Proc. of the Intl. Conf. on Advances in Computing and Information Technology-- ACIT 2014
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-8859-9

A Parallel Implementation for Graph Partitioning Heuristics

[Leonardo Rogerio Binda da Silva, Roney Pignaton da Silva]

***Abstract*—The Graph Partitioning Problem (GPP) has several practical applications in many areas, such as design of VLSI (Very-large-scale integration) circuits, solution of numerical methods for simulation problems that include factorization of sparse matrix and partitioning of finite elements meshes for parallel programming applications, between others. The GPP tends to be NP-hard and optimal solutions for solving them are infeasible when the number of vertices of the graph is very large. There has been an increased used of heuristic and metaheuristic algorithms to solve the PPG to get good results where exceptional results are not obtainable by practical means. This article proposes an efficient parallel solution to the GPP problem based on the implementation of existing heuristics in a computational cluster. The proposed solution improves the execution time and, by introducing some random features into the original heuristics, improve the quality of the created partitions.**

***Keywords*— graph partitioning; parallel computing; grasp algorithms, heuristics**

I. Introduction

The Graph Partitioning Problem (GPP) is defined as follows: given a graph $G = (V, E)$, where V is the set of vertices (or nodes) with their assigned weights and E is the set of edges with their assigned weights, we must find k (where k is a positive integer) subsets N_1, N_2, ..., N_k such that: (a) all vertices of the original graph are distributed among the subsets and those created subsets are disjoints; (b) the sum of the weights of the vertices in each subset is approximately equal to the weight of all vertices divided by the number of subsets k and (c) the cut size, which is the sum of the weights of the edges between the among to be minimized [3].

The GPP-k is the problem of finding k ($k > 1$) subsets of vertices with the lowest possible cut size. In particular, for $k = 2$, the GPP finds a bisection. A very common way of solving this problem is finding bisections recursively [14, 16].

The GPP has several practical applications, such as design of VLSI (Very-large-scale integration) circuits, factorization of sparse matrix and partitioning of meshes of finite element for parallel programming applications [14,16]. The GPP tends to be NP-hard [17]. Optimal solutions for solving them are infeasible when the number of vertices of the graph is very large. There has been an increased use of heuristic and metaheuristic algorithms to solve the PPG to get good results where exceptional results are not obtainable by practical means [16]. R. S. Bonato [4] proposed four heuristic algorithms for the GPP using serial algorithms.

The present work proposes an efficient parallel solution to the GPP problem based on the implementation of existing heuristics in a computational cluster. The proposed solution improves the general performance of the heuristics presented in [4] in two aspects: 1) improvement of the execution time, with a considerable speedup related to the serial solution and, 2) improvement of the quality of the created partitions, by introducing some random features into the original heuristics.

II. Performance Metrics

As mentioned, the performance analysis is based on two metrics. First, we consider the cut size of the partitioned graph. In graph theory, a cut is a partition of the vertices of a graph into two disjoint subsets that are joined by at least one edge. The cut size is the number of edges whose end points are in different subsets of the partition. So, the GPP applied in a parallel computation problem could be used to divide the processing load to each processing node in such way that minimizing cut size should represent minimizing the message passing between processing nodes.

The second metric is the speedup. The speedup refers to how much a parallel algorithm is faster than a corresponding serial algorithm.

Speedup is defined by the following formula:

$$S_p = T_1/T_p \quad (1)$$

where: p is the number of processors, T_1 is the execution time of the serial algorithm and T_p is the execution time of the parallel algorithm with p processors.

Speedup measures can be used to provide an estimate for how well a code sped up if it was parallelized, and also 1) to generate a plot of time vs. processing nodes to understand the behavior of the parallelized code; 2) to see how the parallel efficiency tends toward the point of diminishing returns. With this information, you would be able to determine, for a fixed problem size, what is the optimal number of workers to use.

III. Graph Partitioning Algorithms

A great number of Graph Partitioning Algorithms has been proposed in the computing science literature in recent years, appearing in different fields of application. This section gives a general description of those algorithms by describing some general methods of solution that are used to categorize them.

