Load Balancing for Parallel Motif Discoveries

Angkul Kongmunvattana
School of Computer Science
Columbus State University
4225 University Avenue, Columbus, GA 31907 USA

ABSTRACT
The problem of motif discovery has been studied extensively over the last few decades. Many sequential and parallel algorithms have been proposed and studied. A significant runtime is still required for several challenging instances of the motif search problem. This paper studies parameter spaces to find an optimal point for load balancing between the master and the worker processes, which are collaboratively and concurrently searching for motifs. Extensive experiments have been carried out on the state-of-the-art TACC Stampede System. The results demonstrated that a workload from the parallel motif discovery problem is best divided between the master and worker processes by having the master process worked on the first $\frac{(l-4)}{2}$ nucleotides in the DNA sequences, where l is the total length of the input DNA sequences, before passing the remaining work to the worker process. In addition, the results also shown that the latency tolerance techniques used in the implementation of this work is effective because of the almost linear speedup obtained.

General Terms
Bioinformatics, Computational Genomics, DNA Sequence, Parallel Computing

Keywords
Latency Tolerance, Load Balancing, Message Passing, Motif Discovery

1. INTRODUCTION
In computational genomics, motif is a recurring pattern of nucleotides in DNA that has, or presumed to have, a biological significance [1]. The planted motif search (PMS) problem was introduced to aid the innovation of algorithms for discovering motifs in a set of DNA sequences [2]. The PMS problem can be defined as follows. Given a set of n DNA sequences (each with length m), find all k motifs (each with length l where $l \leq m$ and each with at most d positions of mismatch) that appear as a subset in all of the n DNA sequences. Thus, the PMS problem is also known as the *(l,d)*-motif search problem. For example, given a set of three DNA sequences ($n=3$), each with length $m=10$ (i.e., ACTGACGCAG, TCACAACGGG, and GAGTCCAGTT), there are four motifs of length $l=4$ with at most $d=1$ position of mismatch (i.e., ACAG, CAGA, CCCA, and TCAG).

While numerous algorithms for the PMS problem have been proposed and studied [3], there are several challenging instances of this problem that still cannot be solved (or required significant amount of time to solve). For example, (26,11)-motif search problem still needs more than twelve hours of execution time on 48 CPU cores and (28,12) requires more than 27 hours to complete on the same number of cores using the most efficient algorithm to date [4]. Thus, an improvement over the existing algorithms is still essential. While past research works have focused on improving the algorithms [4], [5], [6], [7], [8], [9], [10], and on designing hardware accelerators [11], [12], [13], [14], [15], for motif discovery, none of them focused on the load balancing issues between the master and the worker processes in the parallel execution. To the best of our knowledge this is the first paper that explored and evaluated all parameters of the parallel motif discovery problem to find the right balance for workload distributions.

In this paper, a simple parallel motif discovery program was designed and implemented for exploring and evaluating the influences of each parameter in the motif search problem. These parameters include the length of motifs, the number of input DNA sequences, the allowable Hamming distance, the pivot for partitioning a workload between the master and the worker processes, and finally, the number of processor cores. A comprehensive set of experiments was carried out on Stampede System at the Texas Advanced Computing Center. The results demonstrated that the proposed simple parallel motif discovery program yields linear speedup as the number of processor cores increases. Furthermore, the results also indicated that a workload should be divided almost evenly between the master and the worker processes.

The rest of this paper is organized as follows. A concise summary on the PMS problem is provided in Section 2. An overview on the implementation of parallel motif discovery algorithm is described in Section 3. The experimental setup and results are discussed in Section 4. The findings are summarized in Section 5.

2. PLANTED MOTIF SEARCH
Planted motif search (PMS) problem was formulated by Pevzner and Sze [2]. The PMS problem has been influential in the development of several algorithms for finding a recurring pattern in DNA sequences and has since been applied to other biological sequences, such as RNA and protein sequences.

