International Journal of advanced studies in Computer Science and Engineering
IJASCSE Volume 3, Issue 6, 2014

Performance Analysis of OpenMP and MPI for NW algorithm on multicore architecture

SHRIMANKAR. D. D, SATHE. S. R.
Department of Computer Science Engineering
Visvesvaraya National Institute of Technology
South Ambazari Road Nagpur, India

Abstract: **- Multicore Architectures now-a-days consist of cluster of SMP nodes. OpenMP and MPI programming paradigms can be used for parallelization of codes for such architectures. OpenMP uses shared memory, and hence is viewed as a simpler programming paradigm than MPI which is primarily a distributed memory paradigm. However, the OpenMP applications may not scale beyond one SMP node. On the other hand, if we use pure MPI we can scale using more SMP nodes, but it might introduce overhead in inter-node communication. In this paper, we analyse the performance of OpenMP and MPI paradigms for the same application. In particular, we look at a basic Needleman-Wunsch sequence alignment algorithm and parallelize it using variable size sequences and tile sizes on multicore architecture. We demonstrate that the overhead from communications in OpenMP loops on an SMP node is significant and increases with the number of cores participating in execution of the loop compared to equivalent MPI implementations. In some cases MPI implementation gives better performance than the OpenMP implementation, but in other cases OpenMP implementation performs better than the MPI counterpart and requires less programming effort as well. To analyse this result, we also present a simple method on how to estimate the overhead of communication in OpenMP loops. Our results are both surprising and of great interest to a different variety of input sequence size, file size, number of threads and processes.**

Key-Words: **- Parallel, OpenMP, MPI, Sequence Alignment, Multicore**

I Introduction

Programming for parallel computing has been dominated by the MPI and OpenMP programming paradigms. Both of the paradigms aim to provide an interface for high performance, but their approach is somewhat different. OpenMP is designed for share memory systems, and has recently gained popularity because of its simple interface. With little effort, loops can easily be parallelized, and much of the synchronization and data sharing is hidden from the user. MPI is designed for distributed memory and is probably the best known paradigm in parallel computing. Communication between processes is done explicitly, and a relatively large set of functions in the API opens up for high performance and tweaking which is not available in OpenMP. Even though it is designed for distributed memory systems, it runs just as good on shared memory systems.

This work searches for the best configuration of OpenMP and MPI for optimal performance. We run a parallel biological application on a modern multicore architecture to measure the effects of the parallel code. After analysing the performance of different configurations from a hardware point of view, we proposed a general model for estimating overhead in OpenMP loops.

On the other hand, the sizes of the sequence files in biological data are so huge that sequence alignment using a sequential algorithm is out of question. Also generic sequence searching becomes one of the most heavily used operations in computational biology [1, 2, 3]. In particular, the size of GenBank/ EMBL/DDBJ doubles every 15 months [4]. Therefore analysing generic databases with such a constant growth, raises a challenge for

scientist, in respect of being time consuming, expensive and impractical[5]. Needleman-Wunsch[7] is one the most significant and widely-used similarity algorithms for biological sequence comparison that adopts the dynamic programming method [1]. despite its high sensitivity in identifying best global alignments, it is very time consuming and computationally expensive process. This algorithm requires quadratic time for each comparison of two sequences[6]. Therefore, because of the complexity of this algorithm, there is a need for a methodology that could reduce the computation time while delivering accurate results. In this study, two programming paradigms are being compared for the parallelization of Needleman-Wunsch algorithm on a cluster of SMP nodes. Specifically, an evaluation between different type of implementation paradigm such as pure OpenMP and pure MPI is provided in terms of execution time and speedup.

The remainder of this paper is organized as follows. In the next section the multicore architecture and Performance Modelling is presented. In section 3 the original NW algorithm is described in detailed while in section 4 our method of parallelization and methodology is described and discussed. Section 5 explains the experimental results and comparisons and finally the last section will conclude the paper.

II Multicore Architecture and Performance Modelling

Multicore Architecture comes in many forms and are in this work roughly divided into two groups: distributed memory systems and shared memory systems. The respective groups are described and illustrated with simplified figures in this section (Fig. 1 and 2).

2.1 OpenMP

Shared memory architectures open up for efficient use of threads and shared memory programming models. The traditional way to take advantage of such architectures is to use threads in some way or other. POSIX threads (pthreads) are mostly used in HPC programming; however thread models can quickly generate complex and unreadable code. Recent advances in processor architectures with several cores on the same chip have made shared memory programming models more interesting. Especially the introduction of dual-core processors for desktops and multicore for workstation and servers, the last decade has made this area of research interesting for other groups than the HPC communities.

OpenMP[14] is an API that supports an easy-to-use shared memory programming model. The easiness of inserting OpenMP directives into the parallel code has made this model popular compared to pthreads. This model leaves most of the work of thread handling to the compiler and greatly reduces the complexity of the code. The directives for parallelization in OpenMP allows the user to decide what variables should be shared and which of them should be private in an easy way, in addition to what parts of the code that should be parallelized. The simplicity of using OpenMP directives has made it a popular way of parallelizing applications.

