Particle Swarm Optimization for the Exploration of Distributed Dynamic Load Balancing Algorithms

Mohammed M. Aldasht
College of IT and computer engineering,
Palestine Polytechnic University, Palestine

Abstract - *Evolutionary algorithms provide mechanisms that can achieve efficient exploration for complex design spaces. Also, they constitute an efficient tool for identifying the best alternatives to implement the solution of a certain problem. In this work we use particle swarm optimization (PSO) to find the best alternatives for the distributed load balancing procedure in heterogeneous parallel computers. We have classified and parameterized the different distributed strategies of the dynamic load balancing, then we have applied a methodology based on PSO capable of analyzing the characteristics of the alternatives of load balancing when considering different types of problems and parallel platforms. As an application example of the proposed methodology we will show the results corresponding to the dynamic load balancing in a heterogeneous cluster of PCs for a parallel branch and bound algorithm.*

Keywords: *Evolutionary algorithms, Particle Swarm Optimization, heterogeneous clusters, dynamic load balancing procedures.*

1. Introduction

The target of a load balancing algorithm is to distribute the computational work between the different processing nodes of the parallel and/or distributed machine, so that the resource utilization is maximized to optimize the performance of the platform [1]. A load balancing procedure must find a trade-off between the utilization of processors that constitute the parallel machine and the costs associated with the communication and synchronization between these processors, so that the execution time of the parallel application is minimized.

Dynamic load balancing exploits the communication resources of the parallel platform to exchange state information and tasks between the processors. Therefore, processors use the local information which they have about the global state of the system, to make decisions that allow obtaining of minimal response time and maximum performance. The efficiency of a load balancing algorithm depends on the communication cost between processors, the complexity associated with the decision making procedure in each processor, and on the cost of maintaining relative information of the global state of the system in each node.

Dynamic load balancing strategies can be implemented either in centralized or distributed model. In the *centralized* case, a processor is responsible for maintaining information of the global state of the system and, then, carries out the tasks assignment to processors. In case of systems with a high number of geographically distributed processors, this strategy turns out to be not feasible, for which the distributed strategies are the most suitable. In the *distributed* strategies, the nodes of the system have local information about the global state of the system and they make the decisions in an autonomous form considering this local state information and the information that they exchange with other processors. Given that the processors use partial information about the state of the system, decision making for load distribution, usually, *may* not be optimal.

In the literature there are described numerous examples of the distributed procedures, which they make up a very complex design space. Therefore, deciding about the type of procedure to choose, according to the problem and the platform, turns out to be quite complicated. Particle Swarm Optimization (PSO) is a kind of swarm intelligence algorithms which are meta-heuristics that simulate the social life of some animals used to solve complex problems. PSO is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995. In this paper we present the possibilities of using the PSO in the field of distributed dynamic load balancing.

In section 2 we describe the characteristics of the distributed procedures for dynamic load balancing. Section 3 presents a generic procedure of load balancing to which there can be applied an optimization method based on PSO. Finally, the experimental results appear in section 4 and the conclusions of the paper in section 5.

2. Classification of load balancing procedures

In the last years a lot of load balancing procedures have been proposed [2-6, 14-16] looking for scalability enhancements and portability. In general, it is an issue of new ideas about how to select and to distribute tasks between processors in order to minimize the volume of work to transfer between processors, and to support or to improve the locality in the communications of the parallel program. With the objective of minimizing the overhead associated with maintaining and updating the load information, or assuring that it keeps within reasonable limits [7].

In a distributed procedure, once the tasks are distributed between the processors, these take charge both of the processing of the tasks that are assigned to them, and of the redistribution of the tasks to respond to the changes that can occur in the state of load in the processors, due to the dynamic nature of the platform (modifications on the load of the different nodes due to the presence of other applications), the characteristics of the applications (irregular applications), or failure in some processor or another element of the hardware of the platform. In a distributed procedure, each processor must implement, in some way, the procedures correspond to the stages of *load evaluation*, *initiation of distribution* [5,8], *calculation of the volume of work to transfer* [5,6,9], *selection of tasks*, and *task migration*. These stages are implemented using a series of policies:

- The *information policy:* this policy determines the characteristics of information exchange between the processors in order to have an updated image about the state of load of the whole machine. The set of local and remote information that a processor has, allows it to determine whether to initiate a redistribution of load, the amount of work that it should transfer, if it should act as transmitter or receiver, etc. The most important dimensions to characterize an information policy are related to the spatial frame and to the temporary frame. Thus, an information policy should fix the topology that the processors define with mutual state information. A processor can have information about all the processors in the platform, of those with which it has neighborhood relations according to the interconnection network, or of groups of processors that it could establish according to some criteria (for example, a group of randomly selected processors) that can change temporarily. Moreover, the information policy must establish the moments in which the processors must exchange information: periodic or on-demand information policy. In the periodic one, the processors exchange load information with a frequency that we will name the load balancing frequency (LBF).In the case of on-demand information policy, the state information is exchanged whenever it is necessary to realize a load redistribution, since it is supposed that it is the moment in which changes in the processors charges will take place that cannot decide locally.
- *The transfer policy* establishes the conditions under which the migration of tasks must take place between processors. For this, every processor may use the load information (local and remote) and it decides if it must transfer tasks or request task to be transferred to it. This way, according to which of the processors will initiate the load balancing operation, the transfer policy can be sender initiated, receiver initiated, or symmetrical. In case of the sender initiated transfer, the overloaded processor will begin the load distributed operation and send tasks to other processors. If the transfer is receiver initiated, the processor that needs work will initiate the load distribution operation. Finally, in the symmetrical transfer policy, the overloaded processor which is going to send work to another processor and the one that requests work they must synchronize to realize the transfer of one to other.
- The *location policy* determines the processors that intervene in the transfer of tasks that take place in the load redistribution. This policy is related to that of transfer, since to decide if the level of load in a processor can make it a sender or a receiver corresponds, precisely, to the location policy. Four different types of location policy: the policy that use *one threshold*, the policy that use *two thresholds*, the policy of *minimum/maximum level* of load, and the random location policy. In the one threshold policy a node is located as sender or receiver as its load is, respectively, above or below a certain level considered as a threshold. In the location policy of two thresholds, two thresholds are used, one of them (the high threshold) establishes the value over which the processor is considered to be overloaded, and other (the low threshold) represents the value below which the processor is considered to be underloaded. A processor is sender if its level of load is over the high threshold, and is receiver if its load is below the low threshold. At the moment of establishing the two values of thresholds it is necessary to have in mind that the load of the application can change over the time, therefore, it is suitable to use thresholds that change dynamically, adapting itself at the level of load of the system.
- The *selection policy* is the one that determines what tasks to chose to be transferred from a processor that has been designated as a sender by the location policy (and in some cases by the distribution policy, as we will see). Two alternatives exist for this policy. On

the one hand the *preemptive* selection policy, in which, between the tasks that can be selected, the task that is being processed is included. On the other hand, non-preemptive selection of tasks; only the tasks that are waiting can be selected. The selection policies only need information that allows estimating the load of work of each one of them.

- The *distribution policy* determines the way of balancing the load between the processors that the location policies have selected as senders or receivers. This policy is related to the stages of load evaluation, task selection and migration. In the distribution policy, the selected tasks to constitute the work to exchange are those that the selection policy determines.

The parameters that must be fixed to, completely, specify the alternatives of different policies are:

- The *granularity*, GRN, which indicates the number of tasks in which the problem is divided. It can vary between 1 and N, where N is the maximum number of tasks that can be considered in the problem. A low value of GRN indicates a fine granularity provided that few tasks are created. A parameter between 0 and 1 is used that relates the value of N and that of GRN.
- The *thresholds*, TSD1 and TSD2, are the values used as thresholds in case of a location policy with one threshold (TSD1) or with two thresholds (TSD1 is the low threshold and TSD2 the high threshold). The values of the thresholds can vary between 1 and GRN. The parameters TSDP1 and TSDP2 are used, with values between 0 and 1, for establishing the relation between TSD1 and TSD2, respectively, with GRN.
- The load balancing frequency, LBF, is a parameter that can vary between 1 and GRN (a parameter called LBP, between 0 and 1, is used to relate between LBF and the granularity). It indicates that during the execution of the parallel program, a processor can initiate operations of load balancing every LBF time intervals.