In PMS problem, $S = \{s_1, s_2, \ldots, s_n\}$ is defined as a set of DNA sequences. A de facto standard from prior work assumed twenty DNA sequences (i.e., $n=20$). Each DNA sequence (s_i) consists of four nucleobases called adenine (A), cytosine (C), guanine (G), and thymine (T) of length m, which is typically set to 600 (i.e, $m=600$). A recurring pattern of nucleobases with at most d mismatched positions found in all of these DNA sequences is called a *motif*. The length of nucleobases in a discovered motif is defined through an l parameter, where $0 \leq d \leq l \leq m$. The goal of PMS is finding all motifs in the given DNA sequences that satisfy both l and d parameters. This is a computational and data intensive task. A brute force method considered all 4^l possible candidates for motifs against $n(m-l)$ subsequences from the given set of DNA sequences. The search space can be drastically reduced from 4^l to by simply taking into account the fact that all motifs must have at most d

mismatched positions from the given DNA sequences. Specifically, instead of generating all 4^l possible candidates from scratch, we can generate candidate motifs from one of the DNA sequences, limiting the search space to $\binom{l}{d}3(m-l)$. This idea formed the basis of the design and implementation of our simple parallel motif discovery program, which is presented in the next section.

3. PARALLEL MOTIF DISCOVERIES

The main goal of the algorithms is finding all motifs that are within a given Hamming distance (d) from all of the input DNA sequences. The algorithm uses the master process to search the space of DNA sequences within a Hamming distance d from the first input DNA sequence by going through all permutations from the least significant nucleotide position (i.e., the rightmost nucleotide) to the $(x\text{-}1)^{th}$ position, where x is a constant that is smaller than the length of DNA sequences (l). The worker process then searches the remaining space of DNA sequences within a Hamming distance d from the first input DNA sequence by going through all permutations from the x^{th} position to the $(l\text{-}1)^{th}$ position (i.e. the leftmost nucleotide).

Computational concurrency in the master and the worker processes is achieved through an immediate dispatch of work on each permutation from the master process to one of the worker processes. The master process dispatches two rounds of work to all worker processes (if there are enough work to go around) using non-blocking communication and moving forward to perform its own portion of the work. Each worker process started working on its part of the work immediately upon receiving the first assigned work from the master process and also using non-blocking communication to receive an additional work from the master process. When a worker process finished each of its assigned works, the results are sent back to the master process before continuing with the next assigned work, if any. Upon receiving the results, the master process dispatches the next assignment to the worker process that has sent in the results. Dynamically distributed the workload across the worker processes. This routine is repeated until there are no work left and termination messages are sent from the master process to the worker processes.

The non-blocking communication was used to send assigned work from the master process to the worker processes. This non-blocking send operation is overlapped with the work that the master process itself has to perform, which includes checking a potential result against the remaining input DNA sequences (i.e., all input DNA sequences except the first one) as well as generating the next potential result. Thus, overlapping of computation and communication is achieved in the master process.

On the side of worker processes, the non-blocking communication was also deployed for receiving assigned work (after the very first assigned work was received through a blocking receive call). This non-blocking receive operation is overlapped with the work that each worker process has to perform (based on the previous assigned work received), which includes generating and checking all potential results against the remaining input DNA sequences as well as sending the results back to the master. Thus, overlapping of computation and communication is also achieved in the worker process.

The message tag in the MPI calls was used for indicating the type of messages between the master and the worker processes. In particular, message tag with value 0 is used for indicating that there are no work left to assign (i.e., termination message), 1 for sending assigned work from the master process to the worker processes, 2 for sending the size of results from the worker processes to the master process, and finally, 3 for sending the results from the worker processes to the master process.

4. RESULTS AND DISCUSSIONS

The experiments were carried out on two 16-core nodes of the TACC Stampede System. Each node has two 8-core Xeon E5 processors with 32GB of DDR3 RAM (2GB per core), running the CentOS version 6.3 with the 2.6.32 x86_64 Linux kernel. These nodes are connected via InfiniBand (56 Gbps). Each of the Xeon E5 core is running at 2.7GHz. The code was developed in C/C++ and compiled with MPICXX for MVAPICH2 version 1.9a2. All results came from an average of data collected from three independent runs.

