A Parallel Memetic Algorithm Applied to the Total Tardiness Machine Scheduling Problem

Vinícius Jacques Garcia[1], Paulo Morelato França[1], Alexandre de Sousa Mendes[2], Pablo Moscato[2]

[1]Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas
C.P. 6101, 13083-852, Campinas, SP, Brazil
{jacques,franca}@densis.fee.unicamp.br

[2]School of Electrical Engineering and Computer Science
The University of Newcastle
Callaghan, 2308, NSW, Australia
{mendes,moscato}@cs.newcastle.edu.au

Abstract

This work proposes a parallel memetic algorithm applied to the total tardiness single machine scheduling problem. Classical models of parallel evolutionary algorithms and the general structure of memetic algorithms are discussed. The classical model of global parallel genetic algorithm was used to model the global parallel memetic analogue where the parallelization is only applied to the individual optimization phase of the algorithm. Computational tests show the efficiency of the parallel approach when compared to the sequential version. A set of eight instances, with sizes ranging from 56 up to 323 jobs and with known optimal solutions, is used for the comparisons.

1. Introduction

In the past decades the Genetic Algorithms (*GAs*) approach has dramatically improved and became a popular methodology to deal with a wide variety of problems. A very important contribution comes from *Hybrid GAs*, which apply some form of problem domain knowledge, generally in the form of good local search operators, in order to improve the search process. Recognizing that the addition of problem-domain knowledge results in important differences and in new practical algorithmic design issues to face, a generalization of *hybrid GAs* was pointed out by Moscato in [14] as a new methodology, and the creation of the new denomination of *Memetic Algorithm* (*MA*) was justified for them.

Several applications of *MAs* have shown that, although a larger computational effort is required by the local search operators, the adoption of such operators leads to better results when compared to those created by an ordinary *GA* [1], [7], [13], [6]. The good performances obtained in all these works is due to a well-tailored local search procedure for the problem being solved, an adequate representation for the chromosome, and the "synergy" of the local search with the recombination operator.

A major challenge found in such algorithms is the design of good local search operators for large instances, since the associated neighbourhood might become extremely large. One of the ways to avoid this problem is to adopt neighbourhood reduction techniques. Nevertheless, in some cases, even with a good reduction, the exploration of the neighbourhood is still very time-consuming. Then, the development of parallel approaches, using parallel execution techniques, becomes a suitable alternative.

In this work is presented a new implementation for master-slave memetic algorithms with hierarchically-structured population, emphasizing on design issues related to load balance and synchronism. This model is applied to solve a set of eight instances of the well-known single machine scheduling problem. Next is presented how this paper is structured: in section 2 we present a description of the problem being addressed; in section 3, the concepts of Parallel *GAs* are described; the sequential and the parallel *MAs* are presented in sections 4 and 5, respectively; computational results are presented in section 6, followed by the conclusions in section 7.

1-4244-0054-6/06/$20.00 ©2006 IEEE

2. Problem statement

The single machine scheduling problem (*SMS*) is one of the most studied problems in the combinatorial optimization field. The interest derives from the frequency it is found in real industry environments. The works of [9] and [10] were among the first articles to address this type of scheduling problem.

There are several variants of the *SMS* problem, depending on the input data and the objective function. A very common one found in the literature is the task of scheduling n jobs, each one with a specific processing time and due date. The objective function is to minimize the total tardiness, characterized by the sum of each individual tardiness in turn related to the jobs' due dates. This problem can become more "complex" if it adds sequence-dependent setup-times for the jobs, precedence constraints, etc.

The simplest total tardiness *SMS* problem, without setup times, is already NP-hard as shown in [5]. Many solution techniques that focus on this problem have been proposed. The references [15] and [11] use dispatch rules together with a priority index to build an approximate sequence, which later will be optimized by a local search procedure. In reference [16], a new recombination operator is created to be used in a *GA*. In [17] a method based on *Simulated Annealing* is developed and in [7] a *MA* with a hierarchically-structured population was presented.

The problem addressed in this paper can be defined as:

1. *Input*: A set of n jobs to be processed in one machine, a list $\{t_1,\ldots,\ t_n\}$ of processing times for each of the jobs and another list $\{d_1,\ldots,\ d_n\}$ of due dates for each one of them. A matrix $\{s_{ij}\}$ of setup times, where s_{ij} is the setup time of job j after the machine has processed the job i.