- Geometrics Methods: The graph partitioning using geometric methods are based only on the information of the coordinates of each vertex of the graph. Therefore, there is not the concept of cutting edges, but these methods minimize metrics such as the number of vertices of the graph which are adjacent to other non-local nodes (size of the border). They tend to be faster than spectral methods, but return partitions with the worst cuts. Examples are Recursive

SEEK DIGITAL LIBRARY

Proc. of the Intl. Conf. on Advances in Computing and Information Technology-- ACIT 2014
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-8859-9

Coordinate Bisection (RCB) and Recursive Inertial Dissection (IBR) [1, 2, 3, 6, 7].

- Spectral Methods: Spectral methods do not make partitions dealing with the graph itself, but with its mathematical representation. The graphs are modeled by relaxation of the problem of optimizing a discrete quadratic function, being transformed into a continuous function. The minimization of the relaxed problem is then solved by calculating the second auto-vector of the Laplacian discretization of the graph. It generates good partitions for a wide class of problems. Examples are Recursive Spectral Bisection (RSB) and Multilevel Recursive Spectral Bisection (RSB Multilevel) [1, 6, 8, 9].

- Combinatorial Methods: Combinatorial methods receive as input a bisection of the graph and try to diminish the cut edges by applying a local search method. Examples are combinatorial methods of the Kernighan-Lin (KL) and Fidducia and Mattheyeses (FM). Both methods attempt to exchange vertices between the partitions in an try to reduce the cut, with the difference that the method KL exchanges pairs of vertices, while the FM method exchanges just a vertex, alternating between the vertices of each partition [3, 5, 10, 11].

- Multilevel Methods: The multilevel partitioning methods consist of three phases: contraction, partitioning and expansion of the graph. In the contraction phase, a number of graphs is constructed by joining the vertices to form a lower graph. This newly built contracted graph is contracted again, and so on until a graph small enough is found. A bisection of this small graph is made in a very fast way, since the graph is small. During the expansion phase, a refining method is applied to each level of the graph as it is expanded. Multilevel methods are present in many packages of software such as Chaco, Metis and Scotch [1, 3, 6].

- Metaheuristics: A metaheuristic is a set of concepts that can be used to define heuristic methods that can be applied to a large set of different problems. In other words, a metaheuristic can be seen as a general algorithmic framework that can be applied to various optimization methods with relatively few modifications to make them suitable for a specific problem. Several metaheuristics have been adapted for graph partitioning, such as simulated annealing, tabu search and genetic algorithms [12, 13, 14, 15].

IV. Proposed Heuristics

R. S. Bonato [4] proposed four combinatorial heuristics for GPP-k, where the first three construct a k-partition of the graph at a time until k subsets be created. The fourth heuristic is a version of the third heuristic that uses recursive bisections to achieve the k subsets. All heuristics presented perform a routine of improvement after partitioning in order to refine the cut size of each partition. The basic idea is to build a partition of the graph, accumulating vertices in their subsets using a given criterion, where the criterion for choosing a vertex to be added to the new subset is the difference between the first three heuristics, except for the fourth heuristic, which is a recursive implementation of the third heuristic.

1) Heuristic 1: At each iteration of the method of construction of the subset in this heuristics, a vertex v is added to the subset p and its adjacent vertices are inserted in a list called frontier that defines the boundary of the subset p with the other subsets. The vertex to be inserted in the subset p is randomly selected among the vertices of the border. After the insertion of the vertex, the cut size value is updated with the gain $g(v)$ of the vertex .

2) Heuristic 2: In the previous heuristic, the vertices are added to the new set being built without any criteria, in a random way. In this new heuristic, the main difference is that now each vertex belonging to the boundary vertex v has its gain $g(v)$ computed and stored in decreasing order of gain values in a data structure called bucket. At each step of execution, the vertex with higher gain is inserted into the expanding set. After inserting the new vertex into that subset, the gains of the vertices adjacent to vertex v belonging to its boundary are updated and the gains of the vertices that are not on the border are inserted into the bucket.

3) Heuristic 3: As in Heuristic 2, Heuristic 3 computes and stores the gains $g(v)$ of each vertex of the boundary of the vertex v in descending order. The difference between them is that Heuristic 3 chooses a random vertex from a restricted subset formed by some vertices (or all) of them that make up the border instead of simply taking the vertex with the highest gain to compose the new subset p.

These border vertices are those that involve a smaller increase in the size of the cut of the graph. This subset is called the Restricted Candidate List (RCL), whose size is defined by the parameter α, with α in the range [0, 1]. This parameter controls the quality of the vertices of the RCL. When $\alpha = 0$, the choice of the vertex to be added to the new subset is totally greedy, making the algorithm behave exactly as Heuristic 2. On the other hand, $\alpha = 1$ means a completely random choice, so that the algorithm will behave like the heuristic one.