The first set of experiments was carried out on 11 different input sets with varying numbers of input sequences (n), ranging from 5 to 15. The other parameters are fixed with *l=38, d=10, x=19, and p=32*, where l is the length of each input sequence, d is the maximum mismatched positions, x is the length of input sequence covered by the master process before passing on the remaining work to the worker process, and p is the number of processor cores. The runtime and speedup plots from this first set of experiments are presented in Figures 1 and 2, respectively. These results indicated that the runtime increases as the problem size grows. It also showed that when the problem size is sufficiently large ($n=15$) the performance is close to linear speedup (i.e., 29.41 times with 32 processor cores deployed).

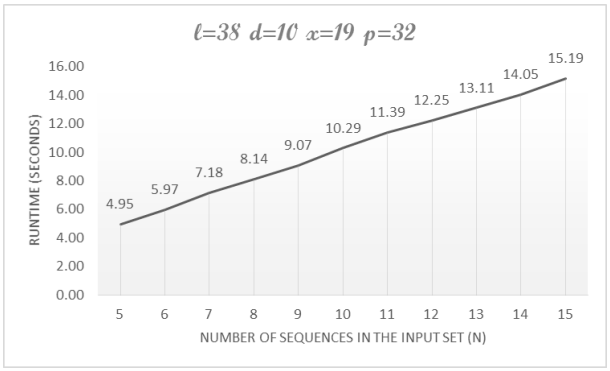

Fig 1: Runtime plot under different input *n* sizes

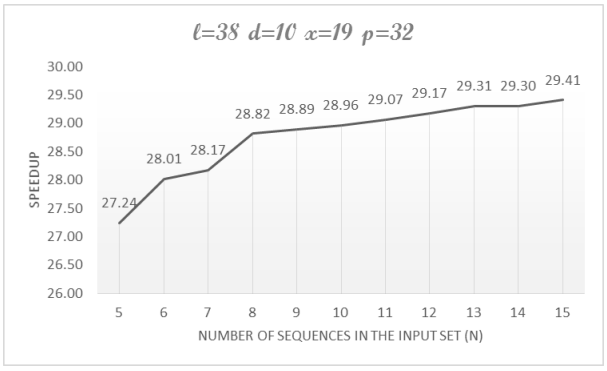

Fig 2: Speedup plot under different input *n* sizes

The second set of experiments deals with different length (l) of the input DNA sequences, ranging from 38 to 48. The other parameters are fixed (i.e., *n=5, d=10, x=19*, and *p=32*). The runtime and speedup plots from this set of experiments are presented in Figures 3 and 4, respectively. These results

indicated that the runtime increases as the problem size grows. It also showed that the peak performance is obtained when the problem size (*l*) is slightly larger than a doubling of the fixed parameter *x*. Specifically, the performance is close to linear speedup at 28.77 times with 32 processor cores deployed when *l=41* and the fixed parameter is *x=19* in this case.