The classification of the dynamic distributed load balancing procedures according to the different policies allows the parameterization of the above mentioned procedures and to transform a generic procedure for the dynamic distributed load balancing that we have developed (it will be described in the following section). We can get the desired procedure on fixing the parameters to the values corresponding to that procedure. Our approach is similar in many aspects to the one that raises the genetic programming [10], whose target is to generate optimum programs.

```
while (not end) do
{
        if (LBC==LBF) then
        {
                load_distribution()
                LBC=0;
        }
        if (work_queue not empty) then
        {
                process_task();
                LBC=LBC+1;
                LI=LI-1;
        }
}
```

Figure 1. Load balancing function call

Table 1. Parameters of the described procedure

Parameter	Range/Definition	Meaning
N	-	***Problem Size.*** The maximum number of tasks in the problem.
P	-	***Processors***
GRN	⌈GRANP×N⌉ (0<GRANP<=1)	***granularity.*** The problem is divided initially into GRN tasks.
TSD1	⌈TSDP1×GRN⌉ (0<TSDP1<1)	***Lower threshold.***
TSD2	⌈TSDP2×GRN⌉ (0<TSDP2<1)	***Higher threshold.***
LBF	⌈LBP×GRN⌉ (0<LBP<1)	***Load balancing frequency***
LI		***Load Index.*** The number of tasks in the work queue of the processor.
LBC		***Load balancing counter.***
p		Current processor

3. General procedure of dynamic and distributed load balancing

In this section we describe a general procedure of dynamic and distributed load balancing [13]. The definitions, the range and the meaning of the parameters and variables used in the procedures are provided in the tables 1 and 2. In the Figure 1 a call to the load balancing procedure is shown. It could be seen that every LBF time intervals of processing a call to the load balancing function is realized, *load_distribution()*. This should not conclude that load distribution operations should have to begin necessarily. Every time interval is equal to the processing time of a task. Thus, the load index LI of a processor decreases by one each time a task is processed. Obviously, the size of the task, and therefore, the duration of the time interval will depend on the granularity, GRN. As shown in Table 1, there exist a relation between LBF and GRN and is controlled by the parameter LBP, which varies between 0 and 1.

Table 2. Parameters of the described procedure (cont.)

Parameter	Meaning
S	Number of processors selected as senders by the location policy.
R	Number of processors selected as receivers by the location policy.
CS	Sum of load indices for all senders
CR	Sum of load indices for all receivers
LI	Load Index. Numbers of tasks reside in the work queue of the processor.
OL	Overload. $OL = LI - \frac{CS + CR}{S + R}$
W(T_i)	Work load asociated with the task T_i

Figure 2 describes the calls that the load distribution procedure realizes to the functions implemented by the different policies. The calls that are finally realized depend to the parameter used in the program execution according to the policies that intervene in the load distribution. Thus, in the function *load_distribution(),* firstly, three conditional statements appear, each of which corresponds to one of the three alternatives that are considered for the transfer policy.

```
Load_distribution()
{
   if  (transfer_policy=sender_initiated)
            {
              if (LI>TSD) then
              {
                 request_locate_receiver(p);
                 load_information_exchange(receiver_proc);
                 select_tasks();
                 send_tasks(receiver_proc);
              }
            }
    if (transfer_policy=receiver_initiated)
            {
             if (LI<TSD) then
             {
                request_locate_sender(p);
                 load_information_exchange(sender_proc);
                 request_tasks(sender_proc);
                 receive_tasks(sender_proc);
                }
            }
    if (transfer_policy=symetrically_initiated)
            {
                 load_information_exchange(sender_proc);
               locate_sender_receiver();
               if (p=sender_proc)
                 {
                 select_tasks();
                 send_tasks(receiver_proc);
               }
                if (p= receiver_proc)
                {
                 request_tasks(sender_proc);
                 receive_tasks(sender_proc);
                }
            }
}
```

Figure 2. Load distribution procedure

If the transfer policy is sender initiated, the processor verifies if its load index, LI, is greater than the high threshold, TSD. If the used location policy is with one threshold, TSD will be equal to TSD1, and if the used location policy is with two thresholds, TSD will be equal to TSD2. If, it is greater than the threshold that indicates the overloading of a processor, it turns into sender. In this case it calls the function *request_locate_receiver()* in order to get the receiver or the set of receivers to which it must send tasks, which will be determined later using the functions *load_information_exchange()*, and *select_tasks()*, and *send_tasks()*. With respect to the exchanged information, apart from the load index, LI, when the system is startup the nodes exchange information as the CPU speed, the available memory, size of cache, etc. This information allows having a better knowledge of the state of the processors, and can be used in the location of processors suitable for load transfer. Besides, calculations of minimums, maximums of load, etc. as we will see, could be needed in some policies.

