DOI: 10.2298/CSIS120529047E

Experimental and Theoretical Speedup Prediction of MPI-Based Applications

Alaa Elnashar and Sultan Aljahdali

Computer Science Department
College of Computers and Information Technology
P.O.B. 888, 21947 Taif, Saudi Arabia
{a.ismail, aljahdali}@tu.edu.sa

Abstract. Prediction of speedup obtained from parallelization plays an important role in converting serial applications into parallel ones. Several parameters affect the execution time of an application. In this paper we experimentally and theoretically study the effect of some of these parameters on the execution time of Message Passing Interface based applications

Keywords: Parallel programming, Message Passing Interface, Speedup

1. Introduction

Speedup is defined as the ratio of serial execution time to the parallel execution time [1], it is used to express how many times a parallel program works faster than its serial version used to solve the same problem.

Prediction of speedup gained from parallelization is an important issue in converting serial applications into parallel ones. Amdahl's Law [2], [3] is one way of predicting the maximum achievable speedup for a given program. The law assumes that a fraction of a program's execution time was infinitely parallelizable with no overhead, while the remaining fraction was totally serial [4]. The law treats problem size as a constant and hence the execution time decreases as number of processors increases. Gustafson law [5] is another one that predicts maximum achievable speedup. The two laws ignore the communication cost; they overestimate the speedup value [6].

Many conflicting parameters such as parallel overhead, hardware architecture, programming paradigm, programming style may negatively affect the execution time of a parallel program making its execution time larger than that of the serial version and thus any parallelization gain will be lost [7]. In order to obtain a faster parallel program, these conflicted parameters need to be well optimized.

Execution time reduction is one of the most challenging goals of parallel programming.

Theoretically, adding extra processors to a processing system leads to a smaller execution time of a program compared with its execution time using a fewer processors system or a single machine[2].

Practically, when a program is executed in parallel, the hypothesis that the parallel program will run faster is not always satisfied. If the main goal of parallelizing a serial program is to obtain a faster run then the main criterion to be considered is the speedup gained from parallelization.

Various parallel programming paradigms can be used to write parallel programs such as OpenMP [8], Parallel Virtual Machine (PVM) [9], and Message Passing Interface (MPI) [10].

MPI is the most commonly used paradigm in writing parallel programs since it can be employed not only within a single processing node but also across several connected ones. MPI enables the programmer to control both data distribution and process synchronization. MPI standard has been designed to enhance portability in parallel applications, as well as to bridge the gap between the performance offered by a parallel architecture and the actual performance delivered to the application [11].

MPICH2 [12] is an MPI implementation that is working well on a wide range of hardware platforms and also supports using of C/C++ and FORTRAN programming languages.

In this paper we discuss some of the parameters that affect the parallel programs performance as a parallelization gain issue and also propose an experimental method to predict the speedup of MPI applications. We focus on the parallel programs written by MPI paradigm using MPICH2 implementation.

The paper is organized as follows: section 2 discusses the different challenging factors in MPI programming. Section 3 presents an experimental method to predict MPI-based application speedup. Theoretical speedup prediction is presented in section 4. In section 5, we present a comparison between the experimental and theoretical speedup prediction for some selected MPI-based applications.

2. MPI programming Challenging Factors

Several factors affect the performance of MPI-based parallel programs. These factors should be adapted to achieve the optimal performance.

2.1 Problem decomposition

When dividing the data into processes the programmer have to pay attention to the amount of load being processed by each processor. Load balancing is the task of equally dividing work among the available processes. This is easy to be programmed when the same operations are being performed by all the processes on different pieces of data.

Irregular load distribution leads to load imbalance which cause some processes to finish earlier than others. Load imbalance is one source of overhead, so all tasks should be mapped onto processes as evenly as possible so that all tasks complete in the shortest amount of time to minimize the processors' idle time which lead to a faster execution.

2.2 Communication pattern

In general, all data movement among processes can be accomplished using MPI send and receive routines. More over, a set of standard collective communication routines [13] are defined in MPI. Each collective communication routine has a parameter called a communicator, which identifies the group of participating processes. The collective communication routines allow data movement among all processors or just a specified set of processors [14].

The cost of communication in the execution time can be measured in terms of latency and bandwidth. Latency is the time taken to set up the envelope for communication, where bandwidth is the actual speed of transmission. Regardless of the network hardware architecture the communication pattern affects the performance of MPI programs. Using collective communication pattern is more efficient than using of point-to-point communication pattern [15], so the application programmer have to avoid using of the latter one as much as possible, specially for large size problems, for the following reasons:

1. Although point-to-point pattern is a simple way of specifying communication in parallel programs; its use leads to large program size and complicated communication structure, which negatively affect the program performance.
2. Send-receive operation does not offer fundamental performance advantages over collective operations. The latter offer efficient implementations without changing the applications.
3. In practice, using the non-blocking versions of send-receive, MPI_Isend and MPI_Irecv, often lead to slower execution than the blocking version because of the extra synchronization.

2.3 Message size

Message size can be a very significant contributor to MPI application performance. The effect of message size is also influenced by latency, communication pattern and number of processors. To achieve an optimal performance, the application programmer should take the following considerations into account:

1. In most cases, increasing the message size will yield better performance. For communication intensive applications, the smaller message size

reduces MPI application performance because latency badly affects short messages..