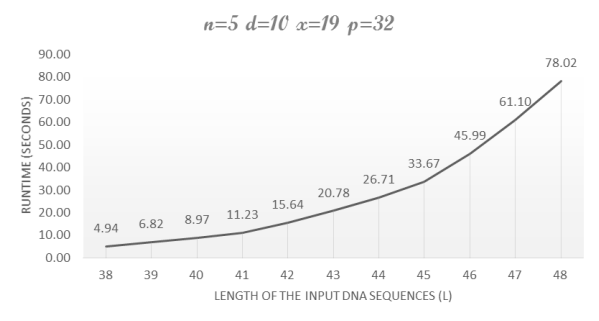

Fig 3: Runtime plot under different input *l* sizes

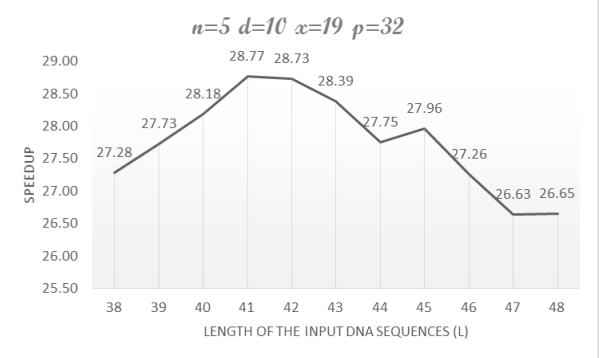

Fig 4: Speedup plot under different input *l* sizes

The third set of experiments deals with different Hamming distances (*d*) in the given input, ranging from 8 to 12. The other parameters are fixed (i.e., *n=5*, *l=38*, *x=19*, and *p=32*). The runtime and speedup plots from this set of experiments are presented in Figures 5 and 6, respectively. The results came from an average of data collected from three independent runs. These results indicated that the runtime increases drastically as the search space grows due to an increase of Hamming distance. It also showed that when the problem size is sufficiently large (*d=12*) the performance is close to linear speedup (i.e., 30.26 times with 32 processor cores deployed).

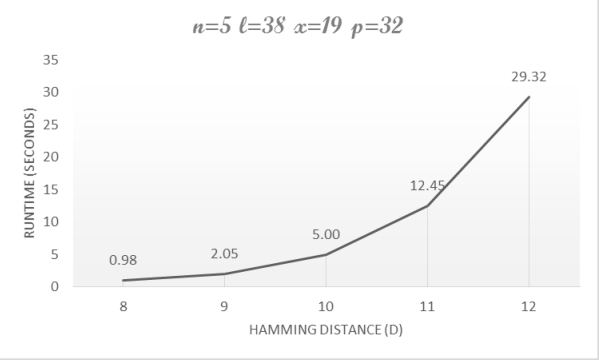

Fig 5: Runtime plot under different Hamming distance *d*

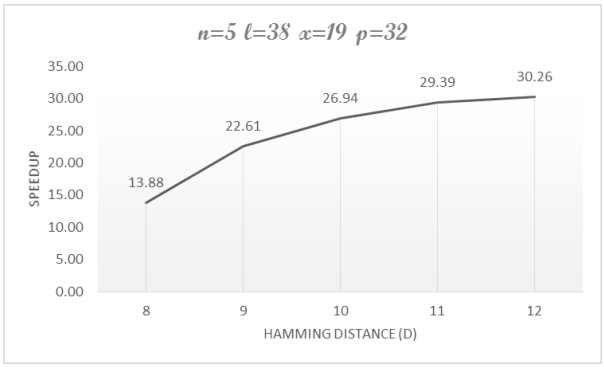

Fig 6: Speedup plot under different Hamming distance *d*

The next set of experiments deals with different values of parameter *x*, ranging from 1 to 24. The other parameters are fixed (i.e., *n=5*, *l=38*, *d=12*, and *p=32*). The runtime and speedup plots from this set of experiments are presented in Figures 11 and 12, respectively. The results came from an average of data collected from three independent runs. These results indicated that the runtime decreases drastically as the value of parameter *x* increases. This is because of a balance in parallel workload distribution between the master and the worker processes. The workload for the master process increases as the value of parameter *x* increases. The point of balance (also observed earlier in the results from the second set of experiments) is around where the value of *l* is slightly higher than doubling of the parameter *x*'s value. The speedup plot in Figure 8 showed that when *x=17* the performance is closest to linear speedup (i.e., 29.44 times with 32 processors deployed) for this input set where *l=38*.

The fifth set of experiments deals with different number of processors deployed for parallel execution, ranging from *p=1* to *p=32*. The other parameters are fixed (i.e., *n=5*, *l=38*, *d=12*, and *x=19*). The runtime and speedup plots from this set of experiments are presented in Figures 13 and 14, respectively. The results came from an average of data collected from three independent runs. These results indicated that the runtime decreases drastically at the beginning when the number of processor cores increases from one to four cores. The reduction in runtime is less significant as the number of cores getting closer to 32. However, the speedup plot presented a different story as the speedup continues to increase consistently and gradually as the number of processor cores increases, peaking at 29.7 times with 32 processor cores deployed.