2. *Output*: A permutation of the jobs that minimizes the total tardiness of the schedule. Tardiness is given by equation 1, where c_k represents the time when job k was finally processed, or in other words, it is the job's *completion time*, and d_k is the due date of the respective job.

$$\sum_{k=1}^{n} max[0, c_k - d_k] \quad (1)$$

Fig. 1 shows a possible schedule for an instance with five jobs (*1-4-3-2-5*), as well as a graphical representation of the total tardiness.

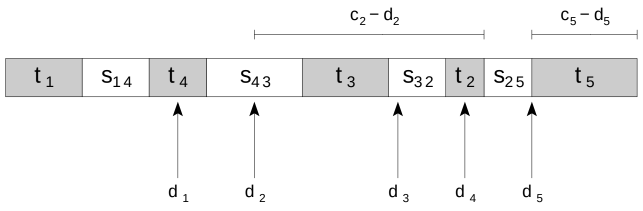

Figure 1. Gantt diagram of a solution for the total tardiness single machine scheduling problem.

3. Parallel population-based algorithms

Many classical forms of parallel genetic algorithms are found in the literature, as shown in [3] and, more recently, in [4]. In [14], a pioneer work in the *MA* field, the importance of such methods was already outlined. Of special importance is Ref. [8], where an *MA* with a matrix-layout population is proposed. Each individual is assigned to a processing unit, occupying a position in this bi-dimensional space. Selection and recombination are done independently after a limited neighbourhood is set.

The parallel model used in this work is called Global Parallel Memetic Algorithm (*GPMA*) (see also [3]). The name derives from the fact that the selection, recombination and mutation operators are applied over the entire population, in contrast with other parallel *MA* approaches where the population is broken into subpopulations. The implementation is generally carried out using *master-slave* programs; a *master* unit assigns some functions of the algorithm to other *slave* units, which execute them and return the result.

The function that is usually distributed to the slaves is the evaluation of individuals, given its independent character. Following this scheme, fractions of the population are assigned to each slave and communication occurs only when they are sent or received. When the master waits for the answer of all slave units, the algorithm is called *synchronous*, preserving all the characteristics of the evolutionary behavior of a sequential method, but with better performance. Another possibility is that the master unit does not need to wait for all the answers, thus characterizing an *asynchronous* method. We note that the *MAs* proposed by Norman and Moscato [14] and ASPARAGOS [8] were of this type. In this case, there is a clear difference with the sequential algorithm: individuals from the same generation can jump to another generation, as if they had migrated, modifying the evolutionary behavior of the algorithm.

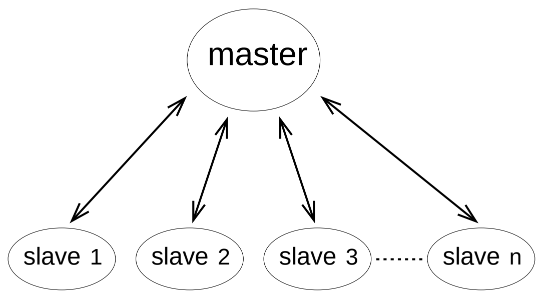

Figure 2. Basic structure of a Global Parallel Memetic Algorithm, each slave would run some individual optimization step on its associated solutions.

The *GPMAs* do not require a specific computational architecture. They can be efficiently implemented in computers with shared or distributed memory. In the first case, the population is stored in the shared memory and each unit of the multi-processor system access parts of it. When distributed memory is used, the population is stored in the processing unit responsible for sending the individuals to the others and for collecting the answers. In both cases there is a cost associated to the communication, resulting in a trade-off between number of slave units and efficiency. This compromise is analyzed in detail in [2]. Fig. 2 outlines a *GPMA*.

4. Sequential Memetic Algorithm

The *MAs* were categorized and described as a new class of evolutionary algorithms in [14]. As the *GAs*, the *MAs* are based on the benefits of selection, reproduction of characteristics of previously discovered good solutions (i.e. forms of generalized recombination) and mutation. What differentiates them is the adoption of a cultural evolution analogy. The most common way that cultural information is transmitted to individuals of the population is to individually evolve them to became fitter individuals by means of a local search procedure. Therefore new individuals are then optimized, leading to a better population. The search in the space of solutions is done with several solutions (individuals/agents) like in other population-based algorithm, making it a concurrent search process. The terminology “sequential” is adopted when the algorithm does not use any explicit parallel mechanism and runs on a single processing unit.