4) Heuristic 4: this heuristic is an recursive application of Heuristic 3. This fourth heuristic forms a k-partition of the graph by applying recursive bisections. The graph is initially bisected, then the improvement method is called to refine the bisection created, and this strategy is applied recursively on the resulting two subsets, and so on. The algorithm builds the bisection adding vertices one at a time until the free half of the vertices of that subset has been inserted.

5) Improvement Method: The improvement method is used by all proposed heuristics to refine the partial cut of the graph. This improvement subroutine receives as parameter two lists, one of which is the subset constructed and the other is the list of vertices in the boundary of that subset. The subroutine attempts to replace vertices from one list to another based on the gain $g(i)$ for each vertex i. This represents the gain and the cut of the graph decreases if the node i is moved from one

SEEK DIGITAL LIBRARY

Proc. of the Intl. Conf. on Advances in Computing and Information Technology-- ACIT 2014
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-8859-9

subset to another. This refinement technique is similar to the FM algorithm. However, while the FM algorithm evaluates all vertices of bisection, the proposed improvement method evaluates only the vertices of the subset constructed and its border [4].

V. The Parallel Heuristics

All the heuristics proposed by R.S. Bonatto [4] have been implemented in a multistart configuration in which the algorithm builds several partitions and uses only those that result in the shortest cut. Thus, more iterations of the algorithm means greater probability of finding good partitions. In a serial implementation, a high number of repetitions of the algorithm could generate a high computational cost (use of resources and execution time).

The heuristics 1, 2 and 3 have been implemented in parallel based on the concept of Heuristic 4, which creates graph bisections recursively until *k* subsets of vertices are constructed.

The parallel algorithms were implemented using the Java language and using a message passing library (or Message Passing Interface-MPI) for sending messages between nodes in the cluster called MPJ Express. Furthermore, the algorithm considers the use of threads for running on clusters with multiprocessor/multicore architecture.

The main algorithm of the partitions consists of two main parts. The first one is executed if the number of nodes used in the cluster is equal to one, thus configuring a serial execution. The second main part of the algorithm is executed if the number of nodes to be used in the cluster is greater than one.

The parallel algorithm is implemented as follows: two loops determine the number of times that the algorithm is executed. The external loop with h rounds will be executed $h_{max} = \log_2 k$ times, where k is the final number of partitions of the graph. For each iteration of this external loop, an internal loop runs 2^h times. The internal loop initializes the number of threads specified for execution. Each thread then will run a number of iterations of partitioning determined by configuration and its related improvement routines to improve the cut size. At the end of execution, each thread will have obtained its best cut in that round.

The algorithm takes the smaller cut numbers between all threads, and stores the initial and final partitions and the border of the new created partition, since they will serve as input parameters for the next round, if the number of partitions of the graph (k) is greater than 2.

At the beginning, initial partition has all vertices of the graph and final partition is empty. At the end of the iteration, both partitions have half of the vertices of the initial partition, with a difference of at most one vertex. The total value of the cut size of the graph is added to the value of this obtained smaller cut after the execution of the threads and their interactions. A new round h begins if $k > 2$.

The main idea behind the parallel partitioner is to distribute pairs of partitions of vertices (initial and final) that make up a graph in that stage of partitioning for each node of the cluster. After that, each node executes the graph partitioning algorithm and the best solution with the smaller cut number between all of them is selected for that particular iteration round in each node. After that, each node returns its best solutions to the root node, which includes the smaller cut numbers and their respective partitions.

On each node of the cluster, threads are also initialized depending on the number of processors and cores of the node, allowing further iterations in an attempt to obtain the lowest cut. On each processor in each node of the cluster, the algorithm behaves exactly as in the serial version.

In the parallel implementation , the parallel partitioner also runs the main external loop $h_{max} = \log_2 k$ times , where k is the final number of partitions of the graph. An important difference considering the serial solutions remains in the fact that the parallel partitioner distributes pairs of partitions according to specific distribution rules, while the serial partitioner works with all partitions without distinction.

Figure 1 shows an example of execution.