Four additional sets of experiments were carried out to find a value of parameter *x* that yields the highest performance. The experiments were carried out with various ranges of values for parameter *x* using 4 different input sets with different values of *l* and *d*.

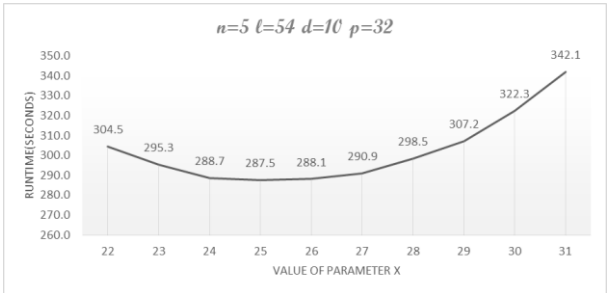

Fig 7: Runtime plot under different value of parameter *x*

The first set of experiments used an input set with the following parameters *n=5*, *l=54*, and *d=10*. The program was executed on 32 processor cores (i.e., *p=32*). The value of parameter *x* was ranging from 22 to 31. The runtime plot from this set of experiments is shown in Figure 7. It showed that the highest performance is obtained when *x=25*.

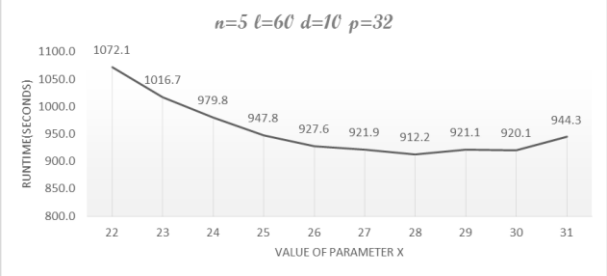

Fig 8: Runtime plot under different value of parameter *x*

The second set of experiments used an input set with the following parameters *n=5*, *l=60*, and *d=10*. The program was also executed on 32 processors (i.e., *p=32*). The value of parameter *x* was ranging from 22 to 31. The runtime plot from this set of experiments is shown in Figure 8. It showed that the highest performance is obtained when *x=28*.

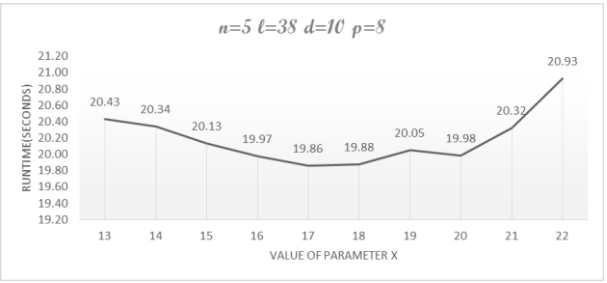

Fig 9: Runtime plot under different value of parameter *x*

The third set of experiments used an input set with the following parameters *n=5*, *l=38*, and *d=10*. The program was executed on 8 processors (i.e., *p=8*). The value of parameter *x* was ranging from 13 to 22. The runtime plot from this set of experiments is shown in Figure 9. It showed that the highest performance is obtained when *x=17* and *x=18*.

Finally, the last set of experiments used an input set with the following parameters *n=5*, *l=38*, and *d=12*. The program was executed on 8 processors (i.e., *p=8*). The value of parameter *x* was also ranging from 13 to 22. The runtime plot from this set of experiments is shown in Figure 10. It showed that the highest performance is obtained when *x=18*.

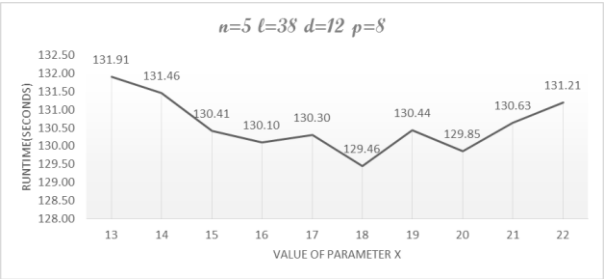

Fig 10: Runtime plot under different value of parameter *x*

Based on these experimental results, it seems the value of parameter *x* for the highest performance is approximately equal to $\frac{(l-4)}{2}$, where *l* is the length of input DNA sequences. This is perhaps due to the fact that this formula, when used to determine the value of *x*, divided the workload almost evenly between the master and all worker processes with a slightly less work for the master process since it is the center of communication processing and results accumulation.

