An Analytic Model for Predicting the Performance of Distributed Applications on Multicore Clusters

Nontokozo P. Khanyile, Jules-Raymond Tapamo, and Erick Dube

Abstract—**Computationally demanding applications can benefit from distributed processing. Distributed processing offers immerse computational power from collections of autonomous systems which can provide up to thousands of processing cores. Over the years, a considerable amount of research has been put towards developing performance prediction models for distributed applications. Amdahl's law states that the speedup of any parallel program has an upper bound which is determined by the amount of time spent on the sequential fraction of the program, no matter how small and regardless of the number of processing nodes used. This paper discusses some of the short comings of this law in the current age. We propose a theoretical model for predicting the behavior of a distributed algorithm given the network restrictions of the cluster used. The paper focuses on the impact of latency and bandwidth which affect the cost of interprocessor communication and the number of processing nodes used to predict the performance. The model shows good accuracy in comparison to Amdahl's law.**

Index Terms—**Latency, propagation delay, distributed programming, bandwidth, performance.**

I. Introduction

DISTRIBUTED systems are collections of autonomous computing systems which are connected by some network. These systems work together as one entity to solve large problems by splitting them up into smaller subproblems in a divide and conquer manner. The processing time of algorithms running on distributed systems is calculated as the total computation time plus the time spent on communication among the processors. Performance is an important part of software development. Clients often need to know the expected performance so that they can make an informed decision on whether or not to invest in a project. For developers, performance prediction gives a good idea of how the system may behave, allowing them to locate possible bottlenecks from the system before development [1]. Performance prediction methods in literature can be classified into three categories; analytical [2]–[7], profile-based [8], [9] and simulation-based [10]–[13]. Analytical methods work by decomposing an application into an algebraic expression [5] and model the performance mathematically. Simulators on the other hand analyze the source code directly, which reliefs users of the duty of having to analyze lengthy programmatic features into mathematical models. They characterize the code and the hardware it is running on and use the resulting models collectively to derive the predictive execution data. Although simulation-based approaches have high accuracy, they have high computational cost [14]. Existing simulations-based approaches include MPI-SIM [11], PACE [15], WARPP [16] and SimOs [12]. Analytical solutions have the advantage of efficiency over the rest of the prediction methods, however, it is limited by the fact that many complex systems are analytically intractable [17].

N.P. Khanyile is with the Modelling and Digital Sciences (Information Security) Division of the Council for Scientific and Industrial Research, Pretoria, South Africa and the School of Electrical, Electronic and Computer Engineering, University of KwaZulu-Natal, Durban, South Africa (e-mail: pkhanyile@csir.co.za).

J.-R. Tapamo is with the School of Electrical, Electronic and Computer Engineering, University of KwaZulu-Natal, Durban, South Africa.

E. Dube is with the Modelling and Digital Sciences (Information Security) Division of the Council for Scientific and Industrial Research, Pretoria, South Africa.

Amdahl came up with a law for predicting performance of parallel systems, which has long after been disputed. The skepticism surrounding Amdahl's law is over the assertion that parallel processing is unscalable [2]. Amdahl's law stipulates that even when the serial fraction of a problem, say s, is considerably small, the maximum attainable speedup is only $\frac{1}{s}$ even for an infinite number of processing nodes [2]. If s is the time spent by N processors executing the serial fraction of the computation time of a program and p is the time spent executing the parallel portion, then Amdahl's law states that the estimated speedup is given by:

$$Speedup = \frac{1}{(s + \frac{p}{N})},\ with\ s = 1\text{-}p \tag{1}$$

Amdahl's law assumes that the problem size remains fixed after parallelization. This, however, is usually not the case. It has been shown that in practise, parallel processing workload scales up with the number of processers [2], [3]. Gustafson [2] discussed the concept of scalable parallel processing and introduced the scaled-sized model for speedup. When it comes to parallel processing, Gustafson states that the parallel portion of the program scales up with the problem size, while the serial portion, comprised of program loading, serial bottlenecks and I/O, stays fixed.

Figure 1 plots 5 curves using Amdahl's law. From the graph, the assumption that p is independent of N is implicit, even though this is hardly ever the case [2]. Figure 2 plots 5 curves using Gustafson's law under conditions identical to those of Figure 1. The plot shows great scalability without any upper bounds. Gustafson argued that the workload of parallel problems scales up with the increase of processing nodes making the speedup linearly dependent on the number of processors N.

To derive Gustafson's law, consider using a serial processor to process the entire workload. It would take $s + pN$ to complete the task. From this the scaled speedup is calculated as:

$$\begin{aligned} Scaled_speedup &= \frac{(s + pN)}{(s + p)} \\ &= s + pN \end{aligned} \tag{2}$$

Hill & Marty [18] revised Amdahl's law for multicore architectures. In order to apply Amdahl's law in a multicore

(Advance online publication: 28 August 2012)

