Parallel Processing Puzzle N^2-1 on Cluster Architectures Performance Analysis

Victoria Sanz[1], Armando De Giusti[2], Franco Chichizola[3], Marcelo Naiouf[4], and Laura De Giusti[5]
Instituto de Investigación en Informática (III-LIDI) –School of Computer Sciences – UNLP
vsanz@ciudad.com.ar,{degiusti, francoch, mnaiouf, ldgiust}@lidi.info.unlp.edu.ar

Abstract. *An analysis of a parallel solution of N^2-1 Puzzle using clusters, is presented. This problem is interesting due to its complexity and related applications, particularly in the field of robotics.*

A variation of classic heuristics for forecasting the work to be done in order to reach a solution is analyzed, and it is shown that its use significantly improves the time of sequential algorithm A^*.

Then, a parallel solution on a distributed architecture is presented and speedup is analyzed based on the number of processors, efficiency, and the possible superlinearity when scaling the problem.

Keywords. Parallel Algorithms, Distributed Processing, Speedup, Superlinearity, Efficiency Scalability.

1. Introduction

Discrete optimization problems encompass a wide variety of areas [24] and are often resolved by means of state-space exploration methods by looking for a "solution" state [13].

These search techniques usually have a high computational cost, and, in many cases, a thorough analysis of the solutions space is impossible, which creates a need for heuristics that approach an optimal solution [8][17].

Complexity and computation time favor the development of parallel algorithms for discrete optimization problems and, in particular, graph processing techniques representing the problem have been of great interest [5][21].

This is the case of BFS (Best First Search) methods, which start from a node of the graph representing the problem and apply some type of work assessment metrics to reach a solution, so as to evolve from an initial state of the graph towards the "optimum solution" state.

The natural parallelization of the technique consists in starting the evolution from different "possible" nodes on the different processors of the multiprocessor architecture. As the algorithm evolves, processors need to be communicated to inform the results achieved or discard partial solutions based on the type of metrics selected [19][12].

Some of the aspects observed when using cluster-type parallel architectures for the resolution of discrete optimization problems [1] are of interest:

- Parallelization granularity is critical, since it will determine the improvement in the time required to find the solution and communications overhead.
- In general, load balancing needs to be dynamic (which calls for communication) because the task to be done is of an exploratory nature and very hard to predict in advance [2].

Usually, in parallel processing, the first point of interest in the resolution of an algorithm on a multiprocessor architecture is the speedup factor (*Sp*), which is a relative performance measure defined as the ratio between the execution time of the best sequential algorithm on a monoprocessor computer and the execution time of the corresponding parallel algorithm on a multiprocessor computer [7][14]. If *Ts* is the sequential execution time and *Tp* is the parallel execution time, then we have *Sp = Ts/Tp*, which is usually maximized as much as possible in the development of parallel applications. Speedup is limited by the maximum concurrence time that can be obtained from the application, by the unavoidable sequential component of the

[1]Student Research Fellow. Teaching Assistant.
[2]CONICET Main Researcher. Full-Time Chair Professor.
[3]CONICET PhD Scholar. Co-Chair Professor.
[4]Full-Time Chair Professor.
[5]Advanced Scholar UNLP. Co-Chair Professor.
This research is partly financed by CIC and YPF Foundation.

algorithm, and by the number of processors (*N*) available for the execution. [18].

A second significant parameter when analyzing parallel applications is the efficiency (*E*) achieved. The efficiency is defined as the ratio between speedup and the number of processors used to obtain it: *E = Sp/N*. This definition yields efficiency values between *0* and *1*. The closer to *1* the value, the closer speedup is to the optimum *N*. The efficiency parameter is a quality and cost metrics for the parallel algorithm that is particularly significant and cannot always be maintained when escalating problems, increasing the number of processors, or carrying the algorithm to a different multiprocessor architecture [3].