5. CONCLUSION

The planted motif search problem is an essential framework for exploring and developing algorithms for finding repeated recurring patterns in DNA, RNA, and protein sequences. Parallel motif discovery enabled a harvesting of the computing power made available by the multicore platforms. A common master-worker model of workload distribution may lead to load imbalance. Research work in this paper demonstrated that non-blocking communication operations can be used to implement latency tolerance techniques by overlapping communication with computation. Furthermore, an extensive set of experimental studies also showed that load balancing can be achieved by dividing the workload almost evenly between the master and the worker processes.

There are several possible future works to further improve the efficiency of parallel motif discoveries. First, decentralization of task scheduler may help alleviating a bottleneck on the centralized master processor core in the current design and implementation. Tree-based structures such as binary tree and binomial spanning tree can be adopted for this purpose as shown in the implementation of a decentralized barrier synchronization operation [16].

Second, given that the runtime of parallel motif discovery program can span days, a checkpoint/restart support is beneficial since it reduces the cost and time for re-computation should a failure (or failures) occurred. Past research work has shown that checkpoint creation library such as BLCR can be deployed for this purpose and Pin-based tool can be developed to reduce its overhead further [17].

Finally, with an increasing number of processor cores in each computing node, a design and implementation of parallel motif discovery program utilizing both multithreading and message-passing operations should be explored to reduce the cost of memory consumption and the amount of data transfer between processes residing on different cores within the same computing node. The current approach only utilizes message-passing operations, and therefore, data transfer is invoked even when both the sender and receiver are locating on the same computing node.

6. ACKNOWLEDGMENTS

The author thanks Texas Advanced Computing Center's technical support staffs for providing a superb platform for this work. This research is supported in part by computing resources from the NSF XSEDE Startup Allocation Award.

7. REFERENCES

[1] P. D'haeseleer, "What are DNA sequence motifs?" Nature Biotechnology, vol. 24, pp. 423-425, 2006.

[2] P. A. Pevzner and S.-H. Sze, "Combinatorial approaches to finding subtle signals in DNA sequences," In Proceedings of the 8th Int'l Conference on Intelligent Systems for Molecular Biology, pp. 269-278, 2000.

[3] M. K. Das and H. K. Dai, "A survey of DNA motif finding algorithms," BMC Bioinformatics, vol. 8, 2007.

[4] M. Nicolae and S. Rajasekaran, "qPMS9: An efficient algorithm for quorum planted motif search," Nature Scientific Reports, vol. 5, 2015.

[5] S. Rajasekaran, S. Balla, and C.-H. Huang, "Exact algorithms for planted motif problems," Journal of Computational Biology, 12(8), pp. 1117-1128, 2005.

[6] J. Davila, S. Balla, and S. Rajasekaran, "Fast and practical algorithms for planted (l,d) motif search," IEEE Transactions on Computational Biology and Bioinformatics, vol. 4, no. 4., pp. 544-552, 2007.

[7] S. Rajasekaran and H. Dinh, "A speedup technique for (l,d)-motif finding algorithms," BMC Research Notes, 4:54, 2011.

[8] Q. Yu, H. Huo, Y. Zhang, and H. Guo, "PairMotif: A new pattern-driven algorithm for planted (l,d) DNA motif search," PLoS ONE, 7(10):e48442, 2012.

[9] S. Bandyopadhyay, S. Sahni, and S. Rajasekaran, "PMS6: A fast algorithm for motif discovery," In Proceedings of the 2nd IEEE International Conference on Computational Advances in Bio and Medical Sciences, pp. 1-6, 2012.