The *MA* implementation presented in this work is described in detail in [7]. A simplified pseudo-code is shown in Fig. 3.

In the pseudo-code of Fig. 3, we emphasize that the optimization procedure (the local search in this case) in step 7 and 9 is executed independently for each individual.

```
Procedure Sequential_Memetic_Algorithm(P)
 1. Inicialize(P);
 2. Evaluate(P);
 3. For i=1 to maxGenerations do
    4. For j=1 to maxNewInd do
       5. parents=selectIndividuals(P);
       6. newInd=recombine(parents);
       7. optimize(newInd);
       8. mutate(newInd);
       9. optimize(newInd);
      10. AddToPopulation(newInd);
11. End.
```

Figure 3. Procedure of a sequential memetic algorithm.

5. Parallel Memetic Algorithm (PMA)

The main motivation for this work comes from the *MA* presented in [7], which has been relatively successful in dealing with the *SMS* problem. Nevertheless, the method has shown some limitations, mainly due to the CPU time required to solve larger instances. A study on the time spent by each step of the algorithm confirmed that the individual optimization step is responsible for over 90% of the total computational effort spent by the algorithm, making it the obvious choice for parallelization. Moscato and Norman like to say that *MAs* constitute a clear example of *agent-driven* parallelism.

Therefore, the individual optimization steps will be performed in parallel and all the other steps of the algorithm (Fig. 3) are sequential. The most adequate model in this case is the *GPMA* that, as described in section 3, uses a master unit. The master will then distribute the optimization task to the slave units. More specifically, the optimization of each individual constitutes a job to be distributed to the slave units. Each slave receives the job, makes the processing and returns the result (the optimized individual) to the master unit.

As mentioned before, our available parallel hardware is a computer network, with several processing units, each one with an independent memory. In order to make an algorithm that centralizes all the jobs and only distributes the individuals’ optimization, we need to simulate a shared memory (Fig. 4). That is, all individuals to be optimized must be kept in a memory that is shared by all processing units.

This simulation is accomplished with *sockets* and, given the communication structure, each socket can only be used by two points, or two processing units.

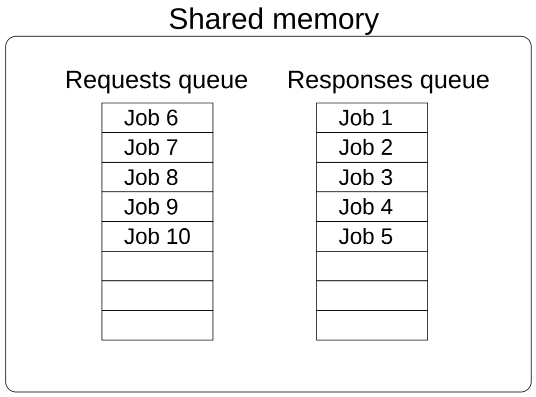

Figure 4. Shared Memory for the parallel memetic algorithm.

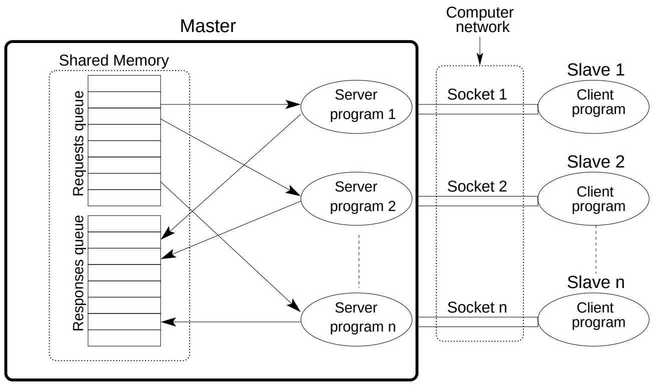

Figure 5. A simplified diagram of the parallel memetic algorithm architecture.

Thus, it is necessary as many sockets as the number of slave units being used. The use of such communication channels does not solve the problem. There is still another one to be considered: how to distribute these jobs to the slave socket units. An important characteristic is that the jobs are built one after the other, due to the concentration of such activities in the master unit. Since the nature of the units might be heterogeneous, with different processor speeds and memory configurations, the goal is to distribute the jobs as to minimize the time required to solve all of them. This problem is very similar to the multiprocessor scheduling problem with *makespan* minimization ($P||C_{max}$) [9]. As the $P||C_{max}$ is NP-hard, it is very difficult to be solved to optimality. Moreover, this problem repeats itself in each generation of the algorithm and any computational effort reduction becomes precious.