2. For smaller message size with less number of processors, it is better to implement broadcasting in terms of non-blocking point-to-point communication whereas for other cases broadcasting using MPI_Bcast saves time significantly.

2.4 Message passing protocol

There are two common message passing protocols, eager and rendezvous [16], [17]. Eager protocol is an asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive. Rendezvous protocol is a synchronous protocol which requires an acknowledgement from a matching receive in order to complete the send operation.
MPI message passing protocols affect the program performance. The performance is implementation dependent. So the application programmer has to consider the following circumstances:

1. In case of eager protocol, the receiving process is responsible for buffering the message upon its arrival, especially if the receive operation has not been posted [16]. This operation is based upon the implementation's guarantee of a certain amount of available buffer space on the receive process. In this case, the application programmer has to pay attention to the following requirements to achieve a reasonable performance
    a. Message sizes must be small.
    b. Avoid using of intensive communication to decrease the time consumed by the receive process side to pull messages from the network and/or copy the data into buffer space.
2. If the receiving process buffer space can't be allocated or the limits of the buffer are exceeded rendezvous protocol is used. In this protocol, sender process sends message envelope to destination process which receives and stores that envelope. When buffer space is available, destination process replies to sender that requested data can be sent, hence sender process receives reply from destination process and then sends data [17]. In this case, the application programmer has to pay attention to the following requirements to achieve a reasonable performance
    a. Message sizes must be large enough to avoid the time consumed for handshaking between sender and receiver.
    b. Using non-blocking sends with waits/tests to prevent program from blocking while waiting for a receiving confirmation from receive process.

2.5 Processors' number

Adding extra processors to the system reduces the computation time but increases the communication time. The increase in communication time may be larger than the decrease in computation time which leads to a dramatic decreasing of performance. In practice, speedup does not increase linearly as the number of processors increases but tends to saturate and accordingly the efficiency drops as the number of processors increases [5].

The effect of processor's number is also influenced by the problem size. Speedup and efficiency increase as the problem size increases on the same number of processors. If increasing the number of processors reduces efficiency, and increasing the problem size increases efficiency, the application programmer should be able to keep efficiency constant by increasing both simultaneously.

2.6 Running processes

MPI implementations allow the programmer to run his application using arbitrary number of processes and processors. The number of processes may be less than, equal to, or greater than the number of processors. It is common to develop parallel applications with a small number of processes on a single processor. As the application becomes more fully developed and stable, larger testing runs can be conducted on actual clusters to check for scalability and performance bottlenecks.

The number of processes per processor affects the application performance so the application programmer has to be aware of the following considerations:

1. In general, maximum performance is achieved when each process has its own processor. When the number of processes is less than or equal to the number of processors, the application will run at its peak performance. Since the total system is either underutilized (there are unused processors) or fully utilized (all processors are being used), the application is not hindered by several parameters such as context switching, cache misses, or virtual memory thrashing caused by other local processes [18].
2. Running too many processes, the processors will thrash, continually trying to give each process its fair share of run time.
3. Running too few processes may not enable the programmer to run meaningful data through his application, or may not cause error conditions that occur with larger numbers of processes.

3. Experimental Speedup prediction

In some cases, the predicted performance may differ from that achieved experimentally. Since both Amdahl's and Gustafson laws ignore the communication cost as mentioned in section 1, we present an experimental method to predict the speedup of MPI applications as a performance measure taking into account the communication cost.

Since modern parallel machines are very costly and not easy to be access, we used an experimental system consists of 8 DELL machines. Each of these machines consists of Intel i386 based P4-1.6GHz processor with 512MB memory running on Microsoft Windows XP Professional Service Pack 2. These machines are connected via a Fast Ethernet 100Mbps switch. These machines are not as powerful as the recent cluster machines in terms of the hardware and performance but they can reasonably perform for testing purposes and also for solving small and middle size parallel problems. MPICH2 version 1.0.6p1, is used as a message passing implementation.

3.1 The proposed method

The proposed method is summarized in the following steps:

1. Execute the serial version of MPI application on a single processor machine.
2. Record the serial execution time, T_s .
3. Execute the parallel MPI application on the same single processor machine repeatedly using arbitrary number of MPI processes, 1, 2, 3,…, n.
4. Record the parallel execution times, $Tp_1, Tp_2, ..., Tp_n$, for each run.
5. Graph the obtained results as a two dimensional graph. The X-axis for MPI processes number and the Y-axis for the parallel execution times, $Tp_1, Tp_2, ..., Tp_n$.
6. If the parallel execution time is rapidly increases as the number of MPI processes increases, this implies that the MPI application will exhibit a poor speedup if it is run in parallel on multiple physical processors.
7. If the parallel execution time remains constant or slowly increases as the number of MPI processes increases, this implies that the MPI application will exhibit a linear speedup if it is run in parallel on multiple physical processors.

We applied this method on two MPI applications. The first one solves the concurrent wave equation [19] which is represented by a partial differential

equation to describe the propagation of waves along a flexible vibrating string stretched between two points on the x-axis.