In case the receiver initiated transfer, first it calls the function *request_locate_sender()*, to identify the possible sender from whom it could receive the needed tasks. Then, load information will be exchanged with the sender or possible senders using the function *load_information_exchange()*, and the sending of tasks is requested by means of the function *request_tasks()*. Then, the processor waits to receive these tasks before continuing with its processing.

If the transfer initiates symmetrically, after exchanging load information, a call to the function *locate_sender_receiver()* is needed. This function determines whether the processor that has issued the call is sender or a receiver and, in each case, returns one or several receivers or senders, respectively. If the function *locate_sender_receiver()* indicates that the processor is not sender nor receiver, the processor will not intervene in the load redistribution. If the processor is selected as sender it will call the functions *select_tasks()* and *send_tasks()*, and if it is designated as receiver it will call the functions *request_tasks()* and then it waits to receive them by calling to *receive_tasks()*.

The function *select_tasks()* allows to determine the set of tasks that they can be considered at the moment of choose those that must be sent. Thus, in the function there appear two options that correspond to two alternatives considered for the selection policy: *preemptive* or *non-preemptive*. In addition, in the function *select_tasks()* the function *choose()* is to be called. This function implements the distribution policy,

where its form depends on the specific characteristics of the type of procedure that it is considered (*diffusion*, *dimensional exchange*, etc.). The function *choose()* is called with a parameter that refers to the set of tasks that can be chosen according to the selection policy (*preemptive* or *non-preemptive*) and, in our implementation, it associates a volume of load superior to a load index OL, whose definition appears in the Table 1. In the calculation of OL we use aspects related to the topology of connection that is considered between the processors (*all*, the *neighbors*, *group members*, etc.).

The form of these location functions depends on the type of transfer policy in use, keeping in mind the clear interrelationship between the two policies. The function *locate_sender_receiver()*, is called when the transfer policy is symmetrical. There appear the options corresponding to four alternatives for the location policy. So, according to the load index LI, according to whether it is less or greater than TSD1 the processor will act as receiver or sender, respectively, in the case of the location policy of a one threshold. In the case of two thresholds location policy, the same thing is to do but, there are in use the thresholds TSD1 as low load threshold and TSD2 as overload threshold. If the location policy is based on the minim/maximum load it will be necessary to compare the load index with the values of maximum and minim load of the set of processors with that a processor can exchange load. These calculations is done when the calls are realized to the function *load_information_exchange()* in the case of using this location policy. The set of processors that intervene at the moment of doing this calculation are determined by the alternative of topology of the information policy. The way of determining the sender and receiver in case of random location policy also requires information about the maximum and minimum load of the correspondent group of processors.

To determine the index (the indices) of the sender (possible senders) or receiver (possible receivers), respectively, for the processor designated as receiver or sender, the functions *sending()* and *receiving()* are used. These functions perform communication between those processors that can transfer tasks according to the topological alternatives of the distribution policies. From these functions, also it is possible to call to the functions *locate_sender()* (in case of *sending()*) and *locate_receiver()* (in case of *receiving()*), which are described later.

The function *locate_receiver()* allows to determine a receiver processor in the sender initiated transfer policy. In addition to be called from the function *receiving()*, the function *locate_receiver()* is also called from the function *request_locate_sender()*, which generates the calls to *locate_sender(processor)* in a processor from all processors that the distribution policy indicates that they can exchange work with the processor. The function executed in the processor *p* sends to *request_locate_sender()* in processor the load index of the processor, p, if it verifies, from the conditions established in the location policy in use, to be a receiver.

The function *locate_sender()* is analogous to the function *locate_receiver()* but in this case for the processor or processors to which it is possible to request tasks for them. This function is called from *request_locate_sender()* in case that the transfer policy is receiver initiated (also it is possible to be called from *sending()*, as indicated before).

To explore the design space of the distributed procedures for dynamic load balancing, we have used a PSO algorithm (figure 3) that it is employed to search the space defined by the different alternatives of the policies of *information*, *transfer*, *location*, and *selection*, in addition to the values that can take the parameters LBF, GRN, and the thresholds TSD1 and TSD2 (real values). Its target is to find the configurations of the policies and the values of the parameters for policies that can provide a better distribution of load for a parallel application.