The proposed solution is to take advantage of the sequential way that the jobs are created, assigning them by demand, that is, the faster a slave unit is, the more jobs it will process. This distribution function is not controlled by the algorithm: the use of *threads* makes it possible to have concurrent programs in the master unit, with shared memory. Therefore, the decision of which program will be executed at a given moment belongs to the *operating system*. The only compromise is to manage the shared memory in order to avoid data inconsistency, to prevent that a given program be interrupted when any operation in the shared memory is being executed. In order to preserve the sequence of job-sending and job-receiving, this memory is composed of two FIFO queues: the first is used by the jobs to be processed (*requests queue*) and the second by the jobs that were already processed (*responses queue*), as show Fig. 4. A simplified diagram of the proposed PMA architecture is shown in Fig. 5.

In the initialization of the master unit, a server program is initialized for each slave unit available. On the slave unit side a client program is started which makes a socket connection and waits for a job to be processed. As soon as the job is received, it is processed and returned to the respective server program. Each one of the server programs permanently verifies the shared memory for jobs to be processed. Such memory accesses are exclusive, that is, only one server program can access it at a given time.

The process of creating the jobs to be processed by the slaves repeats itself every generation. In the individuals' optimization step, each job will be formed by an individual plus the specification of the local search algorithm to be used. Therefore, it is important to keep the synchronism, that is, after all the jobs were built and put in the shared memory, it is necessary to wait for the result (the optimized individuals). In this way, when the master finishes building all the jobs, it must wait (or not) for all the results in order to advance to the next MA step. In the algorithm implemented, we created a parameter k, which determines the percentage of results that must be received by the master before the algorithm skips to the next step. For instance, suppose that the population is composed of 13 individuals and 20 new individuals are created every generation. In such case, 20 jobs must be executed by the slave units. If we set $k = 0.4$ (40%), this means that the algorithm will wait for at least 8 jobs (optimized individuals) to be returned before the next phase of the algorithm starts. As a consequence, value $k = 0$ corresponds to a *master-slave asynchronous* algorithm and $k = 1$ to a *master-slave synchronous* one.

The complete pseudo-code is shown in Fig. 6. When compared to the sequential MA (Fig. 3), it can be no-

```
Procedure Parallel_Memetic_Algorithm(P,k)
 1. Inicialize(P);
 2. Evaluate(P);
 3. For i=1 to maxGenerations do
    4. For j=1 to maxNewInd do
       5. parents=selectIndividuals(P);
       6. newInd=recombine(parents);
       7. mutate(newInd);
       8. createJobForSlaves(newInd);
 9. newIndividuals=waitProcessedJobs(k);
10. For j=1 to maxNewInd do
   11. AddToPopulation(newIndividuals[j]);
12. End.
```

Figure 6. Procedure of a parallel memetic algorithm.

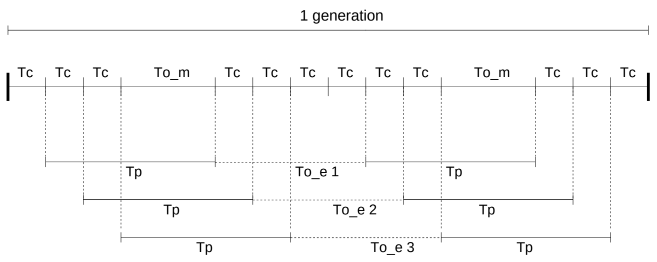

Figure 7. Master-slave parallel memetic algorithm illustration.

ticed that there are just a few modifications. Only steps 8, 9 and 10 were introduced to transform it into a parallel algorithm.