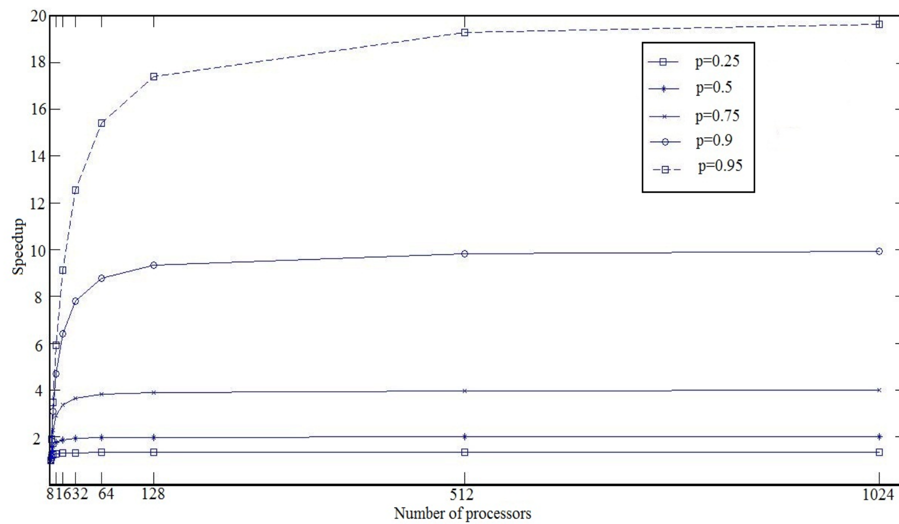

Fig. 1. Amdahl's fixed-size speedup

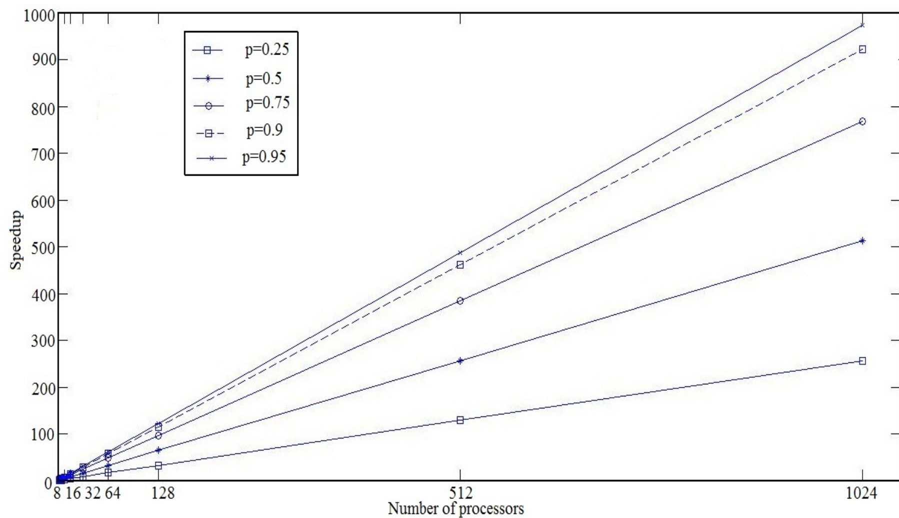

Fig. 2. Gustafson's scaled-size speedup

environment, a cost model for performance of the cores that the chip supports is required. Assume a symmetric multicore architecture with each core having its own L1 cache, where the memory bound is the cumulated capacity of the L1 caches. Variable $perf(r)$ is defined as the sequential performance of a powerful core with r Base Core Equivalents (BCEs). *"Under Admahl's law, the speed up of symmetric multicore chips depends on the software fraction that is parallelizable (f), the total chip resources in the BCEs (n), and the BCE resource (r) devoted to increasing each core's performance"* [18]. The resulting speedup for the symmetric multicore architectures is as follows:

$$Speedup(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f.r}{perf(r).n}} \quad (3)$$

Figure 3 is a plot of 6 curves at different f-values. Like Amdahl's law, Hill & Marty's corollary lacks scalability. Since the corollary applies Amdahl's concepts, it made the same inaccurate assumption that problem workload remains fixed after parallelization. This assumption lead to the conclusion that multicore architectures' scalability is questionable, which was quickly challenged by Sun & Chen [3]. Sun & Chen applied the scalable computing principals presented by Gustafson's law. The same hardware model architecture proposed by Hill & Marty was used to demonstrate the scalability of multicore architectures through a fixed-time model (as opposed to fixed-size) [3]. Sun & Chen define the fixed-time speedup as:

$$Speedup_{FT} = \frac{\textit{Sequential Time of Solving Scaled workload}}{\textit{Parallel Time of Solving Scaled workload}} \quad (4)$$

Let w be the original workload and w' be the scaled workload. Supposing the time taken to process w sequentially is the same as the time taken to process w' in parallel using m processors.

Assuming that the scale of the workload is only on the parallel portion; w' becomes:

$$w' = (1 - f)w + mfw \quad (5)$$

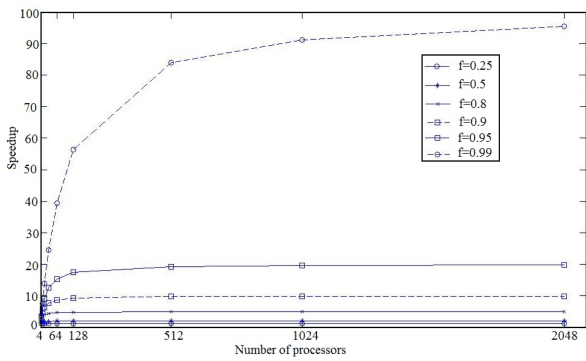

Fig. 3. Hill & Marty's fixed-size performance model

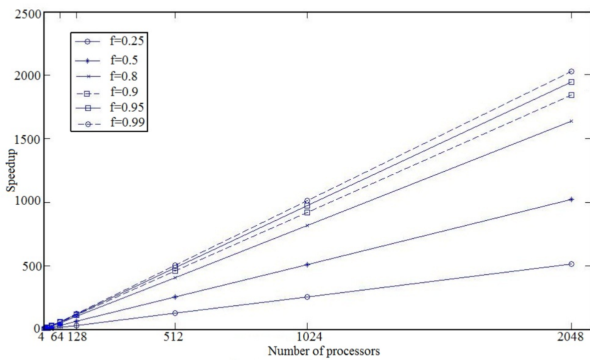

Fig. 4. Sun & Chen's fixed-time performance model

Therefore

$$Speedup_{FT} = \frac{\textit{Sequential Time of Solving w'}}{\textit{Parallel Time of Solving w'}} \quad (6)$$

$$Speedup_{FT} = \frac{\textit{Sequential Time of Solving w'}}{\textit{Sequential Time of Solving w}} \quad (7)$$

$$\frac{w'}{w} = \frac{(1-f)w + mfw}{w} = (1-f) + mf \quad (8)$$

which gives Gustafson's law [2]. The scaled-sized model assumes that the scaling is only at the parallel portion. Based on this assumption and following (3), Sun & Chen constructed the fixed-time speedup model to be:

$$\frac{(1-f)w}{perf(r)} + \frac{fw}{perf(r)} = \frac{(1-f)w}{perf(r)} + \frac{fw'}{perf(r)m} \quad (9)$$