PSO algorithms share similarities with evolutionary computation techniques such as Genetic Algorithms (GA), where PSO starts with a population of random solutions and searches for optima by updating generations. But unlike GA, PSO has no evolution operators such as crossover and mutation. The potential solutions, *particles*, fly through the problem space by following the current optimum particles.

In the swarm of solutions, which evolves from one generation to another, it keeps tracking of the particles conserving the *local* best for each particle and the *global* best from the swarm. Thus, it keeps the most suitable solution found in the previous iterations. Precisely, the ability to remember useful information relative to the past iterations is one of the strategies that have been considered in the application of the evolutionary algorithms to the dynamic optimization problems, in which the obtained solutions must bear in mind the changeable nature of the problem [11,12].

For each particle
  Initialize particle
END
Do
 For each particle
  Calculate fitness value
  If fitness > best fitness value ***pbest*** in history
   set current value as the new ***pbest***
End
  Choose the particle with best fitness value as the ***gbest***
  For each particle
   Calculate particle velocity:

$$v' = v + c1*rand*(pbest - present) + c2*rand*(gbest - present)$$

   Update particle position according Calculated velocity

$$present' = persent + v'$$

  End
While maximum iterations or minimum error criteria is not attained

Figure 3. Pseudocode of the developed PSO algorithm

4. Experimental results

The results that appear here correspond to a parallel algorithm of branch and bound for the traveling salesman problem. In this case the volume of calculation of the program is unknown in advance, and it depends on the implicit search process that the algorithm implements. In this way, it is essential to use a procedure of dynamic load balancing.

They are provided (Table 3 and Figures 4-6) the results obtained for a problem size corresponding to 8 cities, in a cluster with 7 nodes with the characteristics described in table 3 and connected via a Gigabit Ethernet.

Table 3: cluster nodes characteristics

# of nodes	CPU Speed (MHZ)	Cache (KB)	RAM
Node1	Xeon, dual processor	2 MBytes	2 GB
Nodes: 2-4	P4 3.2	512	1 GB
Nodes: 5-7	Core 2 Duo 1.8	512	1 GB

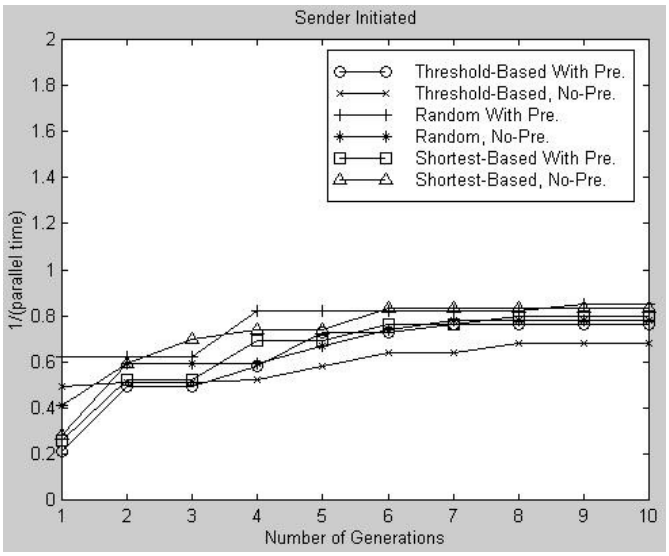


Figure 4: Fitness (1/parallel time) for the alternatives with sender initiated transfer

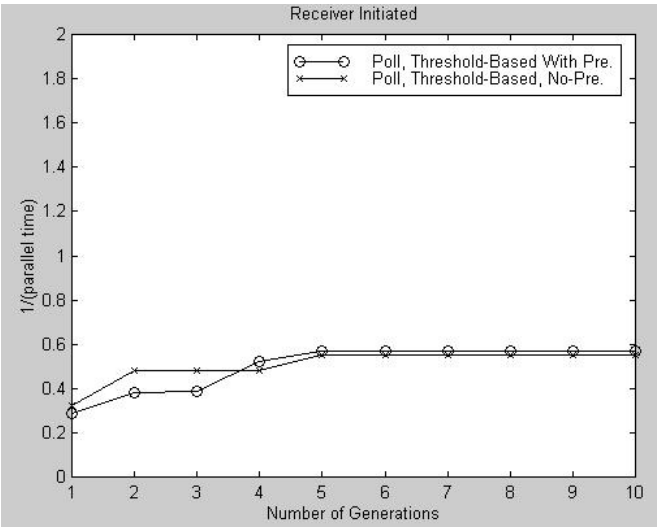


Figure 5: Fitness (1/parallel time) for the alternatives with receiver initiated transfer

