PERFORMANCE COMPARISON OF PARALLEL PROGRAMMING PARADIGMS ON A MULTICORE CLUSTER

Enzo Rucci, Franco Chichizola, Marcelo Naiouf and Armando De Giusti*
Institute of Research in Computer Science LIDI (III-LIDI)
National University of La Plata
La Plata, Buenos Aires, Argentina
{erucci,francoch,mnaiouf,degiusti}@lidi.info.unlp.edu,ar

ABSTRACT
Currently, most supercomputers are multicore clusters. This type of architectures is said to be hybrid, because they combine distributed memory with shared memory. Traditional parallel programming paradigms (message passing and shared memory) cannot be naturally adapted to the hardware features offered by these architectures. A parallel paradigm that combines message passing with shared memory is expected to better exploit them. Therefore, in this paper the performance of two parallel programming paradigms (message passing and combination of message passing with shared memory) is analyzed for multicore clusters. The study case used is the construction of phylogenetic trees by means of the Neighbor-Joining method. Finally, conclusions and future research lines are presented.

KEY WORDS
parallel programming, hybrid programming, multicore cluster, performance comparison, Neighbor-Joining method.

1 Introduction

The study of distributed and parallel systems is one of the most active research lines in computer science nowadays [1, 2]. In particular, the use of multiprocessor architectures configured in clusters, multiclusters, grids and clouds, supported by networks with different characteristics and topologies, has become general, not only for the development of parallel algorithms but also for the execution of processes that require intensive computation and the provision of concurrent web services [3, 4, 5, 6].

The energy consumption and heat generation problems that appear when the speed of a processor is escalated are the reasons of the development of the multicores processors. This type of processor is formed by the integration of two or more computational cores within the same chip. Even though these cores are simpler and slower, when combined they allow enhancing the global performance of the processor while making an efficient use of energy [7, 8].

The incorporation of this type of processors to conventional clusters gives birth to an architecture that combines shared and distributed memory, known as multicore cluster [9, 10]. In this type of architectures, communications between processing units are heterogeneous [11] and can be classified in two groups: inter-node and intra-node. Inter-node communications are between cores that are in different nodes and they communicate by exchanging messages through the interconnection network. Intra-node communications are between cores that are within the same node and they communicate through the different memory levels of the node.

Parallel programming paradigms differ in the way tasks communicate and synchronize. Traditionally, there were two main programming paradigms for parallel architectures: shared memory and message passing. In shared memory architectures, such as multicores, the most widely used paradigm is shared memory. In it, tasks communicate and synchronize by reading and writing variables in a shared address space. OpenMP is the most widely used library to program shared memory [12]. Message passing is the most commonly chosen paradigm for distributed architectures, such as traditional clusters. In it, each task has its own address space and task communication and synchronization is done by exchanging messages. MPI is the most widely used library to program under this paradigm [13].

Multicore clusters are hybrid architectures that combine distributed memory with shared memory, so neither of the previous paradigms naturally adapts to all hardware features in these architectures. For this reason, the scientific community has a great interest in analyzing hybrid parallel programming paradigms that allow communications both through message passing and shared memory. A parallel paradigm that uses shared memory for intra-node communications and message passing for inter-node communications would be expected to better leverage the features offered by multicore clusters [14].

In this paper, the performance of two parallel algorithms designed for the same application but using different programming paradigms is compared over a multicore cluster. The application selected as study case is the construction of phylogenetic trees by means of the Neighbor-Joining method, and it was selected based on its computational complexity ($O(n^3)$). The rest of the paper is organized as follows: in Section 2, related works are described.

*CONICET Main Researcher

In Section 3, the Neighbour-Joining method is explained, together with sequential an the parallel algorithms used. In Section 4, the experimental work carried out is described, whereas in Section 5, the results obtained are presented and analyzed. Finally, in Section 6 the conclusions and future lines of work are presentend.

2 Related Works

There are numerous works that analyze and compare parallel programming paradigms for multicore clusters. To mention but a few, in [15], [16] and [17] there are comparisons between message passing and combinations of message passing with shared memory. Results vary depending on the characteristics of the problem solved, the algorithms used and the features of the hardware architecture used as support; which makes research in this area even more significant.

