 500-100 means that the total input is first *split* into blocks of 500x500 (MPI part), and then each one of these blocks is *split* into chunks of 100x100 (OpenMP part).

These figures measure the computation time only and are just intended to provide an initial exploration of the multiple parameters that may affect the execution, but overall, one of the first conclusions is that the ratio of block size to chunk size shouldn't be too high.

Note that the OpenMP performance analysis described in Section IV-A of the OpenMP version was specially focused on the parallelization of the full matrix, but the size of the matrix OpenMP is dealing with in this hybrid version is much smaller. This is an important aspect that should also be taken into account. Figure 13 shows the speedup of an isolated execution of the OpenMP code with an input size of 2000, with 1, 2 and 4 threads, and as it can be observed, in most cases there is no speedup at all. Using MPI and OpenMP together doesn't guarantee performance improvements *per se*, so the parameters should be carefully studied. As shown in Figure 14, which is limited in input size and MPI block size (20000 and 500 respectively), the distribution of resources between MPI and OpenMP also has impact on the performance. With this setup and 30 tasks, the best configuration is using an OpenMP chunk size of 250, with 15 MPI threads and 2 OpenMP threads.

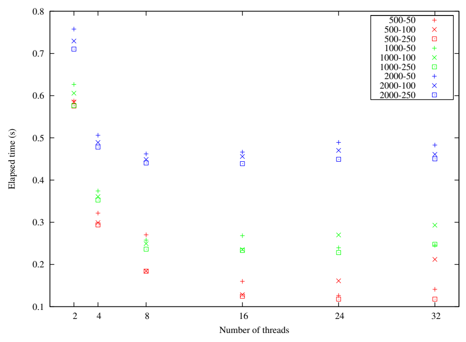


Fig. 10. MPI & OpenMP: elapsed times (input size 10000)

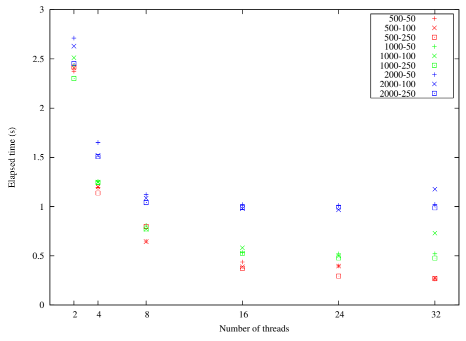


Fig. 11. MPI & OpenMP: elapsed times (input size 20000)

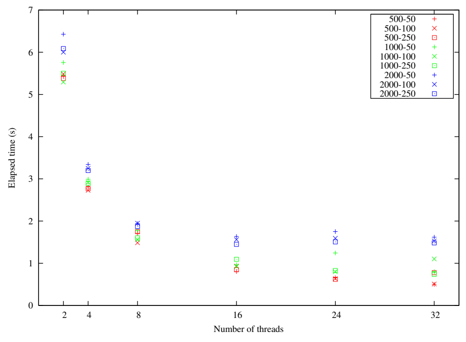


Fig. 12. MPI & OpenMP: elapsed times (input size 30000)

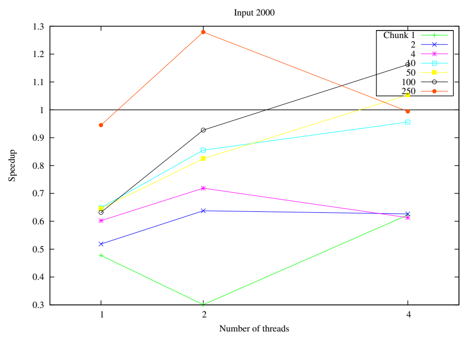


Fig. 13. OpenMP: speedup (input size 2000)

F. Gap Penalties

Finally, another aspect of the Smith-Waterman algorithm that should be taken into account when parallelizing the code are gap penalties. Figure 15 shows the times of the same kind of execution (OpenMP version, input size of 30000), varying