Table 3 presents, for the alternative procedures of the symmetric transfer policy (the one that provides better speedup), the values of the parameters to that genetic algorithm converges and the values of the fitness and of the average utilization obtained (the standard deviations are indicated between parentheses). Since value of fitness used is the inverse of the parallel execution time (we have avoided the evaluation of the sequential execution time of the algorithm, which can become extremely high). Figures 4, 5, y 6 represent the evolution of the fitness across the successive generations of the genetic algorithm in the alternative procedures of each transfer policy.

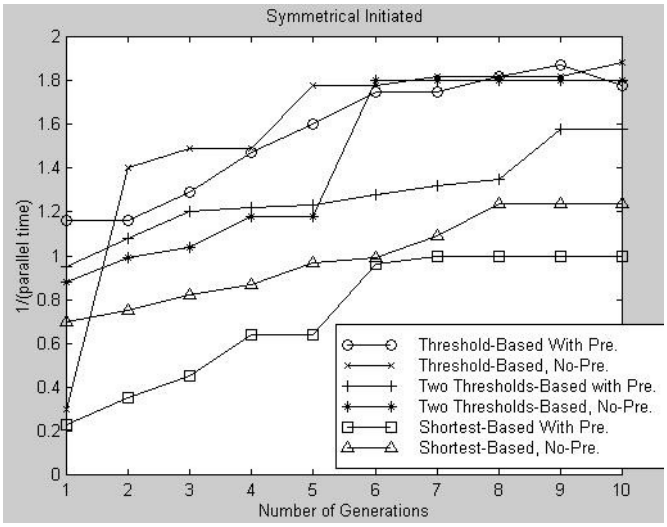
The results obtained for the branch and bound algorithm show that better speedup values are provided

by symmetric transfer policy. The procedures with sender initiated transfer policy are better than those of receiver initiated transfer. Here, the utilizations obtained by the symmetric procedures are similar in some cases to those of the receiver initiated procedures, and the procedures with sender initiated transfer have lower utilization.

If we consider the different alternatives for every transfer policy we have that in case of the procedures with sender initiated transfer (figure 4) and for the receiver (figure 5) all the alternatives converge towards quite close values: it can be considered that they are almost equivalent as for the speedup that they allow to reach. As for the procedures with symmetric transfer, the figure 6 shows relatively different behaviors for different location and selection policies. In this case, the procedures based on a threshold (preemptive or nonpreemptive) and the procedure based on two thresholds with no preemption who provide better performance.

Table 3 Convergence of the parameters for the alternatives with the symmetric transfer policy

Procedure	LBF	GRN	TSD1	TSD2	1/(T Par)	Util. Máx.
Threshould *(Preemption)*	107 (12)	1 (1)	101 (7)	279 (18)	1.87 (0.03)	0.09
Threshould *(No-Preemption)*	103 (8)	2 (1)	106 (8)	279 (16)	1.88 (0.02)	0.11
Two-Threshoulds *(Preemption)*	111 (11)	2 (1)	109 (12)	280 (12)	1.58 (0.02)	0.17
Two-Threshoulds *(No Preemption)*	116 (14)	2 (1)	94 (6)	297 (8)	1.80 (0.03)	0.13
Mín/máx *(Preemption)*	105 (7)	2 (1)	-	-	1.00 (0.01)	0.05
Mín/máx *(No Preemption)*	119 (14)	5 (3)	-	-	1.24 (0.03)	0.03



It figures 6: fitness (1/parallel time) for the alternatives with symmetric transfer policy

The convergence on the parameters GRN, LBF, TSD1, and TSD2 shows greater deviations than in other test programs that have been analyzed [12]. Also, greater changes exist between the convergences of the parameters according to the procedures. For example GRN is between 1 and 2, though in some cases it is equal to 5. The values of LBF to which it converges are between 103 and 121 for the procedures with sender initiated transfer and the symmetric ones, but in case of the receiver initiated transfer they are 85 and 75, respectively, for two considered alternatives. The values of TSD1 are between 90 and 117, and the values of TSD2 are between 273 and 297, though in one case it comes to 328 (sender initiated transfer with location based in the maximum/minimal level of load with nonpreemption). This way, the best procedure for the branch and bound algorithm applied to the traveling salesman problem with 8 cities uses symmetric transfer policy, one threshold location policy, and nonpreemption selection policy; with GRN=2, LBF=103, and TSD1=106.