Scalability is a very important factor in parallel applications: problems usually "escalate", i.e., the volume of work to be done increases, and the multiprocessor architectures used can also "escalate" by increasing the number of processors used. The effect of escalating workload and/or processors on the performance of parallel algorithms, considering speedup and efficiency [11], is of interest.

The maximum theoretical speedup can in some cases be improved, which is known as superlinearity (*Su*). It is interesting analyzing why speedup can surpass ***N***, particularly with discrete optimization problems solved in parallel: the exploration of the total space of possible solutions can be reduced by distributing the workload between ***N*** processors so as to "cut down" or "finish" the global search by reaching the expected result in any of these processors [10][15]. That is, in theory, the cluster architecture will allow superlinearity depending on workload balancing, processor heterogeneity, and the processing time/communication time ratio of the algorithm used [22].

If distributed architectures that are even more weakly coupled are used (such as miniclusters or grids), the relation between processing time and communication time will impose a limit on the possibility of achieving superlinearity [25].

1.1. Contribution

The parallel processing on clusters for solving the N^2-1 Puzzle, a complex NP discrete optimization problem that is of special interest due to the possibility of achieving superlinearity [23] and for its application in robot motion planning on graphs problems [6][16], is studied.

The contributions of this paper are:

- Incorporation to the algorithm presented in [23] of an a priori detection feature that determines the solvability of an initial given configuration.
- Analysis of the implementation of a variant (*MDLC)* of the work prediction heuristic that combines Manhattan Distance *(MD)* with the detection of linear conflicts and allows a very significant improvement in the algorithm's response time, both sequentially and in parallel [9].
- Presentation of a series of experimental results on boards of different dimensions and clusters of 4, 6, 8, 12, and 16 processors, analyzing performance (speedup, efficiency, superlinearity) and a local work parameter (*LW*) that may affect load balancing.

2. Characterization of the N^2-1 Puzzle problem

The N^2-1 Puzzle problem is a generalization of the 15 puzzle problem devised by Sam Lloyd [20]. It consists of N^2-1 pieces numbered from *1* to *N^2-1* placed on an N^2-sized board. N^2-1 squares of the board have exactly one piece, and only one of the squares –called "hole"– is empty.

The purpose of the puzzle is to repeatedly fill in the hole with one of the pieces adjacent to it (horizontally or vertically) until reaching a state where the square *(i,j)* is occupied by the piece numbered as *(i-1)*N + j*, and the square *(N,N)* is the hole.

The solution to the problem posed should be the one that minimizes the number of movements required to achieve the final configuration from the initial given configuration.

Fig. 1 shows an N^2-1 Puzzle and the solution board used with N = 4.

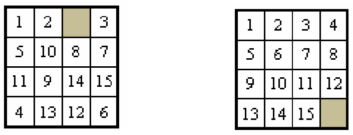


Figure 1. Initial and final board of 15 Puzzle.

2.1. Solvable and unsolvable cases.

It is not always possible to get to the final board from a given initial distribution. This is because only half of the states can be reached from any other state, and therefore the search space for the Puzzle N^2-1 problem is reduced to $N^2!/2$.

The procedure to corroborate if an initial board can be solved is the following:

▪ For each piece *i* (*i* = 2..N^2-1), the number of pieces with a lower number that appear after the piece in question – whether on the same row to the right or on any lower row – are counted. We will call this the *i* inversion number and will denote it as *ni*.

▪ Then, NT = *n2 + n3 +…+ n(N^2-1) + e* is calculated, where e is the number corresponding to the row containing the hole, for both the initial and final boards.

▪ If the parity of both results is the same, then the final board can be reached from the initial board. This is because *(N mod 2)* does not vary with any legal movement.

In the previous example, for Fig. 1, *NT = 28 and 4*. Therefore, the board represented in Fig. 1 can be solved.

2.2. Manhattan distance *(MD)*

Let's assume that each position of the board is represented as an ordered pair. The distance between positions *(i,j)* and *(k,l)* is defined as $|i-k|+|j-l|$. This distance is known as Manhattan distance. The addition of Manhattan distances between board *x* and final board positions will be a minimum estimator of the number of movements required to transform board *x* into the solution board.