3 Neighbor-Joining Method

Studies carried out on molecular mechanisms of organisms suggest that all organisms in the planet have a common predecessor. Then, any set of species would be related, which is called Phylogeny. In general, this relation can be represented by means of a phylogenetic tree. The task of Phylogeny is to infer the previous tree based on observations of the existing organisms [18].

The Neighbor-Joining method (NJ) is based on distance matrixes and is widely used by biotechnologists and molecular biologists due to its efficiency and temporal complexity order. In recent years, it has become very popular through its use in the ClustalW algorithm [19], one of the most widely used tools for multiple sequence alignment. The NJ algorithm was originally developed by Saitou and Nei in 1987 [20]. One year later, Studier and Keppler [21] would review the algorithm and incorporate an improvement that allows reducing the temporal complexity from $O(n^5)$ to $O(n^3)$.

The Neighbor-Joining method starts with a star-shaped tree. In each step, the pair of nodes that is closest to each other are selected and connected through a new, inner node. Then, the distances from this new node to the rest of the nodes in the tree are calculated. The algorithm ends when there are only two nodes that are not connected [18].

Algorithm 1 details the pseudo-code of the Neighbor-Joining method and Figure 1 shows an example of the building process of a phylogenetic tree using the Neighbor-Joining method for a 4x4 distance matrix.

3.1 Sequential Neghbor-Joining Algorithm

The algorithm starts with a distance matrix between pairs of sequences of $N \times N$ denotaded as d, N being the number of sequences. Since this is a symmetric matrix, there is no need to store it in full, but only the lower triangular matrix or the upper triangular matrix can be stored (in this case, the former is chosen).

Algorithm 1 Pseudo-code of the Neighbor-Joining method.

Initialization:

$L = S$.

Iteration:

Pick the pair i,j for which normalized distance $D_{i,j}$ is minimum, where

$$D_{i,j} = d_{i,j} - (r_i + r_j)$$

and r_i is the divergence of node i, where

$$r_i = \frac{1}{[L] - 2} \times \sum_{k \epsilon L} d_{i,k}$$

Define a new node k and assign $d_{k,s} = ½(d_{i,s} + d_{j,s} - d_{i,j})$, for all s in L.

Add k to S with distance edges $d_{i,k} = ½(d_{i,j} + r_i - r_j)$, $d_{j,k} = d_{i,j} - d_{i,k}$, connecting k to i and j, respectively.

Remove i and j from L and add k.

Termination:

When L is formed by two leaves i and j, add the pending edge between i and j with distance $d_{i,j}$.

In each iteration of the main loop, the pair i, j for which $D_{i,j}$ is minimal has to be found. The normalized distance matrix D is not stored, but the value of each position is calculated in each iteration. Node divergences are computed in a one-dimensional arrangement before starting the main loop, and are updated in each iteration of the loop, rather than calculating them every time they are required.

For the list of active nodes L, a one-dimensional flag arrangement is used; the flags indicate which nodes have been selected and which have not.

Algorithm 2 details the pseudo-code of the sequential algorithm.

3.2 Parallel Neighbor-Joining Algorithm

The algorithm was parallelized using different parallel programming paradigms (message passing and combination of message passing and shared memory). Before going into an in-depth review of the solution developed, certain aspects of the Neighbor-Joining algorithm should be analyzed, since these might explain why an efficient parallel solution is difficult to obtain.

First, it should be noted that for each iteration of the main loop, a new node is added to the distance matrix, but those from the two originating nodes are removed. This means that, in a parallel solution, the work carried out by each task in each iteration decreases as the iterations of the main loop progress. Also, in distributed environments, the distances of the new nodes must be distributed among all processes forming the solution.

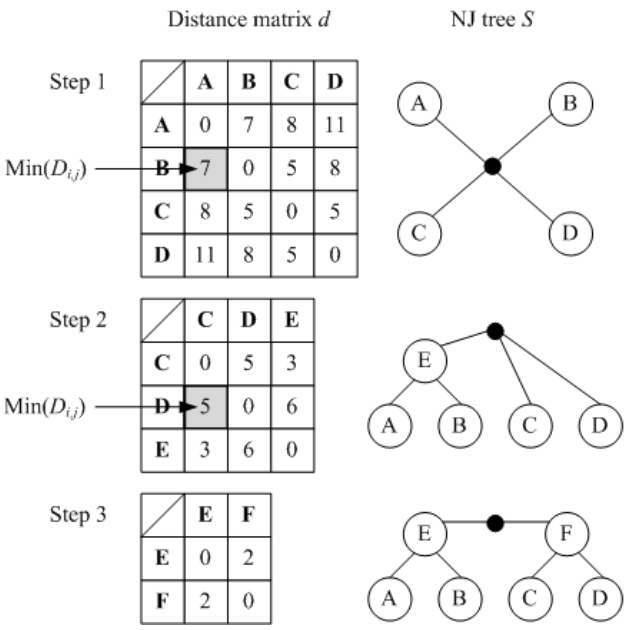

Figure 1. Building process of a phylogenetic tree using the Neighbor-Joining method for a 4x4 distance matrix.