[10] M. Nicolae and S. Rajasekaran, "Efficient sequential and parallel algorithms for planted motif search," BMC Bioinformatics, 15:34, 2014.

[11] S. Sarkar, G. R. Kulkarni, P. P. Pande, and A. Kalyanaraman, "Network-on-chip hardware accelerators for biological sequence alignment," IEEE Transactions on Computers, vol. 59, no. 1, pp. 29-41, 2010.

[12] A. Boukerche, J. M. Correa, A. Melo, and R. P. Jacobi, "A hardware accelerator for the fast retrieval of DIALIGN biological sequence alignments in linear space," IEEE Transactions on Computers, vol. 59, no. 6, pp. 808-821, 2010.

[13] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," In Proceedings of the 20th IEEE Annual Int'l Symposium on Field-Programmable Custom Computing Machines, pp. 161-168, 2012.

[14] Y. Chen, B. Schmidt, and D. L. Maskell, "A hybrid short read mapping accelerator," BMC Bioinformatics, 14:67, 2013.

[15] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, "Reconfigurable acceleration of short read mapping," In Proceedings of the 21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 210-217, 2013.

[16] N.-F. Tzeng and A. Kongmunvattana, "Distributed shared memory systems with improved barrier synchronization and data transfer," In Proceedings of the 11th ACM International Conference on Supercomputing (ICS), pp. 148-155, 1997.

[17] J. Cornwell and A. Kongmunvattana, "Advanced I/O techniques for efficient and highly available process crash recovery protocols," GSTF Journal on Computing, vol. 1, no. 3, 2011.