2.3. Linear conflicts

A more polished heuristic would have the algorithm process a lower amount of nodes, thus reducing total search time.

Let's assume that two pieces, *x* and *y*, are positioned in the correct row, but inverted: this is called *linear conflict*. One of them will have to change rows for the other to be able to move into its final position and, after moving across the columns as its *MD* indicates, the piece will have to return to its target row. The same can be applied to columns. Thus, for each linear conflict, *2* additional movements could be added to the board's Manhattan distance.

Any given piece could be part of one linear conflict per row and another one per column at the most. This restriction prevents that the heuristic overestimates the real cost, which would cause the function to stop being admissible.

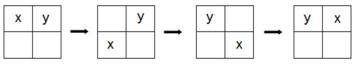


Figure 2. Sequence of movements in case of a linear conflict in a row

Fig. 2 shows this situation in a *2x2* board. Let's assume that *x > y*, and that both pieces are on the correct row. Note that Manhattan distance is 1 for both pieces.

2.4. Manhattan distance and linear conflict detection (MDCL)

The work prediction heuristic analyzed here is based on Hanson's proposition [9] and combines Manhattan distance with the detection of linear conflicts, being defined as: *MDLC (t) = MD(t) + LC(t)*

Where *t* represents the current board, *MD* is the function that calculates Manhattan distance for *t*, and *LC(t) = 2*(amount of linear conflicts in t)*.

3. Sequential solution using MD and MDLC heuristics

A* is one of the variants of the Best First Search technique [21]. Each node *n* is assessed based on the cost of reaching it from the root of the search tree (*g(n)*) and a heuristic that estimates the cost to go from *n* to a solution node (*h(n)*). Thus, the cost function will be *L(x) =g(x)+h(x)*. Algorithm A* always ensures the best solution.

This algorithm keeps a list of unexplored nodes (*open list*) ordered by the value of function *L*, and a second list of already explored nodes (*closed list*) used to avoid loops in the search graph. Initially, the open list contains only one element, the initial node, and the closed list is empty.

After each step, the node with the lowest *L* value (the *best node*) is removed from the open list and examined. If the node is the solution, the algorithm ends. If it is not, the node is expanded and added to the closed list. Each successive node is added to the open list only if it does not appear on the closed list, or if it does but its value *L* is lower than that of the previous node.

Experimental tests were carried out with the two heuristic functions (MD and MDLC) presented before. Results are shown in Table 1 and Fig. 4. It can be clearly seen that the use of the MDLC heuristic allows a noticeable improvement in the response time of sequential algorithm A* to find an "optimal" solution for the Puzzle N^2-1 problem.

Based on these results, the MDLC heuristic is adopted for the parallel processing experiment to be carried out on a cluster.

```
Create open list.
Create closed list.
Insert (open list, x, h(x)).
// h(x) is the heuristic function
while (non-empty open list) and (did not find a solution)
        // Extract the minimum node of the open list, let's call it n
        n = RemoveMinimum (open list)
        // If h(n) = 0 terminates the algorithm, if not n is expanded.
        if (IsSolution(n))
                solution = n
        else
                children = Expand(n)
                Insert(closed list,n)
               //A node is acceptable if it is not on the closed list, or if it
                is but with a cost that is higher than the current cost
                for each child of n
                   if (IsAcceptable(child))
                            Insert(open list, child, h(child) + g(n) + 1)
Return solution.
```

Figure 3. Sequential algorithm.

Table 1. Reduction % of time with MDCL.

