S. B. Ghosn, F. Drouby and H. M. Harmanani, A Parallel Genetic Algorithm for the Open-Shop Scheduling Problem Using Deterministic and Random Moves, International Journal of Artificial Intelligence, vol. 14, no. 1, pp. 130-144, 2016.

A Parallel Genetic Algorithm for the Open-Shop Scheduling Problem Using Deterministic and Random Moves

Steve Bou Ghosn
Department of Computer Science, Westfield State University,
Westfield, MA 01085
steve_j777@yahoo.com

Fouad Drouby and Haidar M. Harmanani
Department of Computer Science and Mathematics
Lebanese American University
Byblos, 1401 2010, Lebanon
fooad2@yahoo.com
haidar@lau.edu.lb

ABSTRACT

This paper investigates the use of parallel genetic algorithms in order to solve the open-shop scheduling problem*. The method is based on a novel implementation of genetic operators that combines the use of deterministic and random moves. The method is implemented using MPI on a* Beowulf *cluster. Comparisons using the* Taillard *benchmarks give favorable results for this algorithm.*

Keywords: Open-shop scheduling, parallel genetic algorithms.

2000 Mathematics Subject Classification: 49K05, 49K15, 49S05.
ACM Subject Classification: Bio-inspired approaches: Genetic algorithms; Approximation algorithms analysis: Scheduling algorithms; Problem Solving, Control Methods, and Search: Scheduling;

1 Introduction

The open-shop scheduling problem consists of a set of n jobs, $\mathcal{J} = \{J_1, J_2, ..., J_n\}$, that are to be processed on a set of m machines, $\mathcal{M} = \{M_1, M_2, ..., M_m\}$. Each job consists of m operations, and each machine can process at most one job at a time. Each operation $o_{i,j}$ must be processed by machine M_i for $p_{i,j} \geq 0$ time units. The objective is to find a schedule for the operations on the machines that minimizes the makespan, C_{max}, that is, the time from the beginning of the first operation until the end of the last operation (Lawler, Lenstra, Kan and Shmoys, 1993). An optimal finish time schedule is one that has the least finish time among all schedules. It has been shown that one can establish the following lower bound for the optimal finish time schedule (Pinedo, 1995):

$$\mathcal{L} = \max\left\{\max_j \sum_{i=1}^{m} p_{ij}, \max_i \sum_{j=1}^{n} p_{ij}\right\} \tag{1.1}$$

where m is the number of machines, and n is the number of jobs.

Jobs	*(Processing Time, Machine)*			
Job 1	(54, M$_3$)	(34, M$_1$)	(61, M$_4$)	(2, M$_2$)
Job 2	(9, M$_4$)	(15, M$_1$)	(89, M$_2$)	(70, M$_3$)
Job 3	(38, M$_1$)	(19, M$_2$)	(28, M$_3$)	(87, M$_4$)
Job 4	(95, M$_1$)	(34, M$_3$)	(7, M$_2$)	(29, M$_4$)

Table 1: A 4x4 *Taillard* benchmark instance for the OSSP

The first part in equation 1.1 deals with the maximum completion time among all jobs that are to be processed while the second part refers to the maximum completion time for the jobs allocated to a given machine. In other words, the makespan is at least as large as the maximum workload on each of the m machines, and at least as large as the total amount of processing to be done on each of the n jobs (Pinedo, 1995). The optimal makespan for the open-shop scheduling problem can never be less than $\mathcal{L}$, but it is not necessarily equal to $\mathcal{L}$. A schedule is said to be *non-preemptive* if for each individual machine there is at most one tuple $< i, s(i), f(i) >$ for each job i to be scheduled, where $s(i)$ and $f(i)$ are the start time and the finish time of job i, respectively. A *preemptive* schedule is a schedule in which no restrictions are placed on the number of tuples per job per machine. The *non-preemptive* open-shop scheduling problem has been shown to be $\mathcal{NP}$-complete (Gonzalez and Sahni, 1976).
We illustrate the OSSP using the 4×4 Taillard benchmark shown in Table 1. The benchmark instance consists of 4 jobs and 4 machines. Figure 1 presents a possible schedule with an optimal makespan of 193.

1.1 Related Work

One of the earliest and most significant efforts related to the open-shop scheduling problem was reported by Gonzalez et al. (Gonzalez and Sahni, 1976) who showed that the problem is $\mathcal{NP}$-complete by reducing it to the partition problem. The authors proposed a linear-time algorithm with $m = 2$, and a polynomial time algorithm to find the optimal schedule for the preemptive open-shop scheduling problem. Some attempts were made at solving the open-shop scheduling problem using branch and bound techniques such as Brucker et al. (Brucker, Huring and Wostmann, 1997), who based their initial efforts on the resolution of a one machine problem with positive and negative time lags. Their later work aimed at fixing the disjunctions on the critical path using a heuristic solution. This method worked for most problem instances, but some problems of size 7 were still not solvable. Gueret et al. (Gueret and Prins, 1998) improved on Brucker's algorithm by using intelligent backtracking. Thus, when discarding a node, the algorithm backtracks not just to the direct parent, but further through the ancestors until locating a relevant section that would avoid excluding optimal solutions. Other approximate methods were reported. For example, Fang et al. (H.-L.Fang, Ross and Corne, 1993) proposed a genetic algorithm with an ordinal chromosomal representation, and adapted the approach to solve the open-shop scheduling problem (H.-L.Fang, Ross and Corne, 1994). The algorithm used a representation that produced valid schedules when altered by the genetic operators; thus allowing more accurate solutions in considerably less time. Khuri et al. (Khuri and Miryala, 1999) presented three different genetic approaches. One of these approaches is based on a genetic algorithm variation that uses a selfish gene technique. Another implementation was based on a hybrid GA implementation that proved to be better than the first