The second application uses brute-force algorithm [20], [21] to find the number of primes and also the largest prime number within an interval of integers [22]. The two applications are also executed in parallel on multiple physical processors. The recorded serial execution time, T_s for both applications is used to find out their experimental speedup to be compared with the predicted ones.

3.2 Predicted versus experimental results

The parallel MPI applications that solve both wave equation and prime numbers generator problems were executed; serial execution time, parallel execution time on a single processor using multiple processes and also parallel execution time on multiple processors for both problems are shown in table 1.

Table 1. Serial and parallel execution times for Wave Equation and Primes Generator

Problem	Serial execution time	Parallel execution			
		Single physical processor		Multiple physical processors	
		MPI processes	Execution time	Physical processors	Execution time
Problem 1 Wave Equation	0.80216	1	1.3561	1	1.3561
		2	3.6942	2	4.0952
		3	6.3833	4	1.2112
		4	9.4002	8	11.4501
		5	12.5629		
		6	15.301		
		7	18.1778		
		8	21.5001		
		9	24.1733		
		10	27.3349		
Problem 2 Primes Generator	55.625	2	55.5887	1	57.625
		4	55.464	2	32.6704
		8	54.9653	4	17.38331
		10	55.5158	6	11.58861
		16	55.1428	8	8.2103
		20	55.9213		

Applying the proposed speedup prediction method to wave equation problem using 10 MPI processes on a single physical processor we predicted that the application will exhibit a poor speedup if it is executed in parallel using multiple physical processors.

Our prediction is based on that the execution time is rapidly increases as the number of MPI processes as shown in figure 1.

To prove that our prediction was true, we executed the same MPI code on 8 physical processors. Knowing the execution time of the serial code version, the experimental speedup was calculated. Figure 2 shows that the maximum speedup achieved by 8 physical processors was only 0.66228534 and hence our prediction was true.

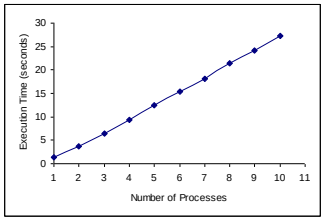


Fig. 1. Execution time using 10 processes on a single CPU for problem 1

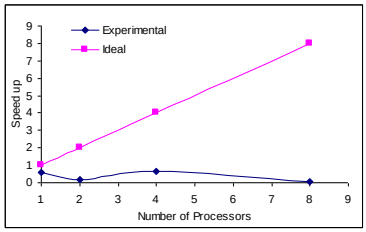


Fig. 2. Experimental speedup for problem 1

To be unbiased, we also re-executed the same parallel code using different number of processes on the same 8 physical processors. Figure 3 shows that the execution time was negatively affected as the number of MPI processes increases except in case of running a small number of MPI processes using 8 physical processors.

The experimental results show that there is no significant speedup improvement as shown in figure 4. This also proves that our prediction was true.

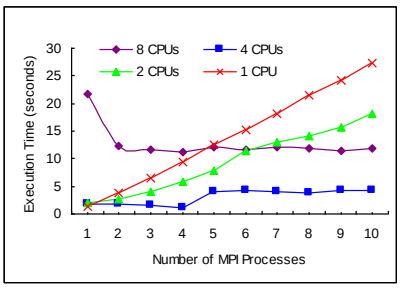


Fig. 3. Effect of processes number on execution time using 8 CPUs for problem1

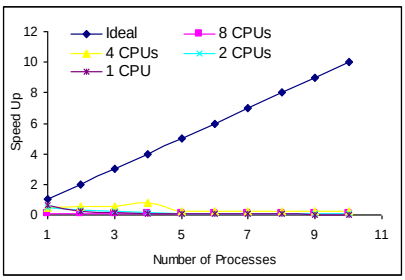


Fig. 4. Experimental vs. ideal speedup for problem 1

Applying the proposed method to prime numbers generator problem using 20 MPI processes on a single physical processor; we predicted that the application will exhibit a linear speedup if it is executed in parallel using multiple physical processors.

Our prediction is based on that the execution time is slowly increases or seems to be constant as the number of MPI processes as shown in figure 5. Running the same MPI code on 8 physical processors achieved a linear speedup as shown figure 6 and hence our prediction was also true.

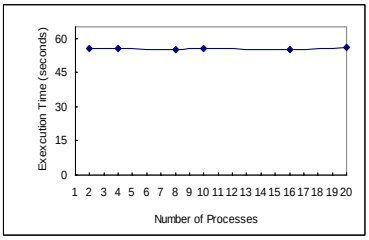


Fig. 5. Execution time using 20 processes on a single CPU for problem 2

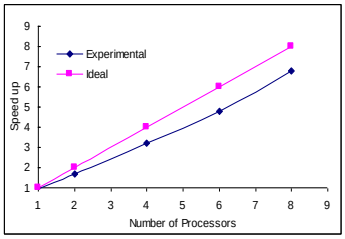


Fig. 6. Experimental speedup for problem2

4. Theoretical Speedup Prediction

From the theoretical point of view, speedup is challenged by various factors such as computation complexity and memory usage. Parallelizing sequential algorithms adds several extra challenges such as application speedup and how can it be affected by the number of cores and/ or the number of the running processes.