Test number	Solution depth	Board Size	Reduction % of time with DMLC
1	38	4x4	63
2	39	4x4	89
3	40	4x4	80
4	42	4x4	98
5	43	4x4	99
6	34	5x5	84
7	34	5x5	95
8	36	5x5	90
9	38	5x5	82
10	39	5x5	85
11	40	6x6	96
12	42	6x6	99
13	44	6x6	95

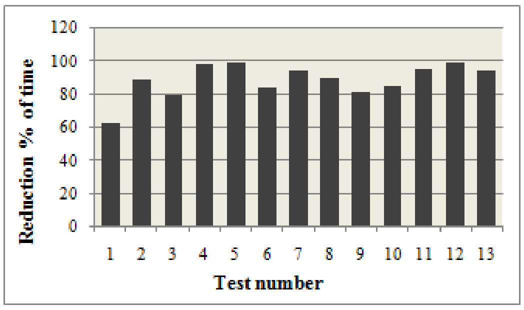


Figure 4. Reduction % of time with MDCL.

4. Outline of the parallel solution on cluster.

The parallelization strategy consists in keeping local open and closed lists on each processor. At the beginning, only one of the processors will work with the initial node. As other nodes are generated, processors will receive them and start working.

The proposed parallel solution does not require a central process. All processors will search locally, building their own closed list – to avoid locally repeated work – as well as their open list. Processors should communicate among them the minimum values of the solutions found in order to minimize unnecessary searches.

For the implementation of the parallel algorithm, the "Asynchronous Round Robin" (ARR) distributed load balancing technique and the modified "Dijkstra's Termination Algorithm" are used [4].

Each of the *p* "worker" processes will have a value indicating the cost of the best solution found so far (*BSC*), which will be used to limit the search process. The initial board is assigned to worker *0*, which is also in charge of detecting the end of the search.

A process that has some work pending on its open list will process at the most a fixed amount of nodes for each iteration (local work parameter, *LW*) or will process nodes until it finds a solution or until its open list is empty. Then, the worker receives the costs of the "best solutions" – if there are any – found so far by the other workers and updates its BSC variable as needed. Thus, the nodes to process will be only those whose cost is lower than BSC.

If the process still has some work pending on its open list, it checks if there are any work requests from other processes, and if there are, it sends the first and last nodes of its open list to the requesting idle processor. It then continues working with its nodes.

If the process does not have any pending work, it will be idle, so it will send a work request to its donor following the ARR algorithm. If the process found a new solution, it sends the corresponding cost to the other processes. It then waits for the following types of messages, which will be processed with no particular order of priority:

- Work request: an idle worker selected this process as its donor.
- Work: the donor sends the requested work. The process is active again.
- Rejection of work request: the selected donor does not have any work. The process must send a work request message to the next donor.
- Token: reception of the token for termination detection. If necessary, the token is updated and the next process begins. Processor 0, upon receiving the token, checks termination.
- New solution found by other worker: if necessary, the BSC variable is updated.

When process 0 detects the termination, it sends a message to the other processes to inform the end of the computation.

The termination token is used to translate the minimum cost solution movements to process 0, so that the messages communicating new solutions found with the algorithm only have an integer number, the cost, to avoid communication overhead.

5. Experimental results with the MDLC heuristic.

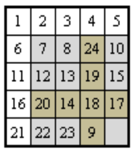
To carry out the tests, a homogeneous cluster composed by 20 Pentium 4, 2.4GHz and 1GB RAM processors was used.

To study the performance of the parallel algorithm developed, tests for different initial states were carried for with *4x4*, *5x5* and *6x6* boards, in all cases using *LW= 250, 500, 750,* and *1000*.

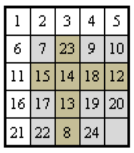
To analyze the behavior of the application upon escalation regarding the architecture, each board was tested with subsets of *P* processors belonging to the cluster described above, where *P = 4, 6, 8, 12 and 16*.

To observe how the algorithm escalates as the size of the problem increases (bigger boards), two types of initial configuration were defined:

- Configuration 1: inversion of the third column and then the third row in the bottom right *4x4* sub-board.
- Configuration 2: inversion of the second column and then the second row in the bottom right *4x4* sub-board.