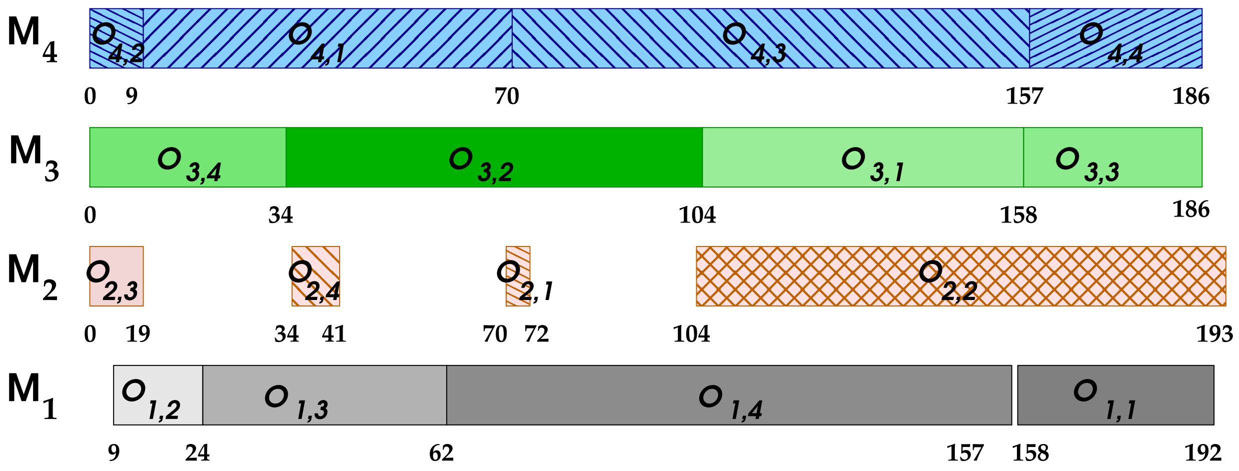


Figure 1: Optimal schedule for problem in Table 1 with makespan = 193

two proposed approaches. Liaw (Liaw, 1998) proposed an iterative improvement method for the open shop scheduling problem. The author, later proposed other efficient solution approaches based on a hybrid GA implementation (Liaw, 2000), a tabu search technique (Liaw, 1999b), and a simulated annealing algorithm (Liaw, 1999a). Recently, Blum (Blum, 2005) proposed a method based on a hybrid approach that combines ant colony optimization with beam search. Jiao et al. (Jiao and Yan, 2011) proposed a novel intelligent algorithm based on simulated annealing in order to solve the job-shop scheduling problem.

1.2 Genetic Algorithms

Genetic algorithms are approximate techniques to find solutions to complex optimization problems by mimicking natural evolution, and were used to solve various optimization problems such as the split delivery vehicle routing problem (SDVRP) (Khmelev and Kochetov, 2015), the optimization of the performance of electrical drives (Zăvoianu, Bramerdorfer, Lughofer, Silber, Amrhein and Klement, 2013), and fuzzy-controlled servo systems (Precup, David, Petriu, Preitl and Rădac, 2013). Genetic algorithms use a group of randomly initialized points, a *population*, in order to non-deterministically search the problem space. The population is characterized by the fact that each individual encodes all necessary problem parameters *(genes)*. The population is modified according to the natural evolution process following a parody of Darwinian principle of the survival of the fittest. Individuals are selected according to their quality to produce *offspring* and to propagate their genetic material into the next generation. Genetic algorithms employ an iterative process of *selection* and *recombination* that are executed in a loop for a fixed number of iterations where each iteration is called a generation. The selection process is intended to improve the average quality of the population by giving individuals of higher quality a higher probability of survival. The quality of an individual is measured by a fitness function. Each offspring undergoes a sequence of probabilistic transformations either by *inversion*, *crossover*, *mutation* or possibly other user defined operators. The process exploits new points in the search space by providing a diversity of the population and avoiding premature convergence to a single local optimum. The iterative process of selection and combination of “good” individuals should yield even better ones, until a solution is found or a stopping criterion is met.