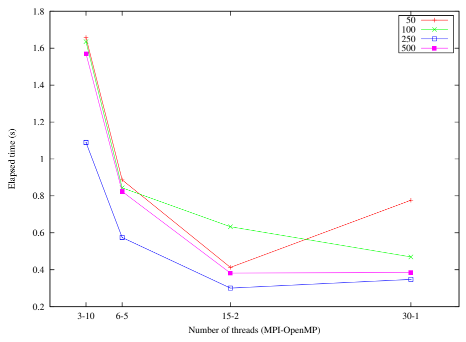


Fig. 14. MPI & OpenMP: elapsed times with different number of MPI and OpenMP threads and varying OpenMP chunk size (input size and MPI block size set to 20000 and 500 respectively)

the gap penalty: -4, -2, 0, 2 and 4. The code is in general slower for negative penalties, but what should be remarked here is that there is variation rather than which one is faster.

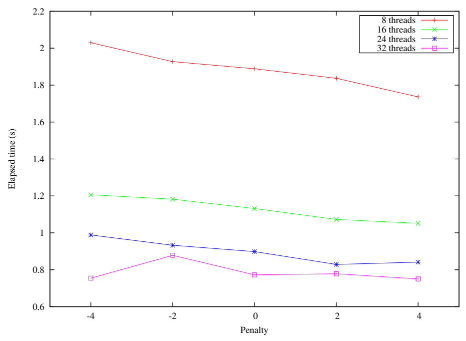


Fig. 15. Gap Penalties

V. Smith Waterman Performance Model

We will proceed by modeling [13] a simple Generic program and proceed towards the Model for MPI based Smith Waterman Algorithm.

A. Understanding a Simple Case

Suppose an algorithm takes a two dimensional matrix as input and does some operation on its each element and produces a same size of matrix at the output. Figure-16 shows a simple execution model in form of three phases. Assume

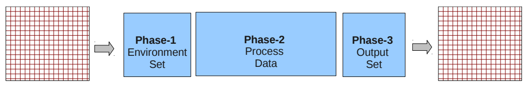


Fig. 16. Generic Program parts

that phases 1 and 3 are sequential and takes a constant time K while time taken by phase-2 depends on

1) Size of the matrix
2) How many processors are doing the job

Assume T is the total execution time of the algorithm and T_{seq}, T_{comp}, T_{comm} are respectively the three **time components** for the sequential part, Parallel computational part and the communication between multiprocessors. Then a simple generic computational model for a program would be :

$$T = T_{seq} + T_{comp} + T_{comm}$$

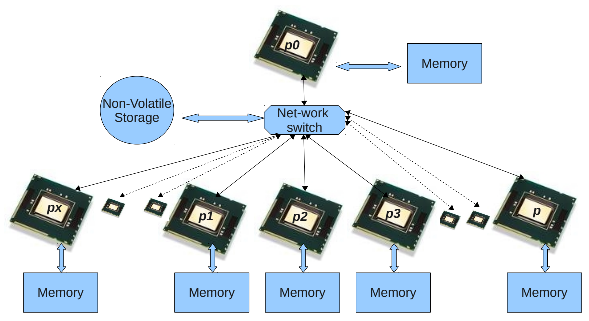


Fig. 17. A Simple Model of Microprocessor Network

In order to make a model for of our simple algorithm, it needs to take care about the programming model being used and as well the program layout. Here we are considering that MPI is the programming model used and all processors in the system can communicate in full-duplex mode along with any processor able to send its data to any other processor/s through the Network-Switch. In Figure-17 we also assume that all processing cores are symmetric in nature. Suppose that the computations in phase-2 (Figure-16) is independent to each other for our simple example and p processors are taking part into the execution. Also suppose that t_c is the computation time for computing one element and the input matrix is a square matrix of size N^2 . Suppose *p0* is managing the algorithm execution then *p0* has to distribute and collect data from other *p-1* processors in the system. Therefore :

$T_{seq} = K$
$T_{comp} = (t_c \times N^2)/p$
$T_{comm} = T_{tx} + T_{rx}$

where T_{tx} is send time from *p0* and T_{rx} is the receive time at *p0* from *p-1* processors. Suppose t_x is the time to send one word on the communication channel.