Several Studies [23], [24] , [25] have been addressed the performance of MPI applications on several hardware platforms, but little attention has been focused on using multi-core architectures supported by Microsoft Windows as an operating system and MPICH2 as an MPI implementation.

Three parallel sorting algorithms namely Bubble sort, Merge sort and Quick sort are designed and implemented using MPI. The effect of the number of cores and also the number of processes on the algorithms speedup is theoretically studied and compared with the experimental speedup.

4.1 Sorting Algorithms

The main function of sorting algorithms is to place data elements of a list in a certain order. Several algorithms are introduced to solve this problem.

Sequential sorting algorithms. Sequential sorting algorithms are classified into two categories. The first category, "distribution sort", is based on distributing the unsorted data items to multiple intermediate structures which are then collected and stored into a single sorted list. The second one, "comparison sort", is based on comparing the data items to find the correct relative order [26].

In this paper we focus on comparison based sorting algorithms. These algorithms use various approaches in sorting such as exchange, partition, and merge.

The exchange approach repeats exchanging adjacent data items to produce the sorted list as in case of bubble sort [27].

The partitioning approach is a "divide and conquer" strategy based on dividing the unsorted list into two sub-lists according to a pivot element selected from the list. The two sub-lists are sorted and then combined giving the sorted list as in case of quick sort [28].

Merge approach is also a divide and conquer strategy that does not depend on a pivot element in portioning process. The approach repeatedly divides the original list into sub-lists until the sub-lists have only one data item. Then these elements are merged together given the sorted list as in case of merge sort [29].

Parallelizing sorting algorithms. Parallelizing sorting algorithms needs a careful design to achieve well efficient results because of the high level date dependency evolved within these algorithms that exhibits parallelism.

Sequential versions of bubble sort, quick sort and merge sort are parallelized using C ++ binding of MPI under MPICH2 for Windows. The "scatter/ merge" paradigm is used in parallelization.

The used paradigm has three fundamentals phases, scatter phase, sort phase and merge phase. The first phase is responsible for distributing the original unsorted data list among the MPI process in such a way each of them accepts a part of the original data to be manipulated with these parallel processes.

In sort phase, each process sorts its local unsorted data list using one of the selected sorting algorithms. All local sorted data are sent from these "slave" processes to only one process which serves as a master process to generate

the sorted list in the merge phase. An MPI skeleton of this paradigm is shown in figure 7.

```
1. Initialize MPI environment.
2. Determine MPI processes' number (p) and their id's.
/* Scatter Phase  */
3. If  id=master then
4.     get unsorted list data  items of size n
5.     compute partition size, s = n/p
6.     broadcast s and n to all processes
7. endif
8. scatter sub-lists to all running processes
/*  Sorting Phase */
9. call Selected_Sorting_Algorithm ( sub-list, s)
/* Merge Phase */
10.  while step< p do
11.    if   id is even then
12.       Send even-sub-list to process id + 1
13.       Receive odd-sub-list from processor id + 1
14.       Merge even and odd sub-lists into sorted-list
15.       Replace even-sub-list by the first half of
          sorted-list
16.    else if id > 0 then
17.       Receive even-sub-list from process id - 1
18.       Send odd-sub-list to process id – 1
19.       Merge even and odd sub-lists into sorted-list
20.       Replace odd-sub-list by the second half of
          sorted-list
21.    end if
22. End while
23. Finalize MPI environment
24. End
```

Fig. 7. MPI scatter/ merge paradigm

4.2 Theoretical speedup prediction

In sequential bubble sort, the computation complexity is $O(n^2)$, *n* is the unsorted list size, in both best and average case.

The parallel version complexity of bubble sort based on "scatter/ merge" paradigm is estimated as $O(n^2/P^2)$, so the time will be reduced by a factor of P^2, *P* is the number of processors. This is due to the partitioning of the total size n of the original list among the running processes *P*. This implies a

theoretical super linear speedup. In case of using only two cores we expect that the computation complexity of parallel bubble sort will be $O(n^2/mP)$, *m* is the number of processes and P is the number of physical cores.

Sequential merge sort algorithm behaves as $O(n \log n)$ computational complexity in all of its cases, worst, average and best [30]. We estimated the parallelized version complexity in case of using a dual core processor as $\frac{2}{P}\log(n/m) + total\ overhead$. The total overhead is the sum of inter-process communications overhead and the MPI processes initialization overhead.

In case of sequential quick sort, the efficiency of the algorithm is influenced the pivot element selection method; we get worst case $O(n^2)$ when the selected pivot is the left most data item. If the pivot is carefully selected, the algorithm behaves in its best case as $O(n \log n)$ complexity [30]. In case of parallelized version, we estimated the complexity for a dual core processor as $\frac{2n}{mP}\log(n/m) + total\ overhead$.

5. Experimental- Theoretical Comparison

An experiment is carried out and applied to the implemented parallel version of the concerned sorting algorithms. The experiment is designed to address the affect of parallel processes number, and also the number of used cores on the execution time.