Shared memory systems (Fig.1) are systems with more than one processor where all the processor share memory. These systems are well suited for OpenMP, MPI, or mixed OpenMP-MPI programming models. Each processor sees the memory as one large memory.

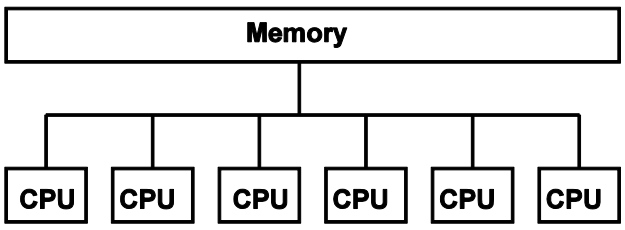


Fig.1. A Shared Memory System

2.2 The Message Passing Interface – MPI

MPI[8] [9] is an industry standard for message passing communication for applications running on both shared and distributed memory systems. MPI allows the programmer to manage communication between processes on distributed memory systems. The MPI is an interface standard for what an MPI-implementation should provide such as functions

and what these functions should do. There are several implementations of MPI and the best known are probably MPICH and OpenMPI, both open source version implementations. Most vendors of HPC resources also have their own proprietary implementation of MPI with bindings for C, C++ and Fortran. SCALI [10] is probably the best known Norwegian vendor of MPI implementations. The first MPI standard was presented at Supercomputing 1994 and finalized soon thereafter. The first standard included a language independent specification in addition to specifications for ANSI-C and Fortran-77.About 128 functions are included in the MPI 1.2 specification. This interface provides functions for the programmer to distribute data, synchronize processes and create virtual topologies for communication between processes.

Distributed memory systems (Fig.2) share

		Sequence : Y						
		A	T	G	C	A	G	T
	0	-1	-2	-3	-4	-5	-6	-7
A	-1	1	0	-1	-2	-3	-4	-5
T	-2	0	2	1	0	-1	-2	-3
A	-3	-1	1	2	1	1	0	-1
A	-4	-2	0	1	2	2	1	0
G	-5	-3	-1	1	1	2	3	2
T	-6	-4	-2	0	1	1	2	4

Sequence : X

interconnection but have private processor and memory. These systems are well suited for message-passing libraries like MPI.

Fig.3. Alignment of two sequences with score = 4.

III. The Original Needleman-Wunsch algorithm

In order to implement this algorithm either the task or the data can be divided between processors. Task decomposition is about breaking down the job in different parts and assign each part of job to a specific processor while in data decomposition, all processors apply the same job on different portion of data. As a matter of fact, in most cases the nature of algorithm and study would determine which type of decomposition has to take place.

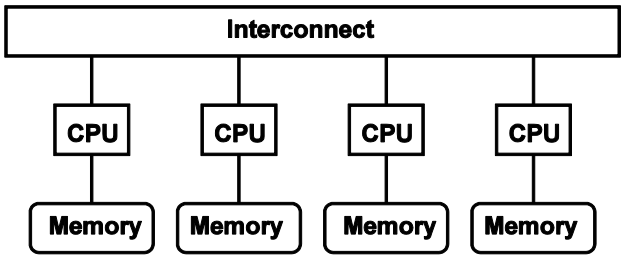


Fig.2. A Distributed Memory System

For this algorithm, data decomposition parallel approach is utilized in both OpenMP and MPI implementation. Here the data is divided according to size of the tile specified.

Sequence X	A	T	A	-	A	G	T
Sequence Y	A	T	G	C	A	G	T
Score	1	1	-1	0	1	1	1

Total = 4

3.1 NW Algorithm for sequence alignment

The algorithm consists of two parts: the calculation of the total score indicating the similarity between the two given sequences, and the identification of the alignments that lead to the score. In this work we have concentrated on the calculation of the score, since this is the most computationally expensive part. The algorithm finds alignments by comparing entire sequences. The sequences are placed along the left margin (X) and on the top (Y). The matrix as shown in Figure 3 is initialized with decreasing values (0, -1, -2, -3 ……)

along the first row and first column to penalize for consecutive gaps.

The other elements of the matrix are calculated by finding the maximum value among the following three values:

$$sim(s[i,j]) = \begin{cases} sim(s[i,j-1]) + gp, \\ sim(s[i-1,j-1]) + ss, \\ sim(s[i-1,j]) + gp \end{cases} ----(1)$$

Here, *gp* is -1, and *ss* is 1 if the elements match and 0 otherwise. However, other general values can be used instead. Following this recurrence equation, the matrix is filled from top left to bottom right with entry $[i,j]$ requiring the entries $[i,j-1]$, $[i-1,j-1]$ and $[i-1,j]$. Notice that $SM[i,j]$ corresponds to the best score of the subsequences $x_1, x_2, ..., x_i$ and $y_1, y_2, ..., y_j$. Since global alignment takes into account the entire sequences, the final score will always be found in the bottom right hand corner of the matrix. In our example, the final score 4 gives us measures of how similar the two sequences are.