Then at *p0*
$T_{tx} = (N^2/p) \times t_x$
because P-1 processors can receive data at the same time and
$T_{rx} = (N^2/p) \times (p-1) \times t_x$
because of the sequential collection of data from p-1 processors.

The proximate model for our simple algorithm would be
$T = K + (t_c \times N^2)/p + (N^2/p) \times p \times t_x$

B. Modeling Smith Waterman Algorithm

The MPI based multiprocessor execution of Smith Waterman algorithm during phase-2 (Figure-16) uses execution pattern as shown in Figure-18

The generic model for this Smith Waterman algorithm is the same as for our general case

$T = T_{seq} + T_{comp} + T_{comm}$

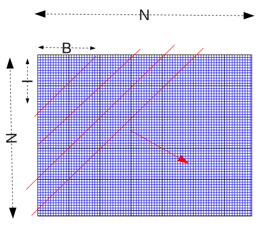


Fig. 18. Flow of Matrix Computation and I & B Parameters Conventions

We are considering that the scoring matrix is a square matrix with each dimension of size *N*. If the matrix is not square, data padding like option can be used to make the matrix square one. Suppose that $N \times N$ matrix is divided into sub-blocks. *I* and *B* are the rows and columns in a sub-block of the matrix. We are also assuming that I and B are multiple factors of *N*.

Assume that p is the number processors being used to execute the MPI program. Further In our multiprocessor execution, we consider that each processor will execute horizontally in the matrix. Therefore N/B blocks will be processed by each processor in a row of blocks and $(N/I)/p$ rows of blocks will be processed with one processor. Here we have assumed that all processors would have equal share in processing of sub-blocks however it is not a necessary condition.

Each processor will proceed computing a sub-block (computing points inside of a sub-block) in column wise order. This means, all points in the first column of the sub-block (*I* points) would be processed first and then computations are shifted to the next column of the same block. Shifting to next sub-block would lead to sending the *B* points in the lowest row of the processed block to the next processor. We also consider that $p > N/B$ would ideally have the same execution time like if $p = N/B$ therefore $p > N/B$ might be only is wastage of resources.

For T_{comm}, according to the algorithm an arbitrary p_r processor will send *B* words after computing current sub-block. The processor processing last row will not send any data to any where there for p-1 processors would be involved in sending the data. Moreover *p0* processor will accumulate data from all *p-1* processors.

$T_{seq} = K$

$T_{comp} = [(I \times B)N/B \times N/(I \times p) + (p-1) \times (I \times B)]t_c$

$T_{comp} = [N^2/p + (p-1) \times (I \times B)] \times t_c$

$T_{comm} = [(N^2/(I \times p) + (p-1) \times B)] \times t_x + [(N^2/(I \times B \times p))] \times \alpha$

where α is the setup overhead time per sub-block.

Therefore

$T = K + [N^2/p + (p-1) \times (I \times B)] \times t_c + [(N^2/(I \times p) + (p-1) \times B)] \times t_x + [(N^2/(I \times B \times p))] \times \alpha$

We assume that the time for collecting data at *p0* from all other *p-1* processors would always be t_x *times* the Number of points computed on processors other than p0 (if $p > 1$) plus some overhead. This as we think, will be due to all sequential transfer to *p0*.

C. Model Verification

In order to verify the model, we used NumPy and SciPy which are used to do numerical and scientific computing in the most natural way with Python. "numpy.array" and "numpy.matrix" are used to handle linear algebra problems like the "least squares estimation' which is useful when we have a set of unknown parameters in a mathematical model. We want to estimate parameters that minimize the errors between the model and our data. The canonical form is:

$Y = Xb$

which corresponds to our model

$T = K + [N^2/p + (p-1) \times (I \times B)] \times t_c + [(N^2/(I \times p) + (p-1) \times B)] \times t_x + [(N^2/(I \times B \times p))] \times \alpha$

as $y = b_0 + x_1 \times b_1 + x_2 \times b_2 + x_3 \times b_3$ where

x_1, x_2 and x_3 are known variables as follows

$x_1 = [N^2/p + (p-1) \times (I \times B)]$

$x_2 = [(N^2/(I \times p) + (p-2) \times B)]$

$x_3 = [(N^2/(I \times B \times p))]$

The model contains unknown variable b_0, b_1, b_2 and b_3 corresponding to

$b_0 = K$

$b_1 = t_c$: Computation Time

$b_2 = t_x$: Communication Time

$b_3 = \alpha$: Data Compilation Overhead

D. Model Result

In order to verify the model we did huge number of runs for the Smith Waterman MPI code however due to noise in time measurements in Altix-47000 environment we ultimately used results for 378 executions of the parallel code. Matrix Dimensions are taken 10,000, 20,000, 30,000. Number of microprocessors used were 2,4,8,16,24,32. The I and B parameters were varied between 100 and 2000 point for both.

The obtained values for $[b_0, b_1, b_2, b_3]$ are:

$[1.47969721e - 1, 9.31158435e - 9, -4.24248181e - 7, 1.50310908e - 3]$

In the following figures (19, 20, 21), the solid lines shows empirically obtained data for different executions and dashed lines are the computed values based on our model and computed coefficients. We can see that our model results are following measured very well.

E. Model Accuracy

The modeled results follow the empirical results precisely with average error less than 5%. However we observe jumps in the modeled curves around central points of the resultant plots. If we closely observe the trends of jumps for the three cases with Matrix Dimensions 10,000, 20,000 and 30,000, the jump value decreases with increase in matrix dimensions. Moreover the jump values increases with increase of number of threads. This somehow makes sense because for central points on size-axis, blocks (sub-matrix) dimensions are maximum which reduces the total number of sub-matrices that could be generated from the main matrix size thus number of blocks available are become even lesser than the total number of threads (processors) available which , in-fact, cause the increase in the execution time for filling the dynamic programming matrix. The difference between empirical implimentation and the modeled implimentation is as mentioned in section-V, the model is designed such that the actual matrix dimensions shall be a multiple for the corresponding dimensions of sub-matrices however the original implimentation can even handle non-multiple sub-matrices.

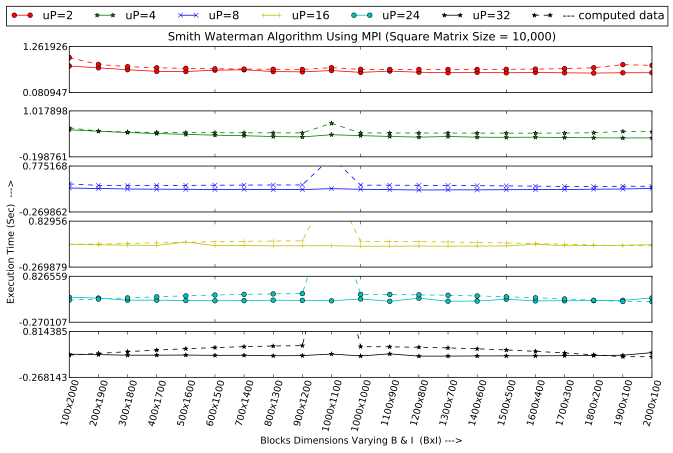


Fig. 19. Model results (Square Matrix Dimensions = 10,000)

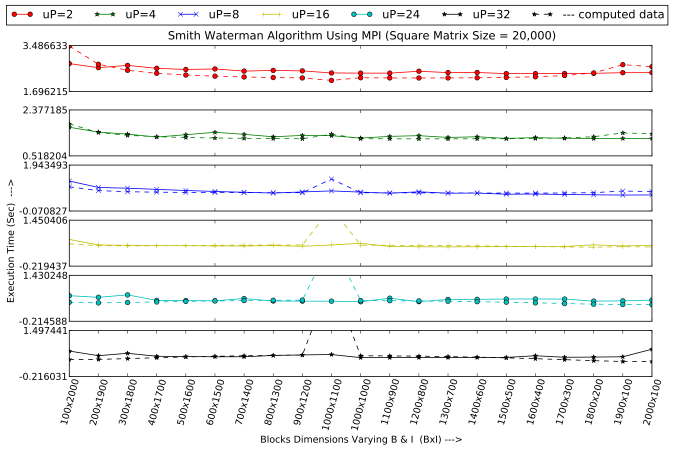


Fig. 20. Model results (Square Matrix Dimensions = 20,000)