The search for the pair of nodes whose distance is minimal represents the most expensive part of the main loop from a computational standpoint. Taking into account that the distance matrix is triangular, distributing the work required for the search process by assigning the same number of rows to each task could result into idle time, since not all rows have the same number of cells. Since idle time negatively affects the performance of an algorithm, it must be removed or, if this is not possible, minimized. For this reason, the workload distribution strategy must be chosen trying to make it as equitable as possible for all tasks.

3.2.1 Message Passing as Parallel Programming Paradigm

This solution uses a master-slave model of P processes as parallelization strategy. The distances of each newly created node are distributed among all processes following a circular order (the node to which it is assigned is the *owner*). Algorithm 3 details the pseudo-code of the parallel algorithm.

3.2.2 Combination of Message Passing and Shared Memory as Parallel Programming Paradigm

This solution is based on the one described in Section 3.2.1, but unlike it, each process generates T threads when computation begins. Then, the iterations belonging to different process loops are distributed among the threads that have been generated.

Algorithm 2 Pseudo-code of the sequential algorithm.

1. $L = S$.
2. foreach $d_{i,j}$ in d do
3. Update r_i.
4. Update r_j.
5. end foreach
6. for h in 1 to N-2
7. foreach $d_{i,j}$ in d do
8. Calculate $D_{i,j}$.
9. Calculate minimum $D_{i,j}$.
10. end foreach
11. Create node k connecting nodes i and j.
12. $S = S + \{k\}$.
13. Calculate $d_{i,k}$ and $d_{j,k}$.
14. $L = L - \{i,j\} + \{k\}$.
15. foreach s in L do
16. Calculate $d_{k,s}$.
17. Update r_k.
18. Update r_s.
19. end foreach
20. end for
21. Group both remaining nodes in L.

4 Experimental Work

4.1 Architecture Used

Tests were carried out on a cluster of Blade multicores with four blades and two quad core Intel Xeon e5405 2.0 GHz processors each[22, 23]. Each blade has 10 Gb RAM memory (shared between both processors) and 2 x 6Mb L2 cache for each pair of cores. The operating system is GNU/Linux Fedora 12 (64 bits).

4.2 Algorithms Used

The algorithms used in this work were developed using C language (gcc compiler version 4.4.2) with the OpenMPI (mpicc compiler version 1.4.3) library for message passing and OpenMP for thread management. The algorithms are detailed below:

Algorithm 3 Pseudo-code of the parallel Neighbor-Joining algorithm.

1. The master process divides distance matrix *d* into *P* portions and distributed *P*-1 among the slaves. Each process keeps approximately ((*N*)×(*N*-1)/*2P*) elements from distance matrix *d*.
2. Each process calculates the partial divergences of the nodes it has, broadcasts them to the other processes, and update its divergence vector based on the partial divergences received from the other processes.
3. for *h* in 1 to *N*-2 do
4. Each process calculates its local minimum $D_{i,j}$.
5. The master process collects all local minimums, calculates the global minimum $D_{i,j}$ and broadcasts it to the other processes.
6. The master process creates a new node *k* and adds it to *S*.
7. The owner process of node *k* calculates $d_{i,k}$and $d_{j,k}$. Each remaining process accumulates in a data structure the distances to the pair of nodes *i,j* it has and then sends it to the owner of node *k*.
8. The owner process of node *k* calculates the distances from it to the rest of the nodes and updates the divergence vector.
9. The owner process of node *k* broadcasts to the other processes the updated divergence vector.
10. Each process removes nodes *i* and *j* from their own *L* and add *k*.
11. end for

- *MP*: this algorithm is based on the solution described in Section 3.2.1, where *P* is the number of cores used.
- *HY*: this algorithm is based on the solution described in Section 3.2.2, where *P* is the number of blades used and *T* is the number of cores in each blade.