Parallel sequence alignment programs based on two different approaches OpenMP and MPI are developed in our research to compare the performance of both the models on our underlined architecture (section 4.1 and 4.2). The two programs differ on how the memory architecture is used during implementation. Both programs are written using C language.

IV. Approach and parallelization methodology

4.1 OpenMP Implementation for sequence alignment

In the shared memory implementation, based on the size of the tile (explained in section 4.2) provided as a parameter by the programmer, the original long sequence is divided into subsequences. One of these subsequences is kept by the main core and the rest of the subsequences are sent to $(n-1)$ cores to execute in parallel. Here number of working cores corresponds to the number of threads which is given as parameter. All these working cores (including main core) will execute in parallel. Each core operate on different set of data for construction of similarity matrix of Figure 3 [11]. After doing the subsequence alignment, all of these $(n-1)$ cores will send the alignment results to the main core. The main core will combine the subsequence alignment, sent by $(n-1)$ cores, to compute the final alignment score.

Our code has been tested on a twelve core workstation with 2.66GHz processor with 8 GB cache system. The architecture we used for our shared memory implementation is as shown in Fig 1. On this architecture OpenMP model Fig. 4 generates threads for parallel execution of the code. The Fig 4 shows that, out of the n threads (n given as parameter), one master thread generates the $(n-1)$ slave threads and each slave thread will run on each core simultaneously. Once the execution has finishes the result was given back to the master. The thread then sets its flag to inform master to issue next data for execution. Each thread executes data from each tile.

4.2 MPI Implementation for sequence alignment

For very long sequences, the sequence will be first cut into several sets of subsequences, and each of these subsequences will be aligned by each MPI processes executing on each cores parallely [12].

Here number of working cores corresponds to number of processes. For the cases when sequences are extremely long and cannot be aligned even with OpenMP due to limitation of memory, they are divided into several shorter pieces according to tile size. The number of iterations (number of tiles) is

$$\frac{Sequence\ Size}{Tile\ Size} ----(2)$$

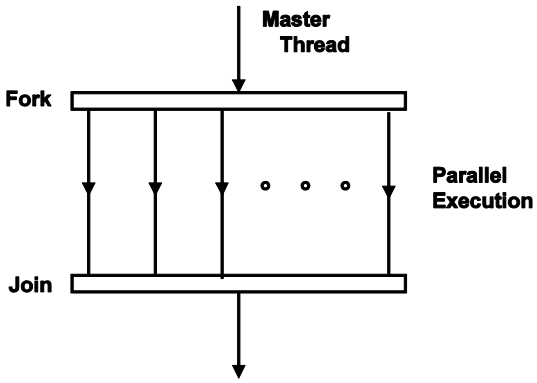


Fig.4. OpenMP – Programming Model

Our input is a matrix with two sequences, one along x axis and other along y axis. The tile size which is also two dimensional (along x and y axis), divides the similarity matrix of Fig. 3. Therefore total number of tiled iterations which we call here as tileXCount and tileYCount will be calculated according to equation 2.

For example, if sequence size is 1 MB and tile size is 16384x16384 B then the number of iterations tileXCount (sequence along x axis) becomes $\frac{1048576}{16384} = 64$. So $M = 64$. (M of Fig. 6) Similar calculation will go for tileYCount (sequence along y axis) depending upon the sequence size and tile size along y axis. That will be the value of N of Fig.6.

The general MPI Model is shown in Fig. 5. In our architecture we have such twelve cores per node.

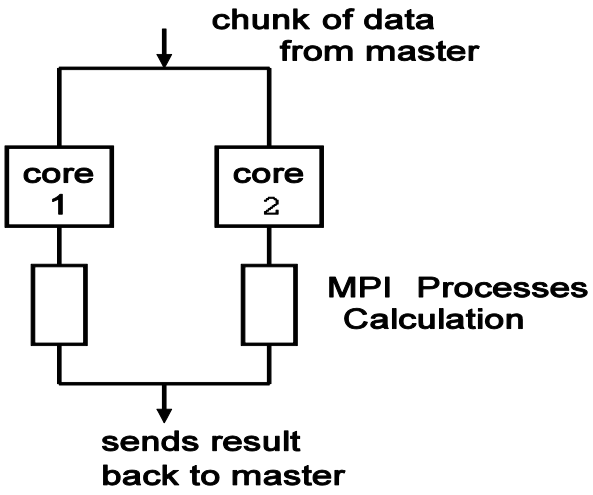


Fig. 5. MPI – Programming Model

As shown in Figure 6 the computation goes in wavefront manner, i.e once the first row and first column, i.e tile(0,0) has been filled with all the computed entries, calculated according to equation 1, the next tile to be calculated is south-east. Its value is dependent upon its immediate left column, top row and top-left element. For long sequence files, the size of the matrix also grows very large and after some point it could not get fit into memory. Hence to handle such long size files, we have used chunks of data (tiled data) for computation as show in the Fig. 6.