If we let $n = mr$ be the scaled number of cores, with $n = r$ being the initial point, then $w' = mw$. The final scaled speedup compared with $n = r$ becomes:

$$\begin{aligned} Speedup_{FT} &= \frac{\textit{Sequential Time of Solving w'}}{\textit{Sequential Time of Solving w}} \\ &= \frac{\frac{(1-f)w}{perf(r)} + \frac{fw'}{perf(r)}}{\frac{w}{perf(r)}} = (1-f) + mf \end{aligned} \quad (10)$$

Figure 4 shows 6 curves of the fixed-time performance model at different f-values. The fixed-time speedup model demonstrates the scalability of multicore systems. Like the scaled-sized model, it is linearly dependent on the number of processors m. Although Sun & Chen successfully model the performance of parallel processing on multicore systems, they do not cater for distributed multicore systems.

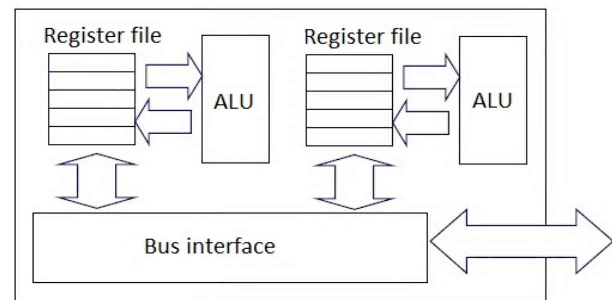

Fig. 5. Structure of a dual-core system (taken from [19])

Distributed computation differs from parallel computation in the way in which memory is used. In parallel systems, all processing elements use the same shared memory for communication and I/O, whereas distributed systems are autonomous systems with private memory connected by a network which is used for communication between the processing nodes. Fig. 5 shows an internal structure of a parallel/shared memory system in the form of a dual-core system, while Fig. 6 shows an example of a multicore distributed system, where multicore machines are combined by a network to function as one.

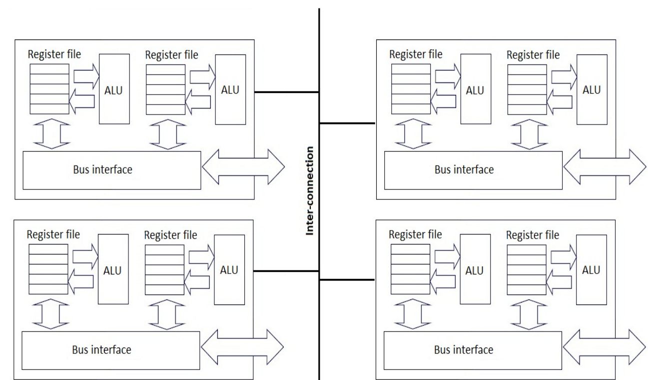

Fig. 6. Example structure of a multicore cluster (adapted from [19])

Multicore clusters allow for mixed-mode parallel and distributed processing through multi-level parallelism. The distributed processing makes up the coarse-grain parallelism while the parallel (shared-memory) processing makes up the fine-grain parallelism within each processing node. This multi-level parallelism leverages the cluster architecture by matching the hardware hierarchy. Figure 7 shows a graphical view of the multi-level parallelism structure. Message passing paradigms such as MPI and PVM can be used for coarse-grain parallelism with multiple threads running on each core for fine-grain parallelism.

Parallel code often runs on the same system and thus has no need for external communication. Distributed code, on the other hand, can not work without external communication. This communication, however, often consists of some overhead which, in large amounts, can affect performance drastically. The performance prediction models discussed above do not address the communication issue associated with distributed processing. For this reason, this paper presents

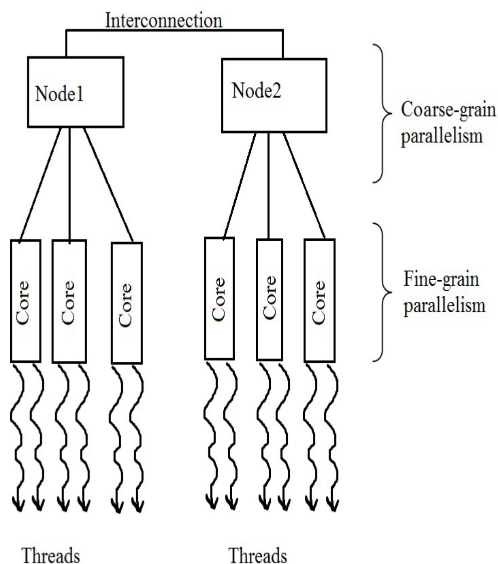

Fig. 7. Multi-layer parallelism architecture

a way of predicting performance for code distributed on multicore clusters. The rest of the paper is organised in the following manner: Section II discusses the factors which influence the performance of a distributed system. Section III gives the related works and background. Section IV discusses the proposed performance model. Section V analyzes the results and Section VI concludes the paper.

II. Factors Affecting Performance in Distributed Systems

The performance of a distributed algorithm is affected by more than just the application efficiency and the number of processing nodes used. Since clusters are connected by networks, network factors like latency and bandwidth have a considerable impact on the performance of a distributed system. As such, it is necessary to take into account the network influence when predicting the performance of these systems.

Bandwidth and latency capture the volume and time dimensions of information processing, respectively. Latency measures the time taken to complete a request, while bandwidth measures the volume of information transmitted in a time interval [20]. The next subsections go into greater detail about the factors that impact information processing performance.