For a better understanding of the individual's optimization-phase parallelization, consider Fig. 7. Suppose that there are 6 jobs (individuals) in the requests queue and 3 slave units. Fig. 7 shows the time events that occur in the master unit during an entire generation of the *PMA*. Note that the time to process a job (*Tp*) is the same in all slave units. The time *To_m* indicates the master unit's idle time and *To_e* represents the slave unit's idle time. If we disregard the amount of time spent in other steps of the algorithm during a generation, and take into account only the optimization step, the duration of a parallel algorithm's generation (Tg_p) is given by equation 2 and the sequential algorithm's duration (Tg_s) by the equation 3. For both algorithms, *Nt* corresponds to the number of jobs (or individuals) to be distributed to the slaves, *Ne* is the number of slave units and *Tc* represents the time needed to transmit a job from the master to a slave or vice-versa.

$$Tg_p = \frac{Nt.Tp}{Ne} + \frac{Nt.Tc}{Ne}(1 + Ne) \tag{2}$$

$$Tg_s = Nt.Tp \tag{3}$$

In order to guarantee that the performance of the *PMA* is better than its sequential version, it is needed that $\frac{Tg_s}{Tg_p} > 1$. After a few mathematical operations in equations 2 and 3, we obtain equation 4. It shows that the job processing time *Tp* must be always larger than *Tc* and that the greater is the relation $\frac{Tp}{Tc}$, the better will be the performance.

$$Tp > Tc.\frac{Ne + 1}{Ne - 1} \tag{4}$$

Equation 4 also shows that the attempt to parallelize other steps of the *MA* is not promising. For instance, making the recombination step become parallel is not viable because the time necessary to make the recombination is almost negligible, but the communication time is not.

6. Computational Experiments

In order to show how efficient is the parallel approach, computational tests were carried out using a set of instances with known optimal solutions.

The *speedup*, defined as the quotient between the time *Ts* to run the sequential algorithm and the time *Tp* for the parallel version, is used as the performance criterion.

The *SMS* instances were created from solved Asymmetric Traveling Salesman Problem (*ATSP*) instances, available at **TSPLIB**[1]. The letter after the name of the instance classify its difficulty and refers to the way the processing times are created; **L** represents a harder instance than **H**. For the **L** instances, the setup-times become more critical in the scheduling, emphasizing the *ATSP* aspect of the problem. For the **H** instances, the processing times become more relevant in the scheduling, being such instances much easier to be solved by the algorithm being used. More information on how the instances were generated can be found in [12].

The local search used to optimize the individuals is based on the well known *all-pairs* neighbourhood. Given a solution, this neighbourhood is constructed by interchanging all pairs of jobs. As mentioned in the introduction, the excessive size of this neighbourhood needs reduction schemes. In our tests two types of reductions are performed, both of them make use of a function which approximates the total tardiness considering setup and processing times without performing the movement. With this approximated value it is

[1]http://www.crpc.rice.edu/softlib/tsplib/

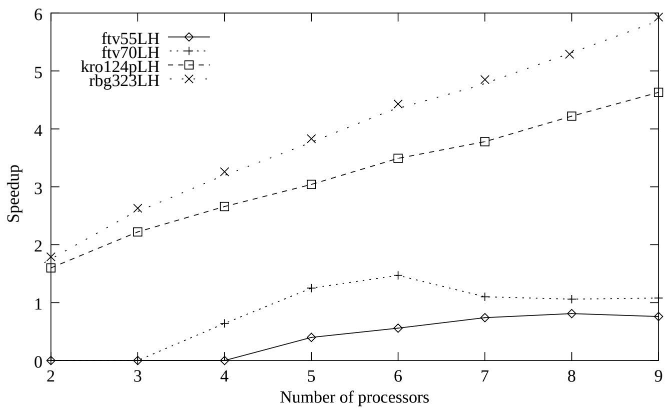

Figure 8. Speedups for the 50%-reduction.

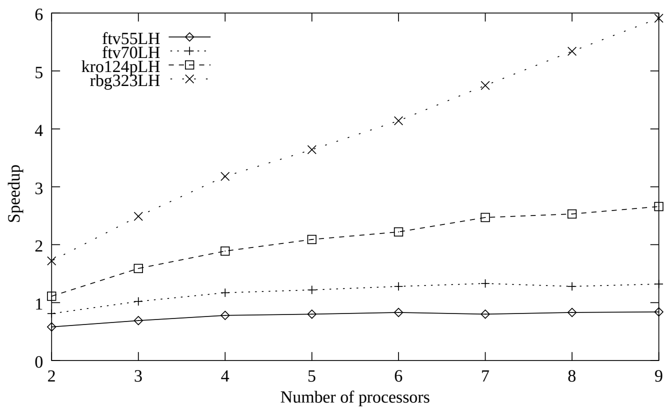

Figure 9. Speedups for the 90%-reduction.

possible to choose for evaluation only movements that yield lower values than the incumbent solution.