Here $\lceil N/T \rceil$ = tileYCount and $\lceil M/T \rceil$ = tileXCount are the number of tiles column-wise and row-wise respectively. Where N and M are the sequence size along y axis and x axis respectively and T is the tile size in bytes. When the execution starts, first tile (0,0) will get executed.

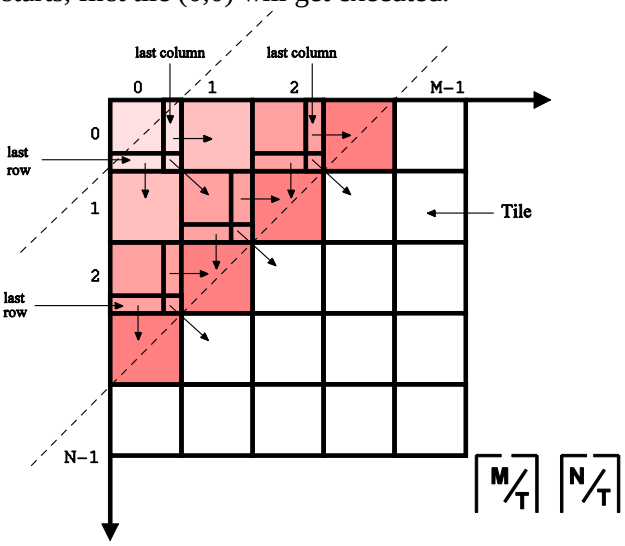


Fig.6. Execution of tiles in wavefront manner

The value will get stored in right column and bottom row of the corresponding tile. In the next run tiles $(0,1)$ and $(1,0)$ will be executed parallel. They will also store their respective computed values in right column and bottom row. Then in the next step, tiles $(0,2)$, $(1,1)$ and $(2,0)$ will get executed simultaneously. Thus the number of steps for execution of tiled iteration will be $M+N-1$, which is the number of diagonals as shown by the dotted lines in the Fig. 6. For example, for a matrix of size 4x4, number of actual steps

(diagonal rows) to be calculated in tiled iteration space would be (4+4-1=7).

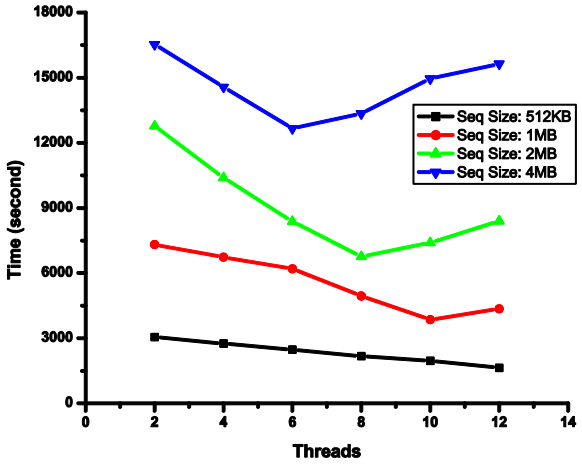
The two different parallel implementations differ on how the memory architecture is used during implementation. One program is called the thread based shared memory, in which computation is started for the full length sequences at the beginning, and after the sequences are cut into sub-sequences, all the sub alignments will follow the single, uniform sequence. The MPI implementation is based on distributed memory architecture, in which sequences will be cut first and each of the sub-sequences will build their own computation for their individual alignments. Both programs are written using C++ language.

V. Results

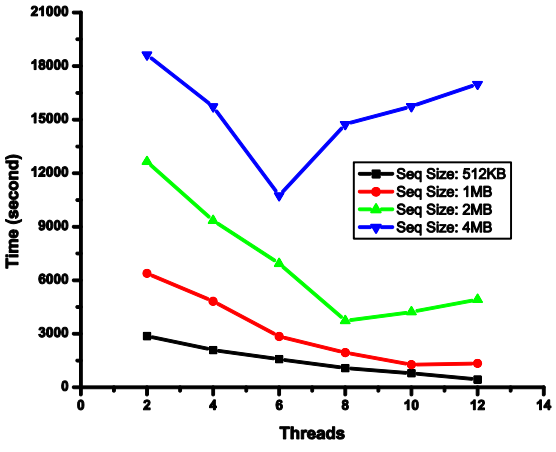
The experiments described in the paper were carried out on two Linux based workstation of Intel Xeon 5650 processor with 2.66GHz clock speed of 12 cores each. We have done the analysis on execution time and speedup with varying number of threads and chunk sizes for OpenMP. We have also done the analysis on execution time and speedup with varying number of processes and tile size for MPI as parameters. The corresponding execution time and speedup are reported in Fig. 7 and Fig. 8. The results were obtained for configuration such that for OpenMP implementation the number of threads generated will correspond to number of working cores and for MPI implementation number of processes corresponds to number of working cores.

The major advantage of MPI programming is program speedup in terms of time, because each process processes a different piece of the same job simultaneously and independently. However, this is obviously the case for the MPI program, but not quite true for the shared memory system, which does not gain any speed improvement after some point. The reason is that in shared memory implementation based on fork and join model, no matter how many processors are used, every time a single (master) thread for the full length sequences is built at the beginning for the processors sub-alignments, and it appears to be the most time-consuming part of all alignment procedures.