5.1 Steps of the experiment

1. Set the number of system cores to 1 and reboot the system.
2. Execute the parallel MPI application on the same single core repeatedly using arbitrary number of MPI processes, 1, 2, 3... , n for the same data with the same size.
3. Record execution time.
4. Set the number of system cores to 2 and reboot the system.
5. Repeat steps 2-4 with the same data and size.

We used an experimental system consists of Pentium[R] Dual-Core CPU E5500@ 2.80 GHZ, 3.21 GB of RAM running on Microsoft Windows XP

Professional Service Pack 2. The experiments codes were written in C++ using MPICH2 version 1.0.6p1, as a message passing implementation.

5.2 Experimental results

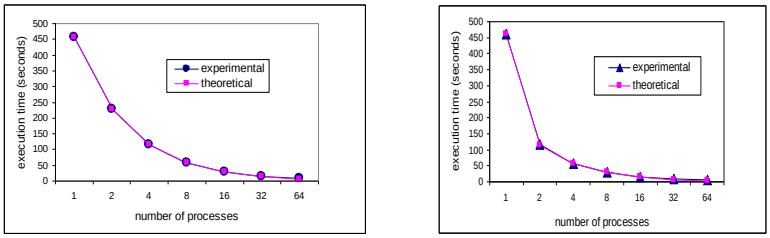
We applied the experiment to the three parallel implementations with a fixed data size 2×10^5 for bubble sort and 6×10^6 for merge sort and quick sort respectively with 1,2, ..64 parallel process using both a single and dual cores as shown in table 2.

We profiled the execution of the tested implementations using jumpshot [31] to address the inter-processes communication. Also the total overhead and computation costs are measured.

As the theoretical expectation, the execution time of bubble sort is reduced as the number of parallel processes increases in case of using either single or dual cores as shown in figure 8. On other hand merge sort and quick sort do not exhibit a speedup behavior as processes number increases as shown in figure 9 and figure 10.

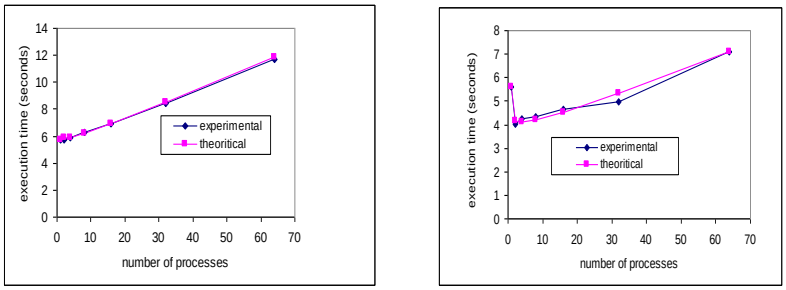
Table 2. Experimental Results

Number of cores	Number of processes	Execution time in seconds		
		Bubble sort, 2×10^5 date items	Merge sort, 6×10^6 date items	Quick sort, 6×10^6 Date items
1	1	457.375	5.718	4.437
	2	229.250	5.765	4.500
	4	115.859	5.906	4.562
	8	57.812	6.296	4.859
	16	28.953	6.953	5.421
	32	15.140	8.421	6.890
	64	9.0460	11.687	10.14
2	1	460.796	5.609	4.203
	2	117.546	4.031	3.203
	4	57.937	4.234	3.328
	8	29.109	4.328	3.453
	16	14.646	4.640	3.796
	32	7.625	4.968	4.265
	64	4.687	7.109	6.265