A. Application Efficiency

Algorithm efficiency is the most crucial factor when it comes to parallel algorithm performance. If an algorithm is not efficient in how it utilizes resources, even the most powerful machines can not improve its performance. Distributed algorithms have to be optimized in two levels: per-processor (i.e. each core of a machine) and across-processors (i.e. communication across the cluster). Optimization techniques include code modifications and compiler optimizations. Per-processor optimizations include but no limited to [21]:

1) Loop optimization
    a) Unrolling
    b) Splitting
2) Memory optimization
    a) Prefetching
    b) Cache alignment and coherence
    c) Stride-one memory access
3) Floating point arithmetic
4) Use of optimized mathematical libraries

Across processors optimizations mainly deal with [21]:

1) Minimizing:
    a) Communication overhead
    b) Synchronization overhead
    c) Load imbalance
    d) Memory consumption
    e) Computation overhead
2) Latency hiding techniques such as overlapping computation and commucation
3) Efficient network interconnection
4) Avoiding master-only operations as they force the rest of the processors to idle

Lastly, compilers like GNU come with optimization flags such as *-ffast-math* which optimizes mathematical functions.

B. Application Latency

Application latency is defined by Shaffer [4] as the total amount of time that an application has to wait for a response after issuing a request for some data. The application delay reflects the total wait time incurred by the system, including all subsystems and kernel overhead as well as network latency [4].

Network latency is the time spent waiting, from the instantiation of an operation until the return of the desired results [4]. A distinction can be made amongst the different types/sources of network latency. Three types of network latencies are discussed; the propagation delay, transmission delay, and physical latency. Fig. 8 is a representation of these two network latency sources as a single server open queueing system.

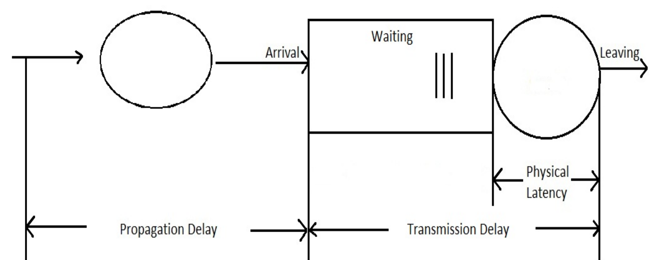

Fig. 8. Network latency presented as a propagation and transmission delay server (adapted from [20])

1) Propagation Delay: Propagation delay is defined as the time taken waiting for the last bit to arrive plus the overhead that comes with the device [4]. The propagation delay can not be eliminated or avoided because the speed of light is inviolable [4]. Transmission speed can not be improved beyond propagation delay. A propagation of a certain system indicates the maximum transmission rate that the system can achieve.

2) Physical latency: Physical latency measures the processing time on a device without waiting (i.e. the service time). It varies according to device utilization or load [20]. The physical latency can be halved to double the bandwidth. Ding [20] showed that halving the physical latency yields better results than actually doubling the bandwidth. The system is able to perform twice the amount of work without saturation.

3) Transmission Delay: Transmission delay is the amount of time taken to transmit all the packet's bits into the link. In most networks, transmission of packets occurs in a first come first serve manner, which results in queueing for transmission rates that are not high [22]. It is determined by the packet size and the transmission rate of the network and not at all affected by the distance.

C. Bandwidth

Bandwidth determines how much information can be processed within a certain time interval. It has a direct impact on the response time of data acquisition [20]. Low bandwidth can result in extremely slow systems. If an application must be able to transmit at a certain rate in order to be effective, then that application must transmit and receive at that rate. If that amount of bandwidth is not available, the application is most likely to give up [22]. Bandwidth may be increased to improve performance of a certain system and compensate for the propagation delay. However, increasing the bandwidth does not automatically guarantee performance gain. In order to benefit from high bandwidth, software often needs to be modified in order to leverage the high bandwidth. For example, applications developed for 32-bit systems may run slower on 64-bit systems [20].

III. Performance Model

Efficiency of a parallel algorithm is measured by the speedup attained. If T_1 is the execution time for the serial implementation, the speedup can be computed as $\frac{T_1}{T_N}$, where T_N is the execution time attained when using N processors. Efficiency is then calculated as:

$$E_N = \frac{T_N}{N} \tag{11}$$

An efficient algorithm attains a speedup close to N for every T_N, (i.e. $E_N = 1$). It has been established in the literature that for distributed systems, this is not always the case. As the number of processors increases, speedup of the distributed systems starts to decline. This is usually because of the increased interprocessor communication, known as message passing. Adding computation nodes increases the networks communication links which ultimately increases propagation delay.

This research focuses on the performance of multicore clusters using an analytical approach based on Sun & Chen's [3] and Shaffer's [4] works. Multicore clusters are ideal for hybrid programming, (i.e. a mixture of distributed and parallel processing). While multicore systems are scalable and provide high performance, they have their limits. Writing thread-safe programs is not easy, especially as the number of threads increases. Enrico Clementi, a former IBM fellow and pioneer in computational techniques for quantum chemistry and Molecular dynamics, once said *"I know how to make 4 horses pull a cart - I don't know how to make 1024"*. Introducing clustered systems relieves the strain of using too many threads on one machine.

A. Computational Cost

In distributed processing, an application can only run as fast as the slowest processor. Thus, following Sun & Chen's fixed-time model, we redefine $perf(r)$ to be:

$$perf(r') = max(perf(r_i)) \tag{12}$$

where $perf(r_i)$ is the sequential performance of a powerful core of a processing node i with r BCEs. Using (10) and the assertion that $w' = mw$ we get the following,

$$\frac{(1-f)w}{perf(r')} + \frac{fw'}{pref(r')m} = \frac{\frac{(1-f)w}{perf(r')} + \frac{fw}{perf(r')}}{\frac{w}{perf(r')}} = (1-f)+mf \tag{13}$$

This gives us the expected speedup [23].