1.3 Problem Description

This paper investigates the non-preemptive open shop scheduling problem using a parallel genetic algorithm motivated by the need to develop efficient and fast solutions. The problem is formally stated as follows (Garey and Johnson, 1979):

> Given a number of machines $m \in Z^+$, a set of jobs $\mathcal{J}$, each $j \in \mathcal{J}$ consisting of m operations $o_{i,j}$ with $1 \leq i \leq m$ (with $o_{i,j}$ to be executed by machine i), and for each operation a length $l_{i,j} \in N$, find a collection of one-machine schedules $f_i : \mathcal{J} \rightarrow N, 1 \leq i \leq m$, such that $f_i(j) > f_i(j') + l_{i,j'}$ such that for each $j \in \mathcal{J}$ the intervals $[f_i(j), f_i(j) + l_{i,j})$ are all disjoint and the completion time of the schedule is minimized:
>
> $$\max_{1 \leq i \leq m, j \in \mathcal{J}} f_i(j) + l_{i,j}$$

The proposed algorithm exploits parallelism using message passing on a Beowulf cluster. The algorithm combines the use of random and deterministic moves in order to efficiently explore the design space.
The remainder of the paper is organized as follows. Section 2 formulates the genetic algorithm for the open shop scheduling problem, and describes the chromosomal representation, the initial population, the fitness function, and the genetic operators. The parallel genetic algorithm is discussed in section 3 while experimental results are presented in section 4. We *conclude* with remarks in section 5.

2 OSSP Parallel Genetic Algorithm

The proposed parallel algorithm starts with a number of machines, and a set of jobs and corresponding tasks. The algorithm generates, through a sequence of random and deterministic operations, a set of compact schedules. The algorithm is implemented as cooperating sequential genetic algorithms, and is based on a single program multiple data (SPMD) model. The algorithm uses message passing in order to allow various processors to operate independently on isolated sub-populations of the individuals, periodically sharing its “best” individuals. Processors are connected through an *Ethernet* network and use the Message Passing Interface (MPI). In what follows, we describe the parallel genetic algorithm with reference to the benchmark in Table 1.

2.1 Chromosomal Encoding

Genetic algorithms work with a coding of the parameter set and not the parameters themselves. Therefore, one requirement when employing GAs in order to solve a combinatorial optimization problem is to find an efficient representation of the solution in the form of a chromosome.
We propose a chromosomal representation that is based on a vector of $(m \times n)$ genes, where m is the number of machines and n the total number of jobs. A gene corresponds to an operation to be performed on a certain machine while a chromosome corresponds to a schedule. The values of the genes are a permutation of integer values between 0 and $(m \times n) - 1$. A gene V is interpreted as the $(V \ mod \ m)$ operation of the $(V \ div \ m)$ job. This is illustrated in Figure 2 where the chromosome is interpreted as follows: schedule the third task of the first job, followed by the first task of the second job, ...

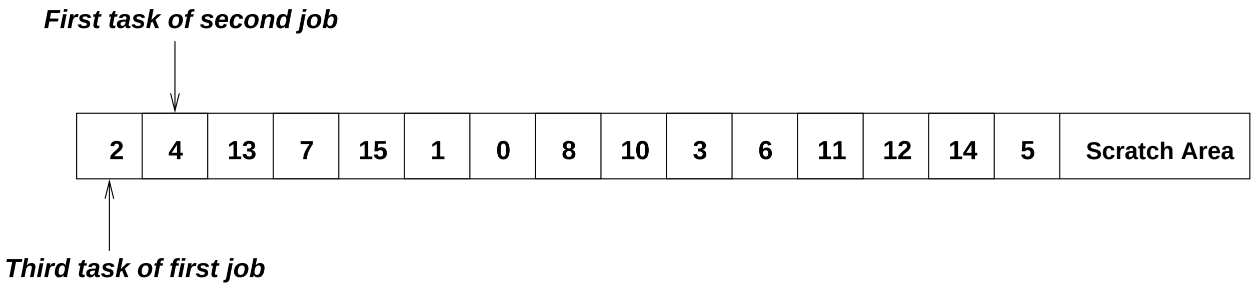


Figure 2: Sample chromosome encoding and interpretation

Finally, a chromosome includes a temporary storage area, *the scratch area*, that is used to pass hints to the genetic operators by including candidate genes to swap as well as a gene's position at which the schedule becomes invalid.

2.2 Initial Population

The initial population is important as it affects the quality as well as the time needed to converge to a final solution. The initial population is created randomly by selecting the values of the genes to be unique permutations between 0 and $(n \times m) - 1$, where n is the number of jobs and m is the number of machines. The algorithm ensures that all $m \times n$ genes have different values. The scratch area is initialized to 0.

2.3 Fitness Function

The fitness function measures the fitness of each chromosome in the population. The fitness of an individual is crucial for the transmission of its gene information to the next generation. The fitness function is given by the following:

$$F = \frac{1}{C_{max}} \tag{2.1}$$

2.4 Selection and Reproduction

Reproduction is the artificial version of natural selection, a Darwinian survival of the fittest. There are many approaches to selecting parent chromosomes for reproduction. We have attempted various techniques, and observed that *roulette wheel* selection worked well