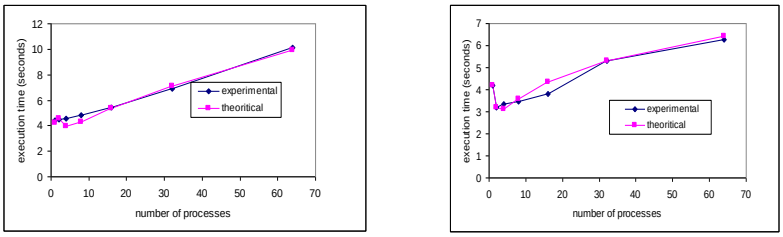
(a) single core (b) dual core

Fig. 8. Experimental and theoretical execution time of parallel bubble sort



(a) single core (b) dual core

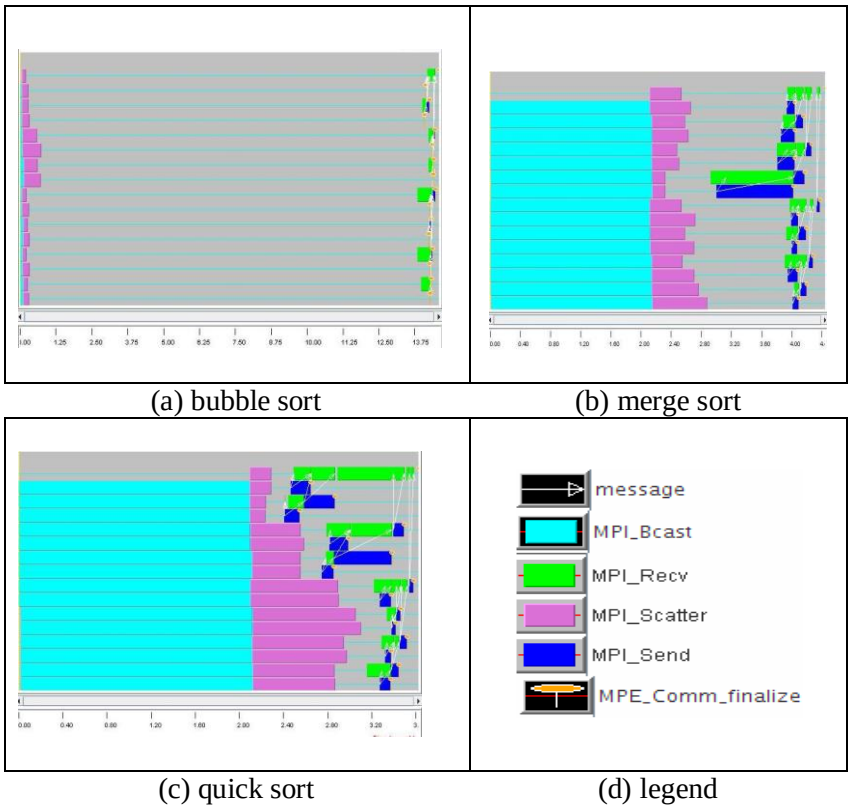
Fig. 9. Experimental and theoretical execution time of parallel merge sort



(a) single core (b) dual core

Fig. 10. Experimental and theoretical execution time of parallel quick sort

To interpret these result, we profiled the execution of the tested implementations using jumpshot to address the inter-processes communication. Also the total overhead and computation costs are measured. Figure 11 shows how the running parallel processes communicate with each others. In bubble sort (figure 11.a) there is a low communication overhead compared with that of computations in contrast to figures 11.b and 11.c that show a higher communication overhead. The excessive inter-process communications overhead noticed in both merge sort and quick sort increases the total execution time.

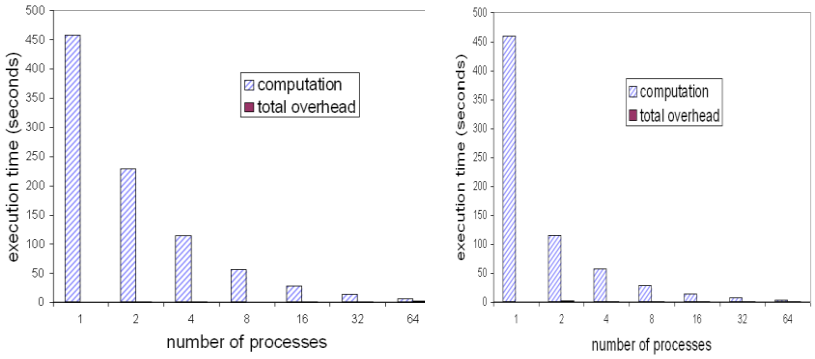


(a) bubble sort (b) merge sort

(c) quick sort (d) legend

Fig. 11. Jumpshot time line for the experiment

We also measured the total overhead of parallel processes compared with the computation cost for the three implementations regarding the number of processes and also the number of cores used as shown in figure 12 , figure 13 and figure 14.



(a) single core (b) dual core

Fig. 12. Bubble sort overhead/ computation ratio

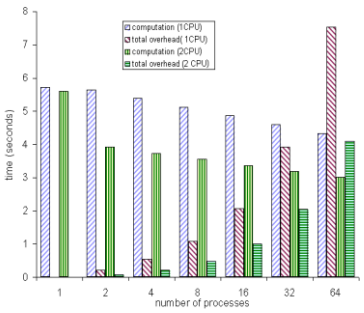


Fig. 13. Merge sort overhead/ computation ratio

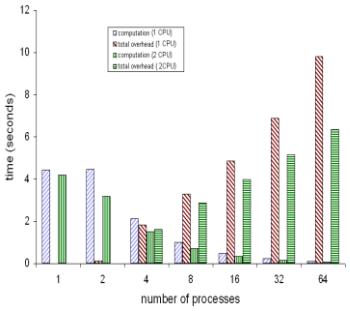


Fig. 14. Quick sort overhead/ computation ratio

6. Related Work

There are two major approaches for parallel performance prediction. The first approach is to build an analytical model for the application under consideration [32], [33], [34], [35]. The main advantage of this approach is its low cost. However, constructing analytical models of parallel applications requires a well understanding of the algorithms and their implementations, and also the models are constructed manually by domain experts, which limit their accessibility to normal users. Moreover, a model built for an application cannot be applied to another one.

The second approach is to develop a system simulator to execute applications to predict their performance. Simulation techniques can capture

detailed performance behavior at all levels, and can be used automatically to model a given program. However, an accurate system simulator is extremely expensive. Existing simulators such as BigSim and MPI-SIM [36], [37] are inadequate to simulate the very large problems.

Trace-driven simulation [38] and macro-level simulation [39] have better performance than detailed system simulators, since they only need to simulate the communication operations.

Yang et al. [40] proposed a cross-platform prediction method based on relative performance between target platforms without program modeling, code analysis, or architecture simulation.

Lee et al. [41] presented piecewise polynomial regression models and artificial neural networks that predict application performance as a function of its input parameters.