B. Communication Cost

The communication overhead associated with message passing can be quite large. Shaffer [4] proposed a theoretical predictive measure of communication cost in wide area distributed systems to be:

$$Comm_Time = m \times [\frac{s}{b} + d \times 7.67 \times 10^{-6} + \epsilon] \tag{14}$$

where m is the frequency of messages needed during the task, b is the bandwidth in bits/second, ϵ is the overhead incurred per message and s and d represent the size of the message and the length of the communication channel in miles, respectively.

Propagation delay is normally calculated as the reciprocal of the speed of light which is currently 299792.458 km/s. However, Shaffer stated that this value is not the same for all types of cables. Different types of cables transmit at different speeds, which is less than the actual speed of light. This speed is known as the normal velocity of propagation (NVP). Optical fiber has an NVP close to 0.7 [4].

We define the cost of sending an L bit message between two processors as:

$$T_{comm}(L) \leq \frac{L}{\tau} + (\sigma_{max} \times dist) + \epsilon_L \tag{15}$$

where τ is the upper bound of the network bandwidth, σ_{max} is the maximum delay incurred by the system, $dist$ is the physical distance between the network points and $\epsilon(L)$ is the overhead associated with each message of size L bits, i.e. the send and receive overhead [23].

IV. Experiments

The total estimated running time is calculated as:

$$T_{EST}(m) = T_{comp}(m) + T_{comm}(m, L) \tag{16}$$

where $T_{comp}(m)$ is the computation time as defined in Section IV(a) and $T_{comm}(m, L)$ is the total time spent by m nodes communicating messages of sizes L.

$$T_{comp}(m) = T_{seq}/Speedup_{FT} \tag{17}$$

$$T_{comm}(m, L) = \sum T_{comm}(L) \quad (18)$$

T_{seq} is sequential time and $Speedup_{FT}$ is defined in (13) [23].

The performance model was tested on two different applications; one communication intensive and one computationally intensive. The experiments were performed on a cluster with the specifications given in table 1.

TABLE I
SYSTEM CONFIGURATION

Model	SuperMicro
Filesystem	GPFS
Network	Gigabit Infiniband
Number of nodes	5
CPU Cores	80
CPU Cores per Node	16
CPU Model	Intel Xeon
CPU Speed	2.4 GHz
Peak Performance	16 TFlops

A. Application 1: Distributed Fingerprint Enhancement Application

The fingerprint enhancement application consists of a series of computationally intensive image processing operations [24], [25]. The enhancement algorithm has been parallelized and distributed using MPI and OpenMP on an earlier publication [24]. Image processing operations require intensive communication with neighbouring processors in order to access neighbouring pixels along the boundaries.

B. Application 2: Large Array Application

This application takes in two large array of numbers in two 1 MB files, then performs a series of mathematical functions (matrix multiplication, covariance matrix, transpose and eigenvectors) and finally writes back the results to a 4 GB file. The application has minimal interprocessor communication. Only the file names are broadcasted.

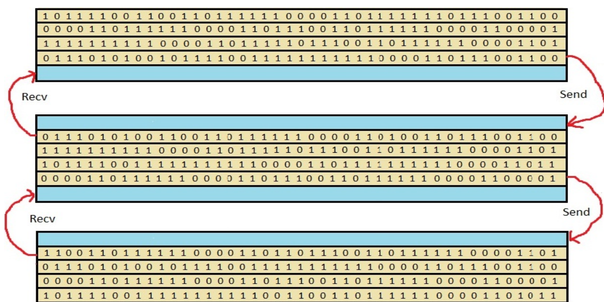

Fig. 9. Halo exchange between three processes

The algorithm [24] proposes three ways of dealing with boundary pixels: files locking, halo exchange and process-rank ordering.

1) **Halo exchange:** To access the boundary pixels, processors establish communication with their neighbouring processors. Each process is allocated ghost cells along the boundaries to hold the data received from neighbours. Processors then send/receive the boundary pixels using message passing. Figure 9 shows three processors engaging in a row-wise halo exchange. Halo exchange is an expensive operation and unless system architecture is optimized for it in its design, often incurs large overhead which need to be minimized.
2) **File locking:** To avoid large communication overhead, an alternative is to overlap data between processors along the boundaries. Figure 9 shows a fileview consisting of overlapped regions. This eliminates the need for communication but introduces a new problem of data consistency. When more than one processor access and operate on the data on the same file region, it often results in data racing and since processors arrive in any arbitrary order, it is impossible to determine beforehand which processor will write/read. For this reason, operational atomicity is required. File locking is perhaps the most intuitive solution to this process. It enforces explicit file locks on the overlapped regions to grant processors exclusive access. These locks can be achieved through the use of mutual exclusion synchronization. When a processor is operating on a specific file region, no other processor can operate on that region.
3) **Process-rank ordering:** Process-rank ordering also uses overlapped boundary file regions but instead of locking the file region, a higher ranking processor is granted exclusive access whenever an overlap is encountered. Lower ranking processor must then modify its request by subtracting the overlaps. This effectively removes all the overlaps and achieve atomicity [24]. Figure 10 shows modified access patterns based on process-rank ordering.