VI. Conclusions

This work describes in detail, how we have implemented different parallelized versions of the original Smith-Waterman algorithm with OpenMP and MPI, as well as the hybrid OpenMP + MPI on Altix-4700 supercomputing system. OpenMP provides an easy and fast way to parallelize code, while MPI offers a much more flexible approach that works on a wide range of hardware but also requires effort for manual decomposing of the problem and dealing with communication. Both platforms make it possible to radically improve the performance of applications, and it is possible to improve it even further if they are combined. As we have shown, we are able to achieve the best results with the hybrid version, but it should also be noted that in addition to the development time it also requires some fine-tuning of the optimal sizes into which the problem is decomposed. Other than performance evaluation, we also presented a mathematical treatment of the problem which can be greatly helpful to estimate a very close to actual performance on a larger set of processors without any real execution.

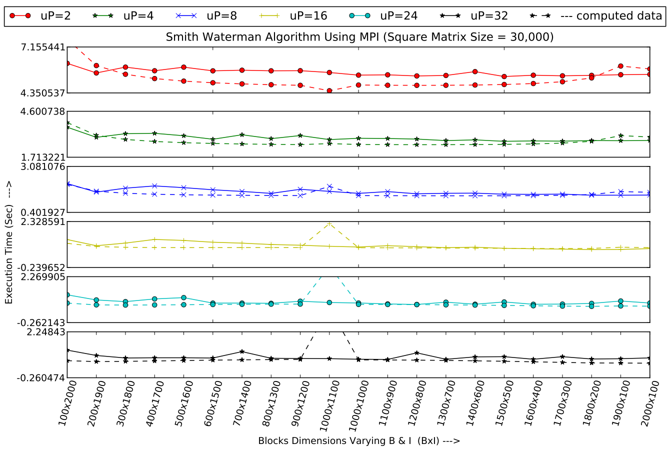


Fig. 21. Model results (Square Matrix Dimensions = 30,000)

References

[1] S. B. Needleman and C. D. Wunsch, “A general method applicable to search for similarities in the amino acid sequence of two proteins,” *Journal of Molecular Biology*, 1970.
[2] T. F. Smith and M. S. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, 1981.
[3] *T. F. Smith and M. S. Watermann. Identification of common molecular subse- quence. Journal of Molecular Biology, 147:196–197, 1981.*
[4] S. US, “A smith-waterman systolic cell,” 2005.
[5] SGI, “Reconfigurable Application-Specific Computing User Guide,” vol. 007-4718-007, 2008.
[6] A. Boukerche and et al, “Parallel smith-waterman algorithm for local dna comparison in a cluster of workstations,” *Book Experimental and Efficient Algorithms*, 2005. [Online]. Available: http://www.springerlink.com/content/xwn2q2qfm4hgvr3t
[7] “Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform,” *Altera White Paper*, September 2007.
[8] I. Li, W. Shum, and K. Truong, “160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga),” *BMC Bioinformatics*, vol. 8, no. 1, p. 185, 2007. [Online]. Available: http://www.biomedcentral.com/1471-2105/8/185
[9] L. Ligowski and W. Rudnicki, “An efficient implementation of smith waterman algorithm on gpu using cuda, for massively parallel scanning of sequence databases,” in *IPDPS ’09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8.
[10] K. Bjorke, “Introduction to image processing on the gpu,” *NVIDIA White Paper*, 2005. [Online]. Available: http://http.download.nvidia.com/developer/Papers/2005/Image_Processing/GPU-Img-Proc-Intro.pdf
[11] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP Portable Shared Memory Parallel Programming*, 2008.
[12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI, 2nd Edition Portable Parallel Programming with the Message Passing Interface*.
[13] B. D. Theelen, *Performance Modelling for System-Level Design*, 2004.