Barnes et al. [42] employ the regression based approaches to predict parallel program scalability and their method shows good accuracy for some applications. However, the number of processors used for training is still very large for better accuracy and their method only supports load-balanced workload.

Statistical techniques have been used widely for studying program behaviors from large-scale data [43].

In this paper we present an experimental method that can be used in speedup prediction for a parallel application without using neither high cost resources nor building an analytical models or simulators.

The experimental results were compared with the predicted theoretical results. We found that experimental results are very close to the theoretical ones

7. Conclusion

In this paper we have studied the conflicting parameters that affect the parallel programs execution experimentally and theoretically, especially for MPI-based applications, showing some recommendations to be followed to achieve a reasonable performance.

The problem nature is one of the most important factors that affect the parallel program speedup. If the problem can be divided into independent subparts and no communication is required, except to split up the problem and combine the final results, then the resultant parallel program will exhibit a linear speedup. If the same instruction set are applied to all data and processes communication is synchronous, speedup will be directly proportional to the computation -communication ratio. If there are different instruction sets to be applied to all data to solve a specific problem and the inter-process communication is asynchronous, the speedup of the resultant parallel application will be negatively affected with extra communication overhead.

We also proposed an experimental method that aids in speedup prediction. It was applied to predict the speedup of MPI applications that solve wave equation and prime numbers generator problems. The predicted speedup was as the same as experimental speedup achieved when using multiple physical processors for both applications.

Theoretical speedup prediction requires extra analysis such as algorithm complexity. The effect of both running processes number and the number of cores on the algorithm complexity have been studied for three parallel sorting algorithms.

The theoretical predicted speedup has been compared with the experimental speedup for the three algorithms. Our theoretical prediction of speedup was very close to the experimental results; this gives a good indication about the scalability of the proposed method.

References

1. Grama, A., Gupta, A., Kumar, V.: Isoefficiency Function: A Scalability Metric for Parallel Algorithms and Architectures. IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice, Vol. 1, No. 3, 12-21. (1993)
2. Amdahl, G. M.: Validity of the Single Processor Approach to achieving Large Scale Computing Capabilities. In Proceedings of the AFIPS Spring Joint Computer Conference. 483–485. (1967)
3. Sun, X., Chen, Y.: Reevaluating Amdahl's law in the multicore era. Journal of Parallel Distributed Computers, 183-188. (2010)
4. Hill, M. D., Marty, M. R.: Amdahl’s Law in the Multicore Era. IEEE Computer Society, Vol. 41, No. 7. 33-38. (2008)
5. Gustafson, J.: Reevaluating Amdahl's Law. Communications of the ACM, Vol. 31, No. 5. 532-533. (1988)
6. Karp, A. H., Flatt, H.: Measuring Parallel Processor Performance. Communication of the ACM Vol. 33 No. 5. (1990)
7. Donghwan, J. G., Chris, L., Michael, T.: Kismet: Parallel Speedup Estimates for Serial Programs. Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (2011)
8. Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., Woodall, T. S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In Proceedings, 11th European PVM/MPI Users’ Group Meeting, Budapest, Hungary. 97–104. (2004)
9. Sunderam, V. S.: PVM: A framework for parallel distributed computing. Concurrency: Practice & Experience, Vol. 2, No. 4. 315–339, (1990)
10. Aoyama, Y., Nakano, J.: Practical MPI Programming. International Technical Support Organization, IBM Coorporation SG24-5380-00. (1999)
11. Buntinas, D., Mercier, G., Gropp, W.: Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem. Parallel Computing, Vol. 33, No. 9. 634-644. (2007)

12. Gropp, W.: MPICH2: A New Start for MPI Implementations. In Recent Advances in PVM and MPI: 9th European PVM/MPI Users' Group Meeting, Linz, Austria, (2002)
13. The MPI Forum. The MPI-2: Extensions to the Message Passing Interface, Available at http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html. (1997)
14. Karwande, A., Yuan, X., Lowenthal, D. K.: CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters. Journal of Parallel and Distributed Computing, Vol. 65, No. 10, 1123-1133. (2005)
15. Gorlatch, S.: Send-Receive Considered Harmful: Myths and Realities of Message Passing. ACM Transactions on Programming Languages and Systems, Vol. 26, No. 1, 47–56.(2004)
16. Liu, J., Vishnu, A., Panda, D. K.: Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation. In Proceedings of the ACM/IEEE SC2004 Conference, 33 – 33. (2004)
17. Brightwell, R., Underwood, K. D.: Evaluation of an Eager Protocol Optimization for MPI. 10th European PVM/MPI Users' Group Meeting, Venice, Italy, 327-334. (2003)
18. Squyres, J. M.: Processes, Processors, and MPI. Cluster World, MPI Mechanic Vol. 1 No. 2. 8-11. (2004)
19. El-Nashar, A.: To Parallelize or not to Parallelize, Speedup Issue. International Journal of Distributed and Parallel Systems, IJDPS, Vol. 2, No. (2011).
20. Mohammad A., Saleh, O., Abdeen, R. A.: Occurrences Algorithm for String Searching Based on Brute-force Algorithm. Journal of Computer Science, Vol. 2, No 1, 82-85. (2006)
21. Atkin, O. L., Bernstein, D. J.: Prime sieves using binary quadratic forms. Mathematics of Computation Vol. 73. 023–1030. (2004)
22. Aziz, I., Haron, N., Tanjung, L. , Dagang, W. W.: Parallelization of Prime Number Generation Using Message Passing Interface. WSEAS Transactions on Computers, Vol. 7, No. 4. 291-303. (2008)
23. Mallón, D. A., Taboada, G. L. , Teijeiro, C., Touriño, J., Fraguela, B. B., Gómez, A., Doallo, R. , Mouriño, J. C.: Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. EuroPVM/MPI LNCS 5759, Springer Berlin Heidelberg. 174-184. (2009)
24. Thakur, R., Gropp, W., Toonen, B.: Optimizing the synchronization operations in MPI one-sided communication. International Journal of High-Performance Computing Applications, Vol 19 No. 2. 119–128. (2005)
25. Gropp, W., Thakur, R.: Revealing the Performance of MPI RMA Implementations. Proceedings of the 14th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2007), 272-280. (2007)
26. Rashid, L., Hassanein, W., Hammad, M.: Analyzing and enhancing the parallel sort operation on multithreaded architectures. Journal of Supercomputing, Vol. 53, No. 2. 293-312. (2010)
27. Astrachan, O.,: Bubble sort: An Archaeological Algorithmic Analysis. ACM SIGCSE Bulletin, Vol. 35 No. 1. (2003)
28. Tsigas, P., Zhang, Yi.: A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on Sun Enterprise 10000. Proceedings of the 11th EUROMICRO Conference on Parallel Distributed and Network-Based Processing (PDP). 372 – 381. (2003)