For testing the implementation over various length Sequences, four sample input files are chosen from a publicly available GenBank database [13] which have benchmark alignments. The first file consists of 512 KB character long, the second file consists of 1MB character long, the third file consists of 2 MB characters long and fourth is 4MB characters long. Table 1 and 2 depicted our analysis of both paradigms; its corresponding graphical representation is shown in Fig. 7, 8 and Fig. 10, 11. It shows that, for MPI implementation we obtained a very good speedup for large sequences compared to OpenMP implementation. We also obtained sufficiently satisfactory speed up for small and medium sequences. The results of the simulations are shown in the following figures:

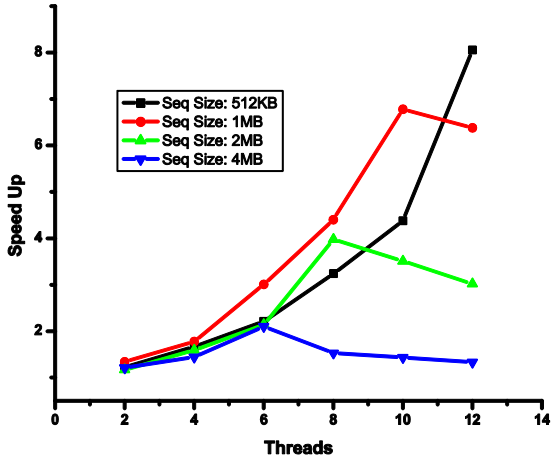


(a) For Tile Size 8192x8192

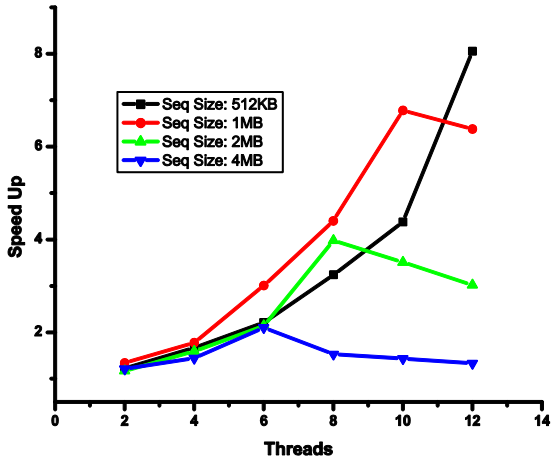


(b) For Tile Size 16384x16384

Fig. 7. Time analysis for OpenMP implementation on input files of various size.



(a) For Tile Size 8192x8192



(c) For Tile Size 16384x16384

Fig.8. Speedup analysis for OpenMP implementation on input files of various size.

From table 1, for both tile sizes, it is observed from the highlighted entries that as the sequence size increases the optimum performance obtained with respect to time and speedup improves with the decreasing number of threads (working cores). For example, as shown in table 1, for sequence size 512KB for both tile sizes the optimum performance obtained on 12 threads. For sequence size 1MB, the optimum performance is on 10 threads. Similarly for sequence size 2MB and 4MB the optimum performance is on 8 and 6 threads respectively. This can also observed from the curves of OpenMP graphs of Fig.7 and 8 for time and speedup respectively.

This is probably because of the following reason. The total number of processors is two with each containing six cores. The layout is illustrated in Fig. 9, reproduced and simplified from [13]. When running tests with both tile sizes on, for example, 4 cores, we must use a minimum of

$$\frac{8192x8192xsizeofBytes}{4(cores)} = 128Mb$$ of data for each core, when 8192x8192 tile size is used

and

$$\frac{16384x16384xsizeofBytes}{4(cores)} = 512MB$$ of data for each core, when 16384x16384 tile size is used.

How the data is exactly located on each core is not known, but assume that the tile size matrix is located as depicted in Fig. 9, where the shaded memory block contains the data from the master working core. Now, when the algorithm uses 12 OpenMP threads, all the cores will access the memory block in all iterations of the solver. For the cores to retrieve this memory, the cores farthest away from the data might have to idle several cycles to get the data to be processed. This will happen even if the data is not distributed as in Fig 6, because of the nature of OpenMP.

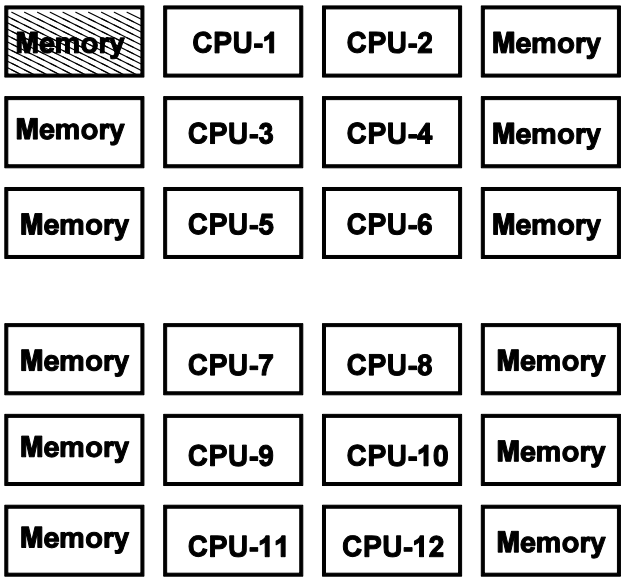


Fig.9. Data resides in the shaded memory block, longest away from CPU11 and CPU12.