1	2	3	4	5
6	7	8	24	10
11	12	13	19	15
16	20	14	18	17
21	22	23	9	



1	2	3	4	5
6	7	23	9	10
11	15	14	18	12
16	17	13	19	20
21	22	8	24	

Figure 5. Examples for 5x5 boards. Conf. 1 (left) and Conf. 2 (right).

Table 2. Speedup, efficiency and optimum LW for tests with configuration 1.

Board	Number of Processors	LW	Speedup	Efficiency
4x4	4	1000	11,44	2,86
	6	1000	12,69	2,12
	8	1000	14,07	1,76
	12	750	25,42	2,12
	16	1000	60,58	3,79
5x5	4	500	15,94	3,99
	6	750	22,37	3,73
	8	750	39,02	4,88
	12	750	51,25	4,27
	16	1000	52,78	3,30
6x6	4	1000	6,70	1,67
	6	1000	13,62	2,27
	8	1000	16,99	2,12
	12	1000	25,10	2,09
	16	1000	32,68	2,04

Table 3. Speedup, efficiency and optimum LW for tests with configuration 2.

Board	Number of Processors	LW	Speedup	Efficiency
4x4	4	1000	13,49	3,37
	6	1000	25,21	4,20
	8	1000	34,75	4,34
	12	1000	52,82	4,40
	16	500	74,52	4,66
5x5	4	250	18,06	4,52
	6	250	36,44	6,07
	8	250	54,11	6,76
	12	250	81,84	6,82
	16	500	87,2	5,45
6x6	4	500	10,16	2,54
	6	500	16,90	2,82
	8	500	22,26	2,78
	12	500	29,26	2,44
	16	500	41,00	2,56

Tables 2 and 3 show the results obtained during the tests with configurations 1 and 2, respectively, for the different board sizes and different numbers of processors. For each case, the *LW* value that optimizes the final time of the parallel algorithm is shown.

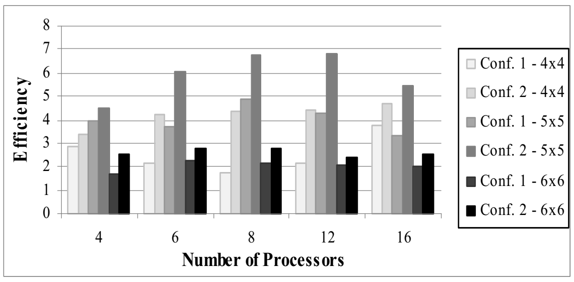


Figure 6. Efficiency of tests.

Fig. 6 is a summary chart of the efficiency achieved during the tests, clearly showing when superlinearity is obtained.

We performed several test for different initial states and a similar behavior has been observed as the one shown in the Table 2, Table3 and Figure 6. The full set of result is in [22].

6. Conclusions and future lines of work

An analysis of the parallel solution for the Puzzle N^2-1 problem on clusters has been presented, incorporating the MDLC heuristic that considers Manhattan distance and linear conflicts for estimating the amount of work to be done from a graph node in order to achieve a solution, so as to discard alternatives that cannot compete in the function to optimize (number of steps to reach the solution).

The advantages of the MDLC heuristics as compared to Manhattan distance has been analyzed, both for the sequential and the parallel algorithms, and speedup, efficiency, and superlinearity have been studied for different configurations of the cluster architecture and different dimensions and initial states of the problem.

It was observed that there is no optimum *LW* for all tests, but that it rather depends on different factors. A future research line is the study of the relationship between these factors: board size, initial disorder, and number of processors, in order to assess the optimum *LW* value a priori for each particular test.

Current research activities are focused on the generalization of the Puzzle N^2-1 problem so as to apply this to the movement of robots, particularly multi-robots with multiple objectives. The migration of the parallel algorithm to multi-cluster and grid is also being analyzed.

7. References

[1] Anderson T., Culler D., Patterson D. A Case for NOW (Networks of Workstations). IEEE Micro 1995; 15(1): pp. 54-64.