29. Lyer, B. R., Dias, D.M.: System Issues in Parallel Sorting for Database Systems. Proceedings of the International Conference on Data Engineering, 246-255. (2003)
30. Biggar, P., Gregg, D.: Sorting in the Presence of Branch Prediction and Caches. Technical Report TCD-CS-2005-57 Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland. (2005)
31. Chan, D. A. , Lusk, R., Gropp, W. :Jumpshot-4 Users Guide. Mathematics and Computer Science Division, Argonne National Laboratory. (2007)
32. Barker, K. J., Pakin S., and Kerbyson D. J.: A performance model of the krak hydrodynamics application. ICPP'06, 245–254. (2006)
33. Zhai, J., Chen, W., and Zheng, W.: PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node. PPoPP'10. Bangalore, India. ACM 978-1-60558-708-0/10/1. 305-314 (2010)
34. Kerbyson, D. J., Alme, H. J., Hoisie, A., Petrini, F., Wasserman, H. J., and Gittings, M.: Predictive performance and scalability modeling of a large-scale application. SC'01. 37– 48. (2001)
35. Mathias, M., Kerbyson, D., and Hoisie, A.: A performance model of non-deterministic particle transport on large-scale systems. Workshop on Performance Modeling and Analysis. ICCS (2003)
36. Wilmarth, T., Zheng, G. J. et al.: Performance prediction using simulation of large-scale interconnection networks in POSE. Proc. 19th Workshop on Parallel and Distributed Simulation. 109–118. (2005)
37. Zheng, G., Kakulapati, G., and Kale, L. V.: Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. IPDPS'04. 78–87. (2004)
38. Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., and Purkayastha, A.: A framework for application performance modeling and prediction. SC'02, 1–17. (2002)
39. Susukita, R., Ando, H., et al.: Performance prediction of large-scale parallell system and application using macro-level simulation. SC'08. 1–9. (2008)
40. Yang, L. T., Ma, X., and Mueller, F.: Cross-platform performance prediction of parallel applications using partial execution. SC'05, 40. (2005)
41. Lee, B. C., Brooks, D. M., and de Supinski, B. R. et al.: Methods of inference and learning for performance modeling of parallel applications. PPoPP'07, 249–258. (2007)
42. Barnes, B. J., Rountree, B., Lowenthal, D. K., Reeves, J., de Supinski, B., and Schulz, M.: A regression-based approach to scalability prediction. ICS'08. 368–377. (2008)
43. Sherwood, T., Perelman, E., Hamerly, G., and Calder, B.: Automatically characterizing large scale program behavior. ASPLOS. 45–57.(2002)

Alaa I. Elnashar received his B.Sc., M.Sc. and Ph.D. Minia University, Egypt, in 1988, 1994 and 2005. He is a faculty in Computer Science Dept., Minia University, Egypt. Dr. Elnashar was a postdoctoral fellow at Kanazawa University, Japan. His research interests are in the area of Software Engineering, Software Testing, and parallel programming. At present, Dr. Elnashar is an Assistant professor, Department of Computer Science, College of Computers and Information Technology, Taif University, Saudi Arabia.

Sultan Aljahdali, Ph.D. secured B.S. from Winona State University, Minnesota in 1992, M.S. with honor from Minnesota State University, Minnesota, 1996, and Ph.D. Information Technology from George Mason University, U.S.A, 2003. Currently Dr. Aljahdali is the Dean of the college of computers and information systems at Taif University. His research interest includes software testing, developing software reliability models, computer security, and medical imaging.

Received: May 29, 2012; Accepted: December 06, 2012