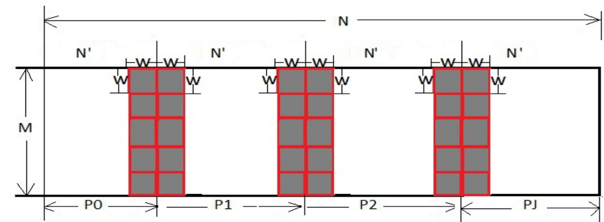

Fig. 10. Data overlapped along processor boundaries. Each overlap portion is of size $W \times M$ in order to ensure that the overall sub-image sizes are in multiples of processing block sizes

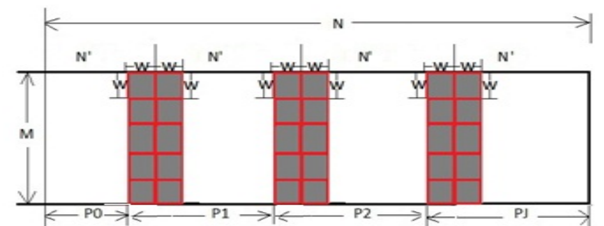

Fig. 11. Overlapped data access using process-rack ordering (adapted from [26])

Hence, the application has a lot of message passing making it a bit harder to scale with respect to processing nodes.

C. Results Analysis

The system used for testing has an NVP of 0.67, hence the expected propagation delay per km is

$$propagation_delay = \frac{1}{299792.458 * 0.67} = 0.0049ms \quad (19)$$

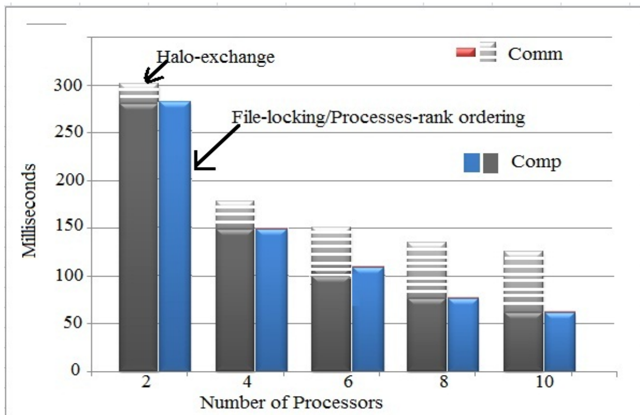

Fig. 12. Results of the predicted performance for the fingerprint enhancement application

The fingerprint enhancement algorithm was tested on 60 CASIA fingerprint database. Fig. 12 shows the results obtained using the prediction model separated into communication and computation time. From the graph, a drastic increase in the communication time with the increase of processing nodes can be observed on the halo exchange predictions. This is mainly due to the frequency of message passing during the prefetching of boundaries cells as well as the size of the messages.

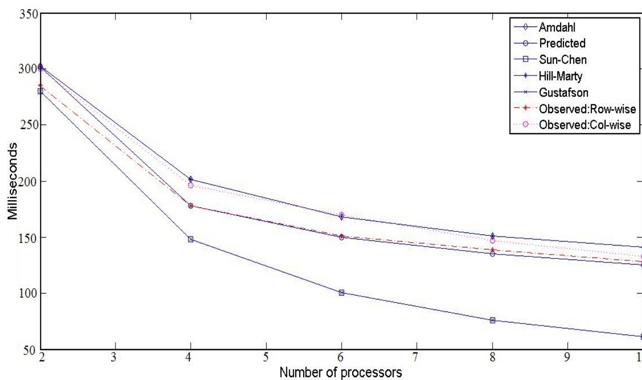

Fig. 13. Experimental results from the halo exchange experiments plotted against the 5 prediction models

A total of 3 sets of experiments were performed using the algorithm. Fig. 13 shows results obtained from using halo exchange on both row- and column-wise partitions of the data plotted with the predictions from the 5 models. From experiment its clear to see that Gustafson's and Sun & Chen's models over estimates the speedup, whereas Hill & Marty's and Amdahl's models under estimates. Gustafson's and Sun & Chen's models do not consider the effects of communication associated with distributed systems. Our model does not give the exact estimates, it over estimates the speedup but the error margins are better than those experienced by other models.

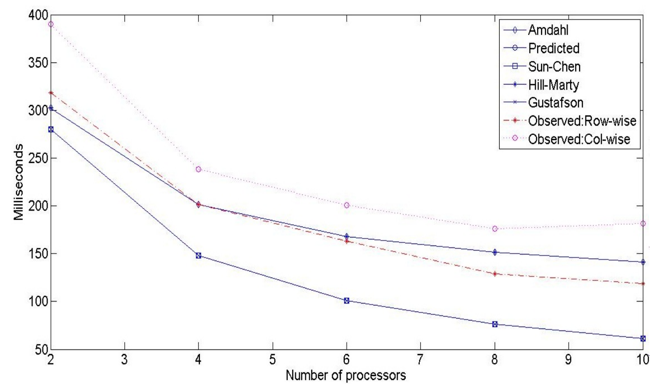

Fig. 14. Experimental results from file locking experiments plotted against the 5 prediction models

Fig. 14 shows the results obtained of using file locking on both row- and column-wise partitions plotted with the predictions made by the 5 models. The performance observed from this experiment shows a good example of poor optimization techniques and bad design choices. When using column-wise partition, file locking is the worst design choice one can make. Column-wise partition causes noncontiguous access patterns. What this mean is if each processor is assigned $M \times N'$ subarray of an image, then the distance between 2 consecutive rows is $N > N'$. It is not possible for a processor to access all its data in a single read/write operation.