[2] Bohn C., Lamont G. Load Balancing for Heterogeneous Clusters of PCs. Future Generation Computer Systems, 2002; 18(3): 389-400.

[3] Buyya R. High Performance Cluster Computing: Architectures and Systems. Prentice-Hall; 1999.

[4] Dijkstra E., Scholten C. Termination detection for diffusing computations. Information Processing Letters 1980; 11(1):1-4.

[5] Ferreira A., Pardalos P. Solving Combinatorial Optimization Problems in Parallel: Methods and Techniques. New York: Springer; 1996.

[6] Fitch R., Butler Z., Rus D. Reconfiguration Planning Among Obstacles for Heterogeneous Self-Reconfiguring Robots. In: Proc. of the IEEE International Conference on Robotics and Automation; 2005. p. 117-124.

[7] Grama A., Gupta A., Karypis G., Kumar V. An Introduction to Parallel Computing. Design and Analysis of Algorithms. Pearson Addison Wesley; 2003.

[8] Grama A., Kumar V. State of the art in parallel search techniques for discrete optimization problems. IEEE Trans. on Knowledge and Data Engineering, 1999; 11(1): 28-35.

[9] Hanson O., Mayer A., Yung M. Criticizing Solutions to Relaxed Model Yields Powerful Admissible Heuristics. Information Sciences 1992; 63(3): 207-227.

[10] Helmbold D., McDowell C. Modeling speedup (n) greater than n. IEEE Trans. on Parallel and Distributed Systems, 1990; 1(2): 250-256.

[11] Hwang K. Advanced Computer Architecture. Parallelism, Scalability, Programmability. McGraw Hill; 1993.

[12] Korf R. Large-scale parallel breadth-first search. In: Proc. of the 20th National Conference on Artificial Intelligence; Pittsburgh, USA; 2005. p. 1380-1385.

[13] Lambur H, Shaw B. Parallel State Space Searching Algorithms. 2004. www.metablake.com/parallel_search_project.

[14] Leopold C. Parallel and distributed computing. A survey of models, paradigms, and approaches. New York: Wiley; 2001.

[15] Manquinho V., Marques-Silva J. Search Pruning Techniques in SAT-Branch-and-Bound Algorithms for Binate Covering Problem. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2002; 21(5): 505-516.

[16] Papadimitriou C., Raghavan P., Sudan M., Tamaki H. Motion Planning on a Graph. In: Proc. of the 35th Annual Symposium on Foundations of Computer Science; 1994. p. 511-520.

[17] Parberry I. A Real Time Algorithm for the (n^2-1) Puzzle. Information Processing Letters 1995; 56(1): 23-28.

[18] Quiin M. J. Parallel Computing: Theory and Practice. McGraw-Hill Companies; 1993.

[19] Rao V. N., Kumar V. On the efficiency of parallel backtracking. IEEE Trans. on Parallel and Distributed System, 1993; 4(4): 427-437.

[20] Ratner D., Warmuth M. The (n^2−1)-puzzle and related relocation problems. Journal for Symbolic Computation 1990; 10(2):11–137.

[21] Reinefeld A. Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in I D A*. In: Proc. of the 13th International Joint Conference on Artificial Intelligence; Chambery Savoi, France; 1993. p. 248-253.

[22] Sanz V. Paralelización de N-Puzzle. Technical Report 2007.

[23] Sanz V., Chichizola F., Naiouf M., De Giusti L., De Giusti A. Superlinealidad sobre Clusters. Análisis experimental en el problema del Puzzle N^2-1. In: Proc. of the XIII Congreso Argentino de Ciencias de la Computación; Chaco-Corrientes, Argentina; 2007. p. 1300-1309.

[24] Sergienko I., Shylo V. Problems of discrete optimization: Challenges and main approaches to solve them. New York: Springer; 2006; 42(4): 465-482.

[25] Wilkinson B., Allien M. Parallel Programming: Techniqus and Applications Using Network Workstation and Parallel Computers. Pearson Prentice Hall; 2005.