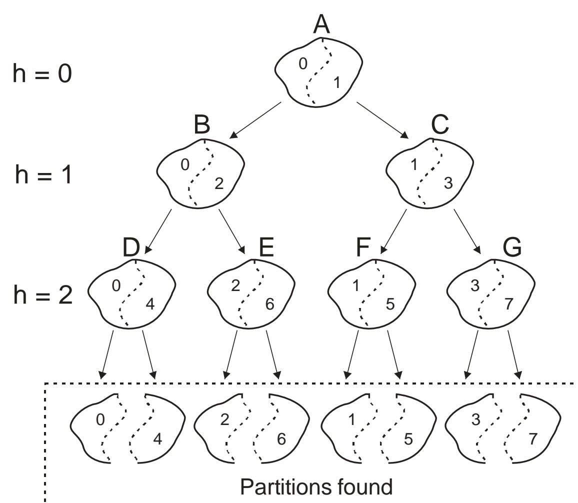

Figure 1. Example of serial partition with 8 partitions.

The rules of distribution of partitions are as follow: in the first round of execution (h = 0), every p (where p is the number of nodes in the cluster) nodes in the cluster receive the initial and final partitions equal to 0 (containing the vertices before partitioning) and 1 (empty before partitions), respectively. All nodes work in parallel trying to find the lowest cutting and its corresponding partitions. All nodes send their best cuts to the root node and the node with the lowest cut size sends its initial (0) and final (1) partitions to the root node. In the next round of execution (h = 1), the nodes numbered (ranked) between 0 and (p/2 – 1) receive partitions 0 (containing one of the partitions obtained in the previous round) and 2 (empty before partitioning), while nodes with rank between (p/2) and (p – 1) receive partition 1 (containing the other partition obtained in the previous round) and 3 (empty before partitioning). Now, there are two halves of the cluster nodes working in parallel to calculate the smaller cut number between partitions 0 and 2 and between partitions 1 and 3. All nodes send to the root node their best cuttings.

SEEK DIGITAL LIBRARY

Proc. of the Intl. Conf. on Advances in Computing and Information Technology-- ACIT 2014
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-8859-9

Finally, the root takes from all nodes the partitions 0, 1, 2 and 3 with the smallest calculated cut size. In the following rounds, the process is repeated such that in each round the number of nodes that perform the partitioning of each pair of partitions is halved while the number of partitions obtained in each round is duplicated.

VI. Results

The algorithms proposed by R.S. Bonatto [4] showed best results for graph bisection when compared to multilevel Metis and Chaco. According to the study proposed by Bonatto and Amaral [4, 16], the Heuristic 3 seems to be the best solution considering overall results. Because they are multistart algorithms, the serial version had to run 10 times, with 100 iterations in each one, in order to obtain the best solution.

Likewise, the parallel version of Heuristic 3 was executed 10 times, with 100 iterations in each one, with the difference that the execution was carried out in a cluster with 16 nodes, each one running 16 threads simultaneously, resulting in a total 16 x 16 x 100 = 25,600 iterations. The smaller cut numbers are shown in the table I.

TABLE I. RESULTS OF EXECUTIONS

Graph	Metis	Chaco	H3 Serial	H3 Parallel
144	**6871**	6994	7248	7575
3elt	98	103	93	**90**
598a	2470	2484	2476	**2463**
add20	741	742	715	**646**
add32	19	11	11	**11**
airfoil1	85	82	77	**74**
Big	165	150	160	**146**
CCC5	28	29	16	**16**
crack	206	266	194	**184**
data	203	234	195	**195**
fe_rotor	2146	2230	2161	**2107**
fe_tooth	4198	4642	4113	**3984**

The parallel solution for Heuristic 3 showed no improvement in the cut of the graph 144. A variation in the configuration of the parallel algorithm of the Heuristic 2 (purely greedy) results in a better cut. With only 10 runs of 10 iterations each, running in the same configuration of 16 nodes and 16 threads per node in the cluster, the cut of the graph has been improved, reaching the value of 6,856.

Compared with the serial algorithms proposed, the total execution time of the parallel algorithm in the cluster was higher. But, in essence, we are comparing only 100 iterations of the serial solution against 25,600 iterations of the parallel algorithm. In the analysis of speedup, when comparing the number of iteration of boot solutions, the parallel solution presents a significant gain of time and improve the quality of the cut.