The first test was carried out with a 50%-reduction in the local search neighbourhood. A complete description of the neighbourhood reductions are available in [7]. Table 1 illustrates the speedup values obtained, as well as the number of jobs (n) present in each instance, the average time to perform a local search in an individual[2] (Tp) and the communication time (Tc). In the second test we used the same neighbourhood but with a reduction of approximately 90% of the search space. The results are shown in Table 2. Fig. 8 and Fig. 9 illustrate the evolution of the speedup for a subset of instances, and for the first and second test configurations, respectively.

The implementation of this work was done using *Java*^{TM} (*Sun Microsystems*), *JDK* version 1.4.1. For the computational tests, we used a 10-Mbits *Ethernet* network, with nine PC-Compatible Intel Celeron 330 MHz computers, each one with 64 MB of RAM. The operating system was Linux, *kernel* version 2.4. All the executions of the sequential and parallel *MA* spent 20 generations and each one was repeated 10 times. The memetic parameters are as follow: population size equal to 13; offspring size equal to 20; and mutation rate equal to 50%. An important detail is that the *master-slave* algorithm used in our experiments is the *synchronous* one (i.e. the parameter k is set to 1).

The results show that the greater is the relation $\frac{Tp}{Tc}$, the larger is the speedup, as shown by equation 4. This fact can be proved when comparing 56 and 100-job instances: the former class has lower speedups than the latter one. In addition, speedups less than 1 for 2 processors indicate that it is not promise to apply the parallel structure proposed: for instances *ftv55H*, *ftv55L*, *ftv70H* and *ftv70L* on Table 1 and *ftv55L* and *ftv70L* on Table 2 there is a marginal benefit in using the master-slave algorithm.

Considering the influence of the all-pairs neighborhood reduction on Tp, we could check that the more restricted are the movements, the lower is the local search time and, thus, the lower will be the speedup. Only for instances *ftv55H* and *ftv70H* there was an increase in Tp when comparing 50%-reduction to 90%-reduction, what leads to higher speedups as pointed out on Tables 1 and 2. For the other instances, in general, we obtained lower *speedup* values for the most restricted reduction (90%) compared to the less restricted one (50%-reduction). Only do give an idea about the time required to solve an instance, the worst case for the instance *rbg323H* takes 7212 seconds, considering a sequential execution (1 processor). For 9 processors, this time is reduced to 1119 seconds.

7. Conclusions

This paper proposes a new implementation for master-slave memetic algorithms with hierarchically-structured population. It is emphasized on design issues related to load balance and synchronism in order to deal efficiently with time-consuming local search operators. Theoretical results provide a trade-off between processing and communication time which is associated with the *speedup* value.

The application of parallel computation techniques in memetic algorithms is very promising when the CPU time is the performance criterion or a critical limitation. The associated complexity is acceptable considering the resulting performance improvement.

The results also validate the efficiency of the parallel model for the set of instances tested. It became evident that the larger is the instance, the greater will be the

[2]When the sequential algorithm is used.

Table 1. Parallel memetic algorithm performance for the 50%-reduction.

Instances	n	Tp $10^{-3}s$	Tc $10^{-3}s$	*Speedup*							
				Number of processors							
				2	3	4	5	6	7	8	9
ftv55H	56	27	10	0.00	0.00	0.18	0.26	0.28	0.29	0.30	0.31
ftv55L	56	76	10	0.00	0.00	0.00	0.40	0.56	0.74	0.81	0.76
ftv70H	71	82	12	0.00	0.00	0.73	0.71	0.71	0.75	0.74	0.79
ftv70L	71	175	12	0.00	0.00	0.64	1.25	1.47	1.10	1.06	1.08
kro124pH	100	356	13	1.43	1.95	2.56	2.69	2.88	3.32	3.33	3.43
kro124pL	100	641	13	1.60	2.22	2.66	3.04	3.49	3.78	4.22	4.63
rbg323H	323	28866	20	1.62	2.65	3.28	3.80	4.55	5.21	5.69	6.44
rbg323L	323	23440	20	1.79	2.63	3.26	3.83	4.43	4.85	5.38	5.93

Table 2. Parallel memetic algorithm performance for the 90%-reduction.