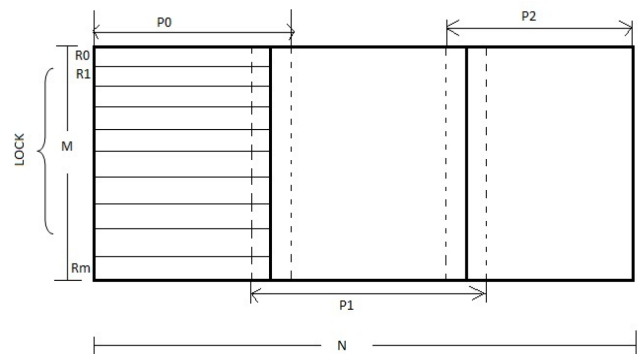

Fig. 15. File locking in column-wise partition

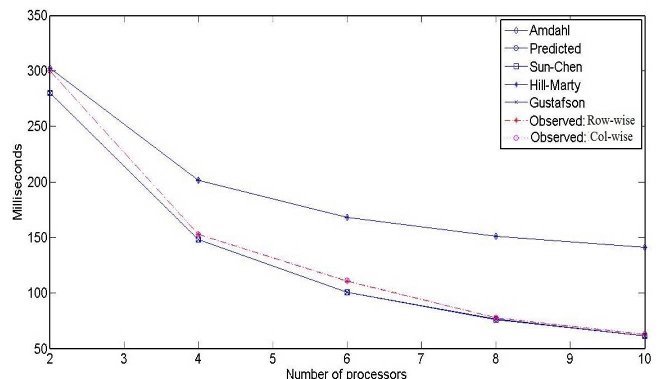

Fig. 16. Experimental results from the process-rank ordering experiments plotted against the 5 prediction models

For processor $P0$ in Fig. 15 to write to the first element of $R1$ after writing to the first element of $R0$, it needs to

traverse all the way to the end of the row (i.e. N columns) before returning to $R1$. Hence, locking an overlapped region ultimately locks the entire file, rendering I/O parallelism useless. Row-wise partition is better in this manner as it does not lock the entire the entire file.

Fig. 16 plots the results obtained using process-rank ordering on row- and column-wise partitions. Process-rank ordering maintains full I/O parallelism for both column-wise and row-wise partitions and hence is expected to out-perform the file locking strategy. Row-wise partition requires less effort to construct, while column-wise partition requires a little more effort due to the noncontiguous access patterns. This would explain the slight variation the performance. Our model, Gustafson's and Sun & Chen's models estimations produces better error margins than those produced by Hill & Marty and Amdahl's models.

Fig. 17 shows the prediction results vs. the observed performance of application 2. A comparison between Amdahl's law and our model is made. Amdahl's predictions show unscalability, performance does not improve beyond the 114 seconds mark. Our model shows great scalability and the error margins are better than those experienced by Amdahl's law. In comparison to computation, communication time is insignificant hence the model becomes almost identical to that of Sun & and Chen's. Also note that Hill & Marty and Gustafson's model give results identical to Amdahl's and Sun & Chen's models, respectively.

V. Conclusions

When communication is insignificant in comparison to the computation, our model gives the same performance as that of Gustafson's and Sun & Chen's models. But when there is extensive communication, our model strives. While Gustafson's and Sun & Chen's models over estimate the speedup, Hill & Marty's and Amdahl's models under estimate it. Our model considers both the computation capability of multicore systems as well as the limitations of the network. The experiments show the importance of choosing the right design strategies. The choice of data partition can have a huge impact on the performance of an algorithm, this evident on the column-wise file locking experiment. Optimizations are also crucial in distributed applications, as unoptimized code fails to scale upwards.

Acknowledgment

We would like to thank the Center for High Performance Computing (CHPC) for granting us access to their clusters.

References

[1] H. Wang, Y. Teo, and S. Tay, "An analytic method for predicting simulation parallelism," April 2000, pp. 211 – 218.

[2] J. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, May 1988.

[3] X.-H. Sun and Y. Chen, "Reevaluating amdahl's law in the multicore era," *Journal of Parallel and Distributed Computing*, vol. 70, pp. 183–188, 2010.

[4] J. H. Shaffer, "The effects of high bandwidth networks on wide area distributed systems," *PhD Thesis, University of Pennsylvania*, 1996.

[5] A. van Gemund, "Symbolic performance modelling of parallel systems," *IEEE Tans. on Parallel and Distributed Systems*, vol. 14, no. 02, pp. 154–165, 2003.

[6] D. Sundaram-Stukel and M. Vernon, "Perdictive analysis of the waveform application using LogGP," in *Proc. 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, vol. 34, no. 08, 1999, pp. 141–150.

[7] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eickev, "LogP: towards realistic model for parallel computation," in *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, vol. 28, no. 07, 1993, pp. 1–12.

[8] J. Bourgeois and F. Spies, "Performance prediction of the NAS benchmark program with ChronosMix environment," in *Euro-Par '00: Proc. 6th Int'l Euro-Par Conf. on Parallel Processing*. Springer-Verlag, 2000, pp. 208–216.

[9] R. Saavedra and A. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Trans. on Computer Systems*, vol. 14, no. 04, pp. 344–384, 1996.

[10] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: A generic framework for large-scale distributed experiments," in *Proc. 10th Int'l Conf. Computer Modelling and Simulation*. IEEE Computer Society, 2008, pp. 126–131.

[11] S. Prakash and R. Bagrodia, "MPI-SIM: Using parallel simulation to evaluate MPI programs," in *WSC: Proc. 30th Conf. Winter Simulation*. IEEE Computer Society Press, 1998, pp. 467–474.

[12] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod, "Using the SimOs mechine simulator to study complex computer systems," *ACM Trans. on Modelling and Computer Simulation*, vol. 07, no. 01, pp. 78–103, Jan 1997.

[13] Y. Luo, "MPI performance study on the SGI origin 2000," *Pacific Rim Conf. on Communications, Computers and Signal Processing*, pp. 269–272, 1997.

[14] B. Cornea and J. Bourgeois, "Performance prediction of distributed applications using block benchmarking methods," in *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb 2011, pp. 183–190.

[15] S. Jarvis, D. Spooner, H. K. L. Choi, J. Cao, S. Saini, and G. Nudd, "Performance prediction and its use in parallel and distributed computing systems," *Future Generation Computer Systems*, vol. 22, no. 7, pp. 745–754, Aug 2006.

[16] S. Hammond, G. Mudalige, J. Smith, S. Jarvis, J. Herdman, and A. Vedgama, "WARPP - a toolkit for simulating high-performance parallel scientific codes," *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, vol. 41, no. 7, pp. 19:1–19:10, March 2009.