In addition to the improvements made in the cuts of bisections, the parallel algorithm brought significant gains in speedup. For an increasing number of iterations, the purely serial implementation would become unworkable in practice, however, the parallel algorithms using both a number of nodes in a cluster and also a number of threads in each processing node, made the execution times dropped considerably.

Table II shows different configurations in terms of number of nodes, threads and iterations that were considered for performance evaluation. For each configuration named by the attribute "name", 6,400 iterations are performed. Note that the configuration 01n01t simulates serial execution and the others its parallel execution in terms of number of processing nodes and threads used.

TABLE II. TEST SCENARIOS

Name	Nodes	Threads	Iterations	Total
01n01t	1	1	6,400	6,400
01n16t	1	16	400	6,400
02n16t	2	16	200	6,400
04n16t	4	16	100	6,400
08n16t	8	16	50	6,400

Table III shows the execution times, in seconds, for the different configurations performed for each graph analyzed.

TABLE III. EXECUTION TIMES FOR DIFFERENT SCENARIOS

Graph	01n01t	01n16t	02n16t	04n16t	08n16t
144	9,987.645	2,267.888	1,825.586	902,987	446,436
3elt	21,511	5,924	3,569	3,318	1,859
598a	4,884.235	1,222.244	868,995	428,393	218,654
add20	12,979	3,608	2,506	1,796	1,431
add32	12,979	3,608	2,506	1,796	1,431
airfoil1	19,142	4,206	3,226	2,367	2,419
Big	84,805	18,077	13,100	7,111	3,701
CCC5	0,685	1,038	0,693	0,567	0,397
crack	48,378	9,636	7,439	4,091	2,509
data	20,819	5,546	4,352	2,161	1,981
fe_rotor	3,004.040	662,326	466,571	246,643	136,502
fe_tooth	1,976.274	424,307	327,463	160,237	86,213

The figure 2 shows the average execution time of all graphs for each configuration. The curve shows a natural decrease of the execution time with the increasing number of execution nodes and threads.

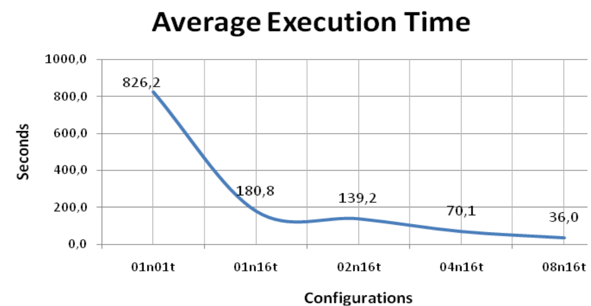

Figure 2. Average execution time for different scenarios showed in Table II.

As a result, the parallelization achieves a very good rate of speedup, especially observed in the execution of large graphs (for example, 144, 598a, fe_rotor).

SEEK DIGITAL LIBRARY

Proc. of the Intl. Conf. on Advances in Computing and Information Technology-- ACIT 2014
Copyright © Institute of Research Engineers and Doctors. All rights reserved.
ISBN: 978-981-07-8859-9

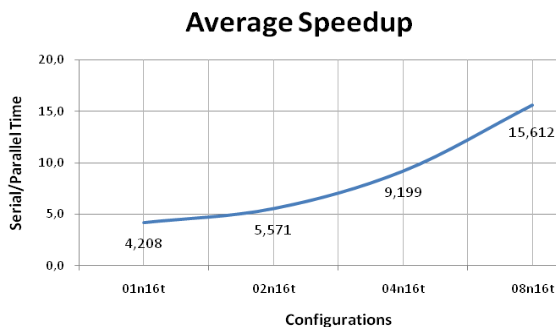

Figure 3. Average execution time for different scenarios showed in Table II.

Note that figure 3 shows the average speedup for running the parallel solution for all graphs. To calculate the speedup for each graph you should use equation (1). For example, the speedup of graph 144 running scenario 08n16t is:

S8= T01n01t / T08n16t = 9.987,645 / 446,436 = 22,37

VII. Conclusions

Optimal solutions to graph partitioning with high numbers of vertices become computationally infeasible in practice. Several heuristics and metaheuristics have been proposed to overcome this difficulty.

The present work proposes an efficient parallel solution to the GPP problem based on the implementation of existing heuristics in a computational cluster with identical processing nodes based on Intel QuadCore processors and supported by MPI/Java programming platform. The proposed solution improves the general performance of the heuristics presented in [4] in two aspects: 1) improvement of the execution time, with a considerable speedup related to the serial solution and, 2) improvement of the quality of the created partitions, by introducing some random features into the original heuristics.