5. Conclusions

According to the alternatives that can be distinguished in the information, transfer, location, distribution, and selection policies, which they define a load distribution procedure, different procedures have parameterized and the genetic search has been applied in the defined space of design. Thus, it is possible to carry out an optimization of the parameters that define the behavior of each load balancing procedure, which, in this way, is achieved by means of an analogous approach to the genetic programming.

The experimental results obtained show that in all the analyzed cases the symmetric transfer policy with

periodic information policy is better than the sender and receiver initiated both with the on-demand information policy. These results coincide with the conclusions obtained in other works [2]. As for location policies, the most efficient is used to be with one or two thresholds, and as for the selection policy, in some cases the best are those of preemption and in others those who use nonpreemption. The procedures with random location policy are those who provide the worst performance.

6. References

[1] Theys, M.D., et al.:"What are the Top Ten most Influential Parallel and Distributed Processing concepts of the past millenium?". Journal of Parallel and Distributed Computing, Vol.61, pp.1827-1841, 2001.

[2] Antonis, K.; Garofalakis, J.; Mourtos, I. ; Spirakis, P. : "A hierarchical adaptive distributed algorithm for load balancing". J. Parallel Distrib. Comput., 64, pp.151-162, 2004.

[3] Barker, K.; Chernikov, A.; Christoides, N.; Pingali, K.:"A Load Balancing Framework for Adaptive and Asynchronous Applications". IEEE Trans. on Parallel and Distributed Systems, Vol.15, No. 2, pp.183-192. Febrero, 2004.

[4] Watts, J; Taylor, S. "A Practical Approach to Dynamic Load Balancing". IEEE Trans. on Parallel and Distributed Systems, Vol. 9, No. 3,pp. 235-247 March 1998

[5] Xu, C.; Lau, F.:"Load Balancing in Parallel Computers". Kluwer Academic Publishers, 1997.

[6] Willebeek-LeMair, M; Reeves, A.: "Strategies for Dynamic Load Balancing on Highly Parallel Computers", IEEE Transactions on Parallel and Distributed Systems, 4:979-993, 1993.

[7] Dandamudi, S., Piotrowski, A. ,"A Comparative Study of Load Sharing on Networks of Workstations", Proceedings of the International Conference on Parallel and Distributed Computing Systems, New Orleans.

[8] Muniz, F.; Zaluska, E.:"Parallel Load-Balancing: An extension to the Gradient Model". Parallel Computing, Vol. 21, pp.287-301, 1995.

[9] Horton, G.:"A Multi-Level Difussion Method for Dynamic Load Balancing". Parallel Computing, Vol.19, pp.209-218, 1993.

[10] Koza, J.R.:"Genetic Programming: On the Programming of Computers by Means of Natural Selection". MIT Press. Cambridge, Mass., 1992.

[11] J. Kennedy and R. C. Eberhart. Swarm Intelligence. Morgan Kaufmann. 2001.

[12] Branke, J.:"Evolutionary Optimization in Dynamic Environments". Kluwer, 2001.

[13] Aldasht, M.; Ortega, J.; Puntonet, C.G.; Díaz, A.F.:"A Genetic Exploration of Dynamic Load Balancing Algorithms". IEEE Conference on Evolutionary Computation, Portland, Oregon, 2004.

[14] Kai Lu; Riky Subrata; Albert Y. Zomaya,: "On the performance-driven load distribution for heterogeneous computational grids". Journal of Computer and System Sciences, Volume 73, Issue 8, December 2007, Pages 1191–1206

[15] David Clarke; Alexey L. Lastovetsky; Vladimir Rychkov: "Dynamic Load Balancing of Parallel Computational Iterative Routines on Highly Heterogeneous HPC Platforms". Parallel Processing Letters 06, 2011; 21(2):195-217.

[16] D Clarke; A Lastovetsky; V Rychkov: "Column-Based Matrix Partitioning for Parallel Matrix Multiplication on Heterogeneous Processors Based on Functional Performance Models". Lecture Notes in Computer Science 01/2012; 7155:450-459.