4.3 Tests Carried Out

Based on the features of the architecture, both algorithms were tested using all the cores with different numbers of nodes: two, three and four; this means that *P* = {16, 24, 32} for *MP*. In the case of *HY*, one process per node was used; this means that *P* = {2, 3, 4} and *T* = {8}. Various problem sizes were used: *N* = {4000, 6000, 8000, 10000, 12000, 14000, 16000}. Each particular test was run five times, and the average execution time was calculated for each of them.

5 Results

To assess the behavior of the algorithms developed when escalating the problem and/or the architecture, the *speedup* and *efficiency* of the tests carried out are analyzed [1, 3, 24].

The speedup metric is used to analyze the algorithm performance in the parallel architecture as indicated in Equation (1).

$$Speedup = \frac{SequentialTime}{ParallelTime} \quad (1)$$

To assess how good the speedup obtained is, the efficiency metric is calculated. Equation (2) indicates how to calculate this metric, where *p* is the total number of cores used.

$$Efficiency = \frac{Speedup}{p} \quad (2)$$

Figure 2 shows the efficiency achieved by the algorithms *MP* and *HY* when using two, three and four blades of the architecture for different problem sizes (*N*). For readability, only the results for *N* = {4000, 8000, 12000, 16000} are shown.

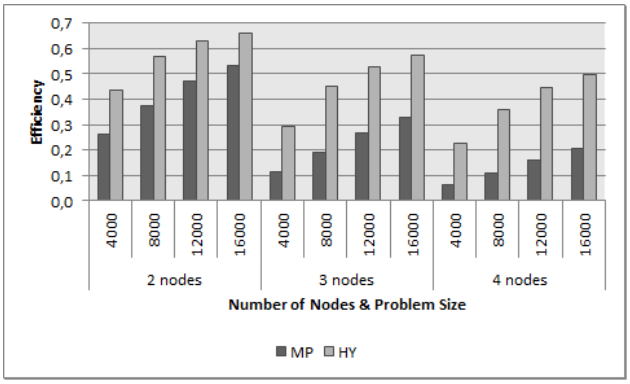

Figure 2. Efficiency achieved by algorithms *MP* and *HY* when using two, three and four blades of the architecture for different problem sizes (*N*).

This chart shows that both algorithms increase their efficiency as the size of the problem increases and, on the other hand, as it is to be expected in most parallel systems, the efficiency decreases when the total number of nodes used increases. The efficiency levels obtained with both algorithms are low due to the number of communication and synchronization operations carried out and the idle time that processes and threads might have. Despite this, it can be seen that the best efficiency levels are obtained by *HY*.

The superiority of *HY* over *MP* can be analyzed in detail in Figure 3, which shows the percentage of the relative difference between the efficiencies of both algorithms (*prd*), calculated by means of Equation (3).

$$prd = \frac{efficiency(HY) - efficiency(MP)}{efficiency(MP)} \times 100 \quad (3)$$

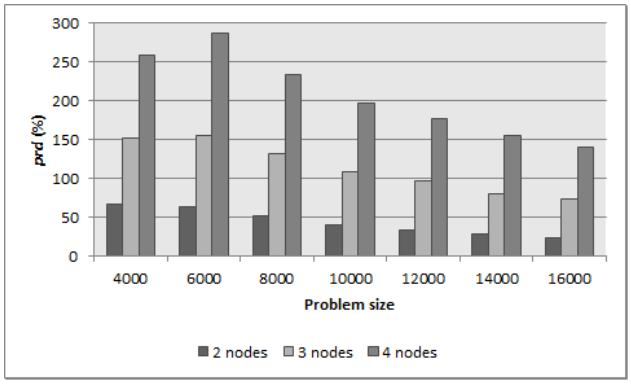

Figure 3. *prd* with two, three and four blades of the architecture for different problem sizes (*N*).

The chart in Figure 3 shows that *HY* is better than *MP* in all cases, achieving improvement percentages higher than 250%. It can also be seen that the improvement increases with the number of nodes and that it decreases when the size if the problem increases. The difference in favor of *HY* is due to several factors:

- First, *HY* reduces latency and maximizes the bandwidth of the interconnection network, since, by using a single, multi-threaded process instead of multiple processes for each node, it groups all task messages corresponding to a node in a single, larger message. It also removes competition for the network at node level.
- Finally, since the distance matrix *d* is divided in less parts and the work assigned to each of these portions is distributed dynamically among the threads in each process, *HY* achieves a more balanced work distribution versus the fully static strategy used by *MP*.