The total overhead, O_t, for an OpenMP loop on processors executed by a number of threads can be expressed as

$$O_t = T_{comm} + T_{threads} \text{ ---------- (3)}$$

Where T_{comm} is the time required for communication and $T_{threads}$ is the overhead from spawning, destroying and switching between active threads. For a for-loop in OpenMP, a loop of i^{th} iterations is divided into chunk size of k, so that the number of messages sent is

$$\frac{i}{k} \text{ ---------- (4)}$$

and the size, m of the messages are

$$m = k * sizeof(datatype) \text{----------(5)}$$

The cost of communication of k bytes is known to be

$$T_{comm} = T_m + \beta m \text{ ----------(6)}$$

Where T_m is the latency of the respective cache level or memory where the data is located and β is the bandwidth between the processors. Summarizing these formulas, we get a general expression for overhead of communication in an OpenMP for-loop,

$$T_{comm} = \frac{i}{k}(T_m + \beta * k * sizeof(datatype)) \text{ ------(7)}$$

There are three variables in this equation we should look into to provide a better understanding of the estimate, the bandwidth β, cache-latency T_m and chunk size k.

- The bandwidth could well be related to how many processors share the communication bus, so this variable could be decreasing as the number of processors increases.
- T_m can be found with the same method as β, only with NULL-size messages. Depending on how many processors that participate in the computation, the length of the data bus and how many processors using the data bus at the same time will influence the latency. As more cores participate in the for loop, it is likely that we end up with a decreasing average latency.
- The tile size k can be specified statically with the OpenMP keyword schedule (static, tile size). This is however not recommended, since a normal programmer seldom knows the optimal tile size. By not specifying the tile size, it is set to be decided in runtime and therefore difficult to know. Finding the optimal tile size can be done empirically by comparing different tile sizes to the default schedule (runtime).

Further the speedup graphs indicate that we get little speedup in our NW algorithm with this strategy. This could be because of border value exchanges between tiles during the calculation, or that the latency, T_m, is less significant than the bandwidth, β, in Equation 7.

If a thread migrates to another CPU for some reason, this migration takes extra time and must be included in the overhead calculation. To test if this was a significant source of overhead, we used the bindprocess() function-call to bind the OpenMP threads to a given CPU. However, this did not give any notable performance improvements. Since the bindprocess() function did not give any improvement, it might be that threads already are bound to a CPU in IBM's OpenMP-implementation, or that threads did not migrate before we used the bindprocess() call.

In case of pure MPI, the program is designed to have multiple instances of itself running in parallel on different processors. Each process has its own local memory and communicates with other instances by sending messages to them. To be able to distinguish between those processes, a unique id is assigned to each of them. Such message passing implementation is more difficult to program than multithreaded programs as all the communication between nodes has to be done explicitly.

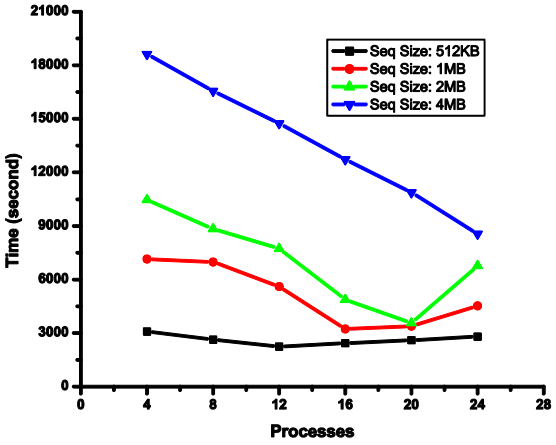
A lot of care has to be taken to maximize the communication throughput by grouping communication tasks where it is possible. The overhead created by message passing can be outweighed by the advantage of explicitly expressing locality of a process. Making the communication between processes explicit also reduces problems with *false sharing*.

In our MPI implementation, the complete data which is to be computed is distributed to the node where the cores will execute it in parallel. These data is located in a memory bank close to the core on each node. So the idle time when waiting for memory is avoided because of the close proximity to the memory[13]. With larger sequence size, the MPI processes achieve greater performance because all of the data resides in the level 3 cache which can also be inferred from figure 7.