8. APPENDIX

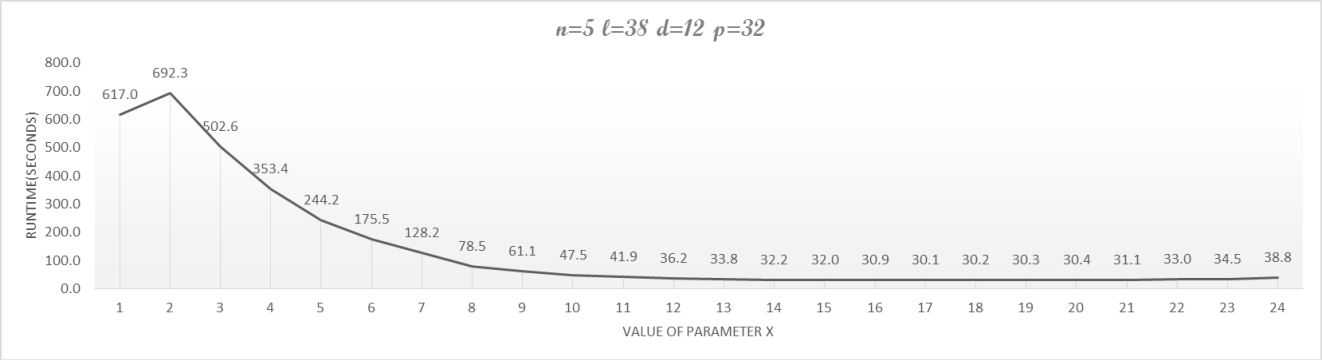

Fig 11: Runtime plot under different value of parameter *x*

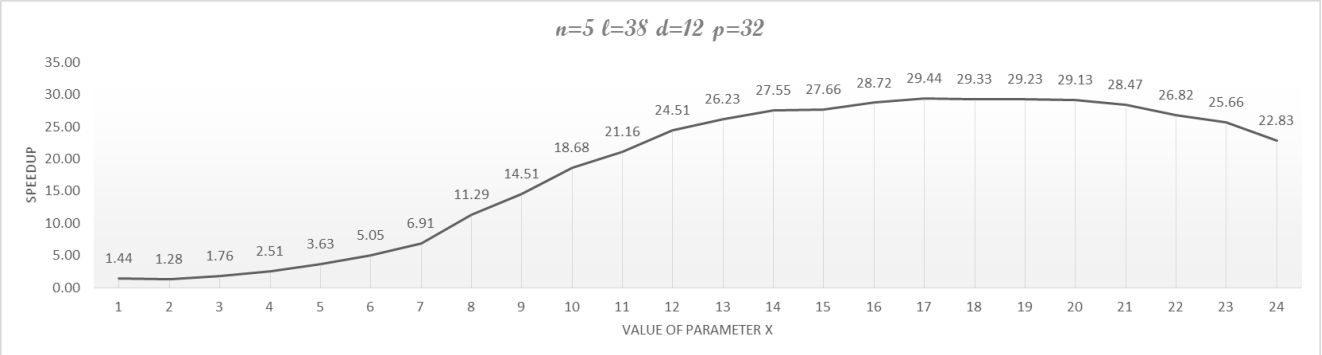

Fig 12: Speedup plot under different value of parameter *x*

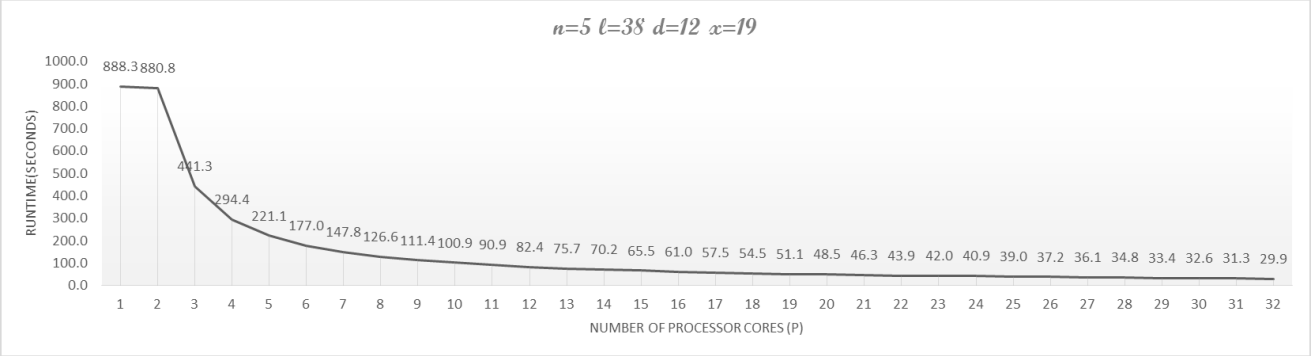

Fig 13: Runtime plot under different number of processor cores (*p*)

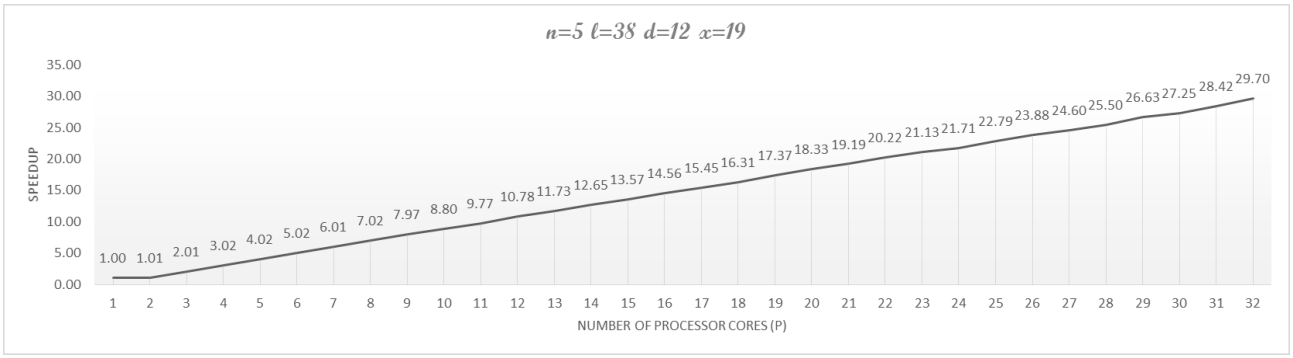

Fig 14: Speedup plot under different number of processor cores (*p*)