After the parallelization of the heuristics proposed by R.S. Bonatto [4], the performance of the parallel solution of Heuristic 3 showed improvement in 50% of the cut size of the graphs when compared to serial algorithms. In the remaining 50%, the cuts were at least equal, never worse than in the other serial algorithms.

The Heuristic 3 showed the best results, except for the graph 144 that, among all graphs analyzed, is the one with the largest number of vertices. In this case, Heuristic 2, purely greedy, was more appropriated. The main focus of this work was to improve the cut size of the partitioning graph problem. However, time savings were also obtained when comparing the execution of the same number of iterations of the algorithm on an isolated node of the cluster (monothread and serial execution) with the implementation of 16 nodes, each running 16 threads in parallel.

In addition to improve graph cuts, time savings were also obtained when comparing the execution of the same number of iterations of the algorithm on singlethread processor (running purely serial) with the execution using 1, 2, 4 and 8 nodes, each one running 16 threads in parallel.

As conclusion, the implementation of threads in parallel algorithms for use in multi-core processors itself has reduced substantially the execution times for graph bisection.

References

[1] K. Schloegel, G. Karypis and V. Kumar, "Graph partitioning for high performance scientific simulations," CRPC Parallel Computing Handbook. Morgan Kaufmann, 2001.

[2] C. Ou and S. Ranka, "SPRINT: Scalable Partitioning, Refinement, and INcremental partitioning Techniques," unpublished.

[3] P.-O. Fjallstrom, "Algorithms for graph partitioning: a survey," in Linkoping Electronic Articles in Computer and Information Science, vol. 3, no. 10, 1998.

[4] R.S. Bonatto, "Algoritmos heurísticos para partição de grafos com aplicação em processamento paralelo," Dissertação de mestrado, Universidade Federal do Espírito Santo, Vitória, 2010.

[5] U. Benlic and J.-K. Hao, "Hybrid metaheuristics for the graph partitioning problem," unpublished.

[6] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," in Siam J. Sci Comput., vol 20, no 1, pp 359-392, 1998.

[7] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," in Journal Of Parallel And Distributed Computing, no 48, pp 96-129, 1998.

[8] S. Guattery and G. L. Miller, "On the performance of spectral graph partitioning methods," in Sixth Annual ACM/SIAM Symposium on Discrete Algorithms, 1995.

[9] H. Qiu and E. R. Hancock, "Graph matching and clustering using spectral partitions," in Pattern Recognition, no 39, pp 22-24, 2006.

[10] C. Fiduccia and R. Mattheyeses, "A linear-time heuristic for improving network partitions," in 19th IEEE Design Automation Conference, pp 175-181, 1982.

[11] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," in The Bell System Technical Journal, pp 291-307, 1970.

[12] D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Schevon, "Optimizationby simulated annealing: an experimental evaluation; part I, graph partitioning," Oper. Res., no 37, pp 865-892, 1989.

[13] E. Rolland, H. Pirkul and F. Glover, "Tabu search for graph partiotining," in Ann. Oper. Res., no 63, pp 209-232, 1996.

[14] T.N. Bui and B.R. Moon, "Genetic algorithm and graph partiotining," in IEEE Transactions and Computers, no 45, pp 841-855, 1996.

[15] Dorigo, M. (n.d). Citing Websites. Metaheuristics Network. Retrieved May 13, 2013, from http://www.metaheuristics.net

[16] R.S. Bonatto and A. R. S. Amaral, "Algoritmo heurístico para partição de grafos com aplicação em processamento paralelo," apresentado no XLII Congresso da Sociedade Brasileira de Pesquisa Operacional, Bento Gonçalves, 2010.

[17] S.E. Schaeffer, "Graph clustering," in Computer Science Review, no I, pp 27-64, 2007.

About Author (s):

Roney Pignaton da Silva, São Mateus, ES, 08/04/1972. Graduated with a BS from Federal University of ES, Brazil in 1997, a MS from the Federal University of ES, Brazil in 1999, and a PhD from Polytechnic University of Madrid, Spain in 2004. He is a Professor of Engineering Computing at Federal University of ES, Brazil. His current research interests include High Performance Parallel Software applied to simulation, Network and Telecommunication.

SEEK DIGITAL LIBRARY