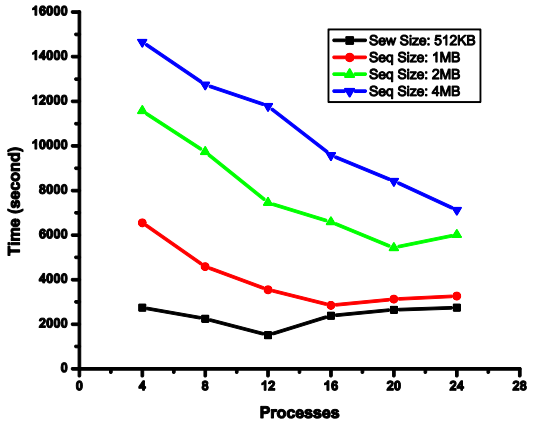
With several cores sharing data on each node, a significant source of communication overhead between the cores is reduced. Hence with pure MPI model, for large size file, this technique gives improved performance. Because if the sequence size is small the overall time required to distribute data to all nodes will be comparatively more than the time required for this distribution for large sized data.

As discussed above, our results also show the same observation for MPI model. Table 2 shows from the highlighted entries that as the sequence size increases the optimum performance obtained with the increasing number of processes and tile size. For example, when sequence size is 512KB the optimum performance obtained on 12 processes. For sequence size 1MB the optimum performance was on 16 processes. Similarly for sequence size 2MB and 4MB the optimum performance is on 20 and 24 processes respectively. This can also observed from the curves of MPI graphs of Fig.10 and 11 for time and speedup respectively.

This is because for big tile size, more data has been send to the slave process and hence communication between the processes gets reduced, which would be otherwise more for smaller tile size. Similarly as number of processes increases more data gets computed simultaneously. Hence for bigger file size MPI gives improved performance.

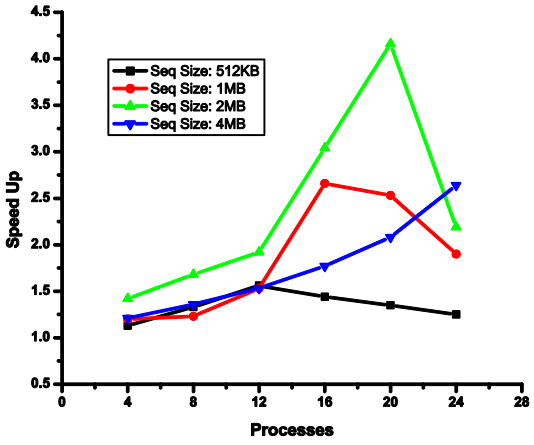


(a) for tile Size (8192x8192)

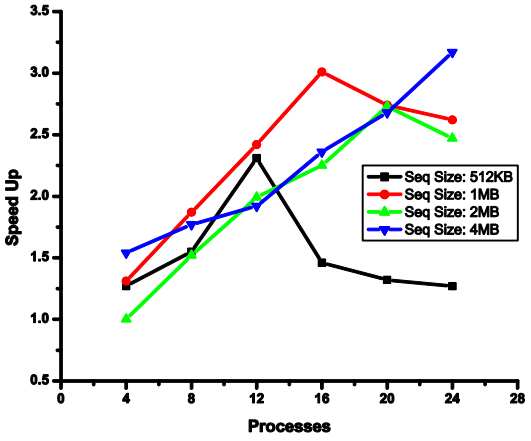


(b) for tile Size (16384x16384)

Fig.10. Time analysis for MPI implementation on various sequence size input files



(a) for tile Size (8192x8192)



(b) for tile Size (16384x16384)

Figure 11. Speedup analysis for MPI implementation on various sequence size input files

VI. Conclusion

In this paper we have

1. Parallelized and optimized an existing Needleman-Wunsch algorithm with OpenMP and MPI and studied the performance with respect to time and speedup for large sequence size using appropriate tile size.
2. Did a study of OpenMP and MPI on NW algorithm and conclude that depending upon sequence size, tile sizes and number of threads and processes as parameters on our architecture, in some cases OpenMP shows better performance while in all the other cases MPI provides the optimum performance.
3. Proposed a general model for estimating overhead in for-loops in OpenMP.

To parallelize the Needleman-Wunsch algorithm codes we have used both OpenMP and MPI to explore the possibilities of maximum performance. MPI was used globally to exchange border-values and particles while OpenMP was used to parallelize compute-intensive for-loops. While OpenMP is the easiest way to parallelize loops, our results showed that overhead from communication between the processors makes it a low-performing API compared to MPI where data is communicated explicitly. We suggested a general expression to model the overhead of communication in OpenMP loops based on number of loop-iterations, chunk size, latency and bandwidth. We also improved the way memory was allocated and freed dynamically during parallel execution in order to handle large size sequence file and tried to improve the scalability of the parallel code. Our results show that we have achieved speedup of 8.06 on 512KB sequence size with 12 threads for OpenMP and 4.16 on 2MB sequence size with 20 processes for MPI implementation.

References:

[1] Hsien-Yu, L., Meng-Lai, Y., & Yi, C. (2004). A parallel implementation of the Smith-Waterman algorithm for massive sequences searching. *Engineering in Medicine and Biology Society, 2004. IEMBS apos;04. 26th Annual International Conference of the IEEE*, (pp. 2817-2820). San Francisco, CA, USA.