6 Conclusions and Future Works

In this paper, the performance of two parallel programming paradigms (message passing and hybrid) was compared for current cluster architectures, taking as study case the construction of phylogenetic trees by means of the Neighbor-Joining method. The algorithms were tested using various work and architecture sizes. The results obtained show that the hybrid parallelization better leverages the hardware features offered by the support architecture, which in turn yields a better performance.

Future lines of work include the development and optimization of hybrid solutions for other types of applications and their comparison with solutions based only on message passing or shared memory.

References

[1] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *An Introduction to Parallel Computing. Design and Analysis of Algorithms.*, 2nd ed. Addison Wesley, 2003.

[2] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, 2nd ed. Addison Wesley, 2006.

[3] Z. Juhász, P. Kacsuk, and D. Kranzlmuller, Eds., *Distributed and Parallel Systems: Cluster and Grid Computing.* Springer Science + Business Media Inc., 2005.

[4] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, *The Sourcebook of Parallel Computing*. Morgan Kauffman, 2003.

[5] M. D. Stefano, *Distributed data management for Grid Computing*. John Wiley & Sons Inc, 2005.

[6] M. Miller, *Web-Based applications that change the way you work and collaborate online*. Que, 2009.

[7] AMD. (2009) Evolución de la tecnología de múltiple núcleo. [Online]. Available: http://multicore. amd.com/es-ES/AMD-Multi-Core/resources/Technology-Evolution

[8] T. Burger. Intel Multi-Core Processors: Quick Reference Guide.

[9] L. Chai, Q. Gago, and D. K. Panda, "Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid.*, 2007.

[10] M. McCool, "Scalable programming models for massively multicore processors," in *Proceeding of the IEEE*, 2012.

[11] C. Zhang, X. Yuan, and A. Srinivasan, "Processor affinity and mpi performance on smp-cmp clusters," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 2010.

[12] (2012) OpenMP.org. [Online]. Available: http://openmp.org/wp/

[13] (2012) The Message Passing Interface (MPI) standard. [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/

[14] N. Drosinos and N. Koziris, "Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters," in *18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Papers*, 2004.

[15] E. Rucci, A. De Giusti, F. Chichizola, M. Naiouf, and L. De Giusti, "DNA sequence alignment: hybrid parallel programming on a multicore cluster," in *Recent Advances in Computers, Communications, Applied Social Science and Mathematics*, N. Mastorakis, V. Mladenov, B. Lepadatescu, H. R. Karimi, and

C. G. Helmis, Eds., vol. 1, no. 1. WSEAS Press, September 2011, pp. 183–190.

[16] F. Leibovich, L. De Giusti, and M. Naiouf, "Parallel algorithms on clusters of multicores: Comparing message passing vs hybrid programming," in *Proceedings of the 2011 International Conference on Parallel and Distributed processing Techniques and Applications (PDPTA2011)*, 2011.

[17] X. Wu and V. Taylor, "Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-scale Multicore Clusters," *The Computer Journal*, vol. 55, pp. 154–167, 2012.

[18] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchinson, *Biological Sequence Analysis. Probabilistic models of proteins and nucleic acids.*, 7th ed. Cambridge University Press, 2002.

[19] ClustalW and Clustal X Multiple Sequence Alignment. [Online]. Available: http://www.clustal.org/clustal2/

[20] N. Saitou and M. Nei, "The neighbor-joining method: a new method for reconstructing phylogenetic trees," *Mol. Biol. Evol.*, vol. 4, pp. 406–425, 1987.

[21] J. A. Studier and K. J. Keppler, "A note on the neighbour-joining algorithm of saitou and nei," *Mol. Biol. Evol.*, vol. 5, pp. 729–731, 1988.

[22] HP. HP BladeSystem. [Online]. Available: http://h18004.www1.hp.com/products/blades/components/c-class.html

[23] ——. HP BladeSystem c-Class architecture. [Online]. Available: http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00810839/c00810839.pdf

[24] B. Wilkinson and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers.*, 2nd ed. Prentice Hall, 2005.