[17] R. Bagrodia, E. Deelman, S. Docy, and T. Phan, "Performance prediction of large parallel applications using parallel simulations," *ACM SIGPLAN 1999 Symposium on Principles and Practice of Parallel Programming*, pp. 151–162, May 1999.

[18] M. Hill and M. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[19] J. Barbic. (2007, May) Multi-core architectures. Lecture Notes. [Online]. Available: http://www.co-array.org/cafvsmpi.htm

[20] Y. Ding, "Bandwidht and latency: Their changing impact on performance," in *Proc. 35th Computer Measurement Group Conference*, 2005, pp. 67–78.

[21] R. Rabenseifner, G. Hager, G. Jost, and R. Keller, "Hybrid MPI and OpenMP parallel programming: MPI + OpenMP and other models on clusters of SMP nodes," *Tutorial M09 at SUPERCOMPUTING*, Nov 2008.

[22] J. Kurose and K. Ross, *Computer Networks: A Top-Down Approach Featuring the Internet*. Addison-Wesley, 2001.

[23] N. P. Khanyile, R.-J. Tapamo, and E. Dube, "Performance prediction model for distributed applications on multicore clusters," *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engingeering 2012, WCE 2012*, vol. 02, pp. 1119–1124, July 2012.

[24] N. P. Khanyile, E. Dube, and R.-J. Tapamo, "Distributed fingerprint enhancement on a multicore cluster," *The 16th International Conference on Image Processing, Computer Vision, and Pattern Recognition*, vol. 02, pp. 890–897, July 2012.

[25] L. Hong, Y. Wan, and A. Jain, "Fingerprint image enhancement: Algorithm and performance evaluation," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 20, no. 08, pp. 777–789, Aug 1998.

[26] W.-K. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and N. Pundit, "Scalable design and implementations for MPI parallel overlapping I/O," *IEEE Trans. Parallel and Distributed Systems*, vol. 17, no. 11, pp. 1264–1276, Nov 2006.

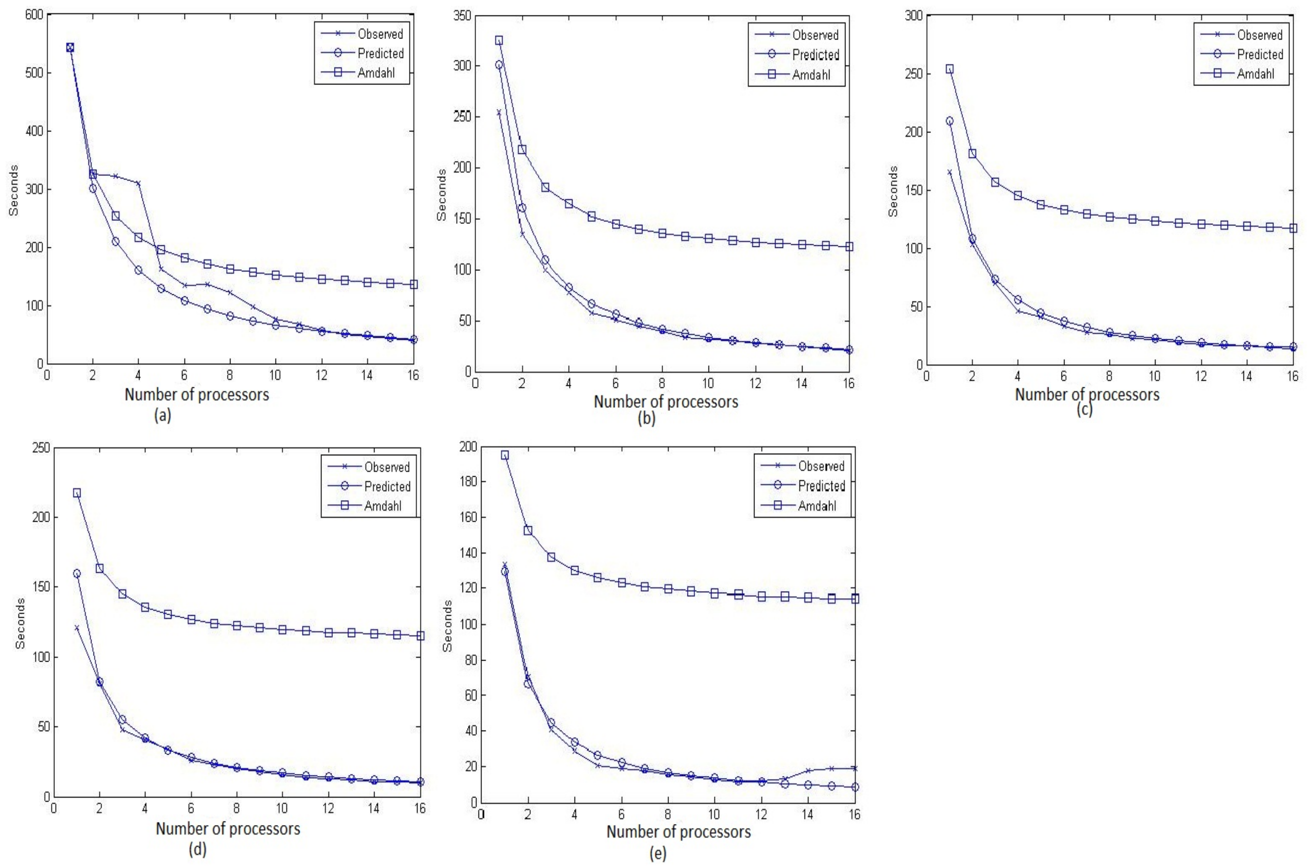

Fig. 17. Results of the predicted performance for the big array application. (a) Performance of the application using only one node. (b) Performance using two nodes. (c) Performance using three nodes. (d) Performance using four nodes. (e) Performance using five nodes