[2] Murakami, M. M., Maria, E., Walter, M. T., & Martins, W. S. (2003). Parallel Implementation of the Smith-Waterman Algorithm for Large Scale Database Search.*The 1st International Conference on Bioinformatics and Computational Biology - ICoBiCoBi, 2003.*RibeirãoPreto.

[3] Gotoh, O. (1982). An improved algorithm for matching biological sequences.*Journal of Molecular Biology , 162*, 705-708.

[4] Benson, D. A., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., & Wheeler, D. L. (2008). GenBank.*Nucleic Acids Research* , 25-30.

[5] Meng, X., &Chaudhary, V. (2005). Exploiting Multi-level Parallelism for Homology Search using General Purpose Processors.*Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops (ICPADS'05) -Volume 02* (pp. 331-335). Washington, DC, USA: IEEE Computer Society.

[6] Sanchez, F., Salami, E., Ramirez, A., & Valero, M. (2005). Parallel processing in biological sequence comparison using general purpose processors.*Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, (pp. 99-108).

[7] Saul B. Needleman and Christian D. Wunsch."A general method applicable to the search for similarities in the amino acid sequence of two sequences," *Journal of Molecular Biology*, pp. 443-453, 1970

[8] The Message Passing Interface (MPI) Standard - http://wwwunix. mcs.anl.gov/mpi/ and http://www.mpi-forum.org
[9] Anne C. Elster and David L. Presberg, "Setting Standards For Parallel Computing: The High Performance Fortran and Message Passing Interface Efforts ", May 1993, Theory Center SMART NODE Newsletter, Vol. 5, No.3 . http://www.idi.ntnu.no/ elster
[10] SCALI higher performance computing, http://www.scali.com/
[11] Viktor K. Decyk and Charles D. Norton, "UCLA Parallel PIC Framework,", January 2006, University of Los Angeles, USA.
[12] Ted Retzlaff and John G. Shaw , "Simulation of Multi-component Charged Particle Systems", 2002, Wilson Center for Research and Techology , Xerox Corporation, USA.
[13] GenBank, http://www.ncbi.nlm.nih.gov
[14] Alejandro Duran, Marc Gonzàles, and JulitaCorbalán.Automatic Thread Distributionfor Nested Parallelism inOpenMP. In *19th ACM International Conference onSupercomputing*, pages 121–130, Cambridge, MA, USA, June 2005.

Table 1. Performance Evaluation of OpenMP on both tile sizes.

	Sequence Size: 512KB				Sequence Size: 1 MB				Sequence Size: 2MB				Sequence Size: 4MB			
→ Tile Size	8192x8192		16384x16384		8192x8192		16384x16384		8192x8192		16384x16384		8192x8192		16384x16384	
Threads	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up
1	3498		3498		8573		8573		14,834		14,834		22,583		22,583	
2	3056	1.14	2875	1.22	7321	1.17	6385	1.34	12,765	1.16	12,643	1.17	16,538	1.37	18,638	1.21
4	2754	1.27	2095	1.67	6739	1.27	4823	1.78	10,384	1.43	9346	1.59	14,574	1.55	15,735	1.44
6	2473	1.41	1584	2.21	6194	1.38	2847	3.01	8376	1.77	6934	2.14	**12,654**	**1.78**	**10,749**	**2.10**
8	2176	1.61	1080	3.24	4945	1.73	1947	4.40	**6754**	**2.20**	**3727**	**3.98**	13,354	1.69	14,745	1.53
10	1963	1.78	798	4.38	**3856**	**2.22**	**1265**	**6.78**	7394	2.01	4226	3.51	14,959	1.51	15,746	1.43
12	**1642**	**2.13**	**434**	**8.06**	4356	1.97	1343	6.38	8394	1.77	4915	3.02	15,637	1.44	16,984	1.33

Table 2. Performance Evaluation of MPI on both tile sizes.

	Sequence Size: 512KB				Sequence Size: 1 MB				Sequence Size: 2MB				Sequence Size: 4MB			
→ Tile Size	8192x8192		16384x16384		8192x8192		16384x16384		8192x8192		16384x16384		8192x8192		16384x16384	
Processes	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up	Time (sec)	Speed up
1	3498		3498		8573		8573		14,834		14,834		22,583		22,583	
4	3087	1.13	2754	1.27	7148	1.20	6548	1.31	10,463	1.42	11,568	1	18,623	1.21	14,658	1.54
8	2638	1.33	2254	1.55	6983	1.23	4587	1.87	8835	1.68	9735	1.52	16,563	1.36	12,745	1.77
12	**2246**	**1.56**	**1514**	**2.31**	5598	1.53	3547	2.42	7723	1.92	7456	1.99	14,749	1.53	11,789	1.92
16	2429	1.44	2389	1.46	**3224**	**2.66**	**2845**	**3.01**	4873	3.04	6587	2.25	12723	1.77	9576	2.36
20	2598	1.35	2647	1.32	3387	2.53	3125	2.74	**3565**	**4.16**	**5426**	**2.73**	10873	2.08	8423	2.68
24	2798	1.25	2748	1.27	4523	1.90	3268	2.62	6761	2.19	6014	2.47	**8564**	**2.64**	**7124**	**3.17**