Instances	n	Tp $10^{-3}s$	Tc $10^{-3}s$	*Speedup*							
				Number of processors							
				2	3	4	5	6	7	8	9
ftv55H	56	62	10	1.04	1.32	1.48	1.64	1.73	1.77	1.80	1.83
ftv55L	56	22	10	0.58	0.69	0.78	0.80	0.83	0.80	0.83	0.84
ftv70H	71	145	12	1.30	1.81	2.05	2.39	2.58	2.79	2.73	3.04
ftv70L	71	45	12	0.81	1.02	1.17	1.22	1.28	1.33	1.28	1.32
kro124pH	100	355	13	1.59	2.19	2.84	3.30	3.45	3.80	4.12	4.47
kro124pL	100	118	13	1.11	1.59	1.89	2.09	2.22	2.47	2.53	2.66
rbg323H	323	26052	20	1.91	2.68	3.38	3.84	4.23	4.84	5.39	5.97
rbg323L	323	5184	20	1.72	2.49	3.18	3.64	4.14	4.75	5.34	5.91

time spent doing local search and, consequently, the greater will be the *speedup*. In this work, the instances with 323 jobs had the best *speedup* values.

As future works we shall extend this approach to other problems in order to check if such results hold. A detailed investigation on the behavior of the *asynchronicity* to the proposed algorithm is also relevant.

Acknowledgements

This work was supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (*CAPES - Brazil*), Fundação de Amparo à Pesquisa do Estado de São Paulo (*FAPESP - Brazil*), and CNPq. The authors also wish to thank Eder Nicoletti Mathias and Celso Maciel da Costa by providing us with all the computational resources necessary for this work. P.M.'s work is supported by CNPq, Proc. 52.1100/01-1.

References

[1] L. Buriol, P. França, and P. Moscato. A new memetic algorithm for the asymmetric traveling salesman problem. *Journal of Heuristics*, 10(5):483–506, 2004.

[2] E. Cantú-Paz. Designing efficient master-slave parallel genetic algorithms. Technical Report 97004, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 1997.

[3] E. Cantú-Paz. A survey of parallel genetic algorithms. Technical Report 97003, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, 1997.

[4] E. Cantú-Paz and D. Goldberg. Efficient parallel genetic algorithms: theory and practice. *Computer methods in applied mechanics and enginneering*, 186:221–238, 2000.

[5] J. Du and J. Leung. Minimizing total tardiness on one machine is NP-hard. *Mathematics of Operations Research*, 15:483–495, 1990.

[6] P. França, J. Gupta, A. Mendes, P. Moscato, and K. Veltink. Metaheuristic approaches for the pure flowshop manufacturing cell problem. *Computers & Industrial Engineering*, 48(3):491–506, 2005.

[7] P. França, A. Mendes, and P. Moscato. A memetic algorithm for the total tardiness single machine scheduling problem. *European Journal of Operational Research*, 132–1:224–242, 2000.

[8] M. Gorges-Schleuter. Asparagos: An asynchronous parallel genetic optimization strategy. In *Third International Conference of Genetic Algorithms*, page 422, 1989.

[9] R. Graham, E. Lawler, J. Lenstra, and A. Rinooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[10] S. Graves. A review of production scheduling. *Operations Research*, 29:646–675, 1981.

[11] Y. Lee, K. Bhaskaran, and M. Pinedo. A heuristic to minimize the total weighted tardiness with sequence-dependent setups. *IIE Transactions*, 29:45–52, 1997.

[12] A. Mendes, P. França, and P. Moscato. Fitness landscape for the total tardiness single machine scheduling problem. *Neural Network World*, 2(2):165–180, 2002.

[13] A. Mendes, F. Müller, P. França, and P. Moscato. Comparing meta-heuristic approaches for parallel machine scheduling problems with sequence-dependent setup times. *Production Planning & Control*, 13(2):143–154, 2002.

[14] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms. Technical Report C3P 826, Caltech Concurrent Computation Program, 1989.

[15] N. Raman, R. Rachamadugu, and F. Talbot. Real time scheduling of an automated manufacturing center. *European Journal of Operations Research*, 40:222–242, 1989.

[16] P. Rubin and G. Ragatz. Scheduling in sequence dependent setup enviroment with genetic search. *Computers and Operations Research*, 22–1:85–99, 1995.

[17] K. Tan and R. Narasimhan. Minimizing tardiness on a single processor with a sequence-dependent setup times: a simulated annealing approach. *OMEGA - International Journal of Management Science*, 25–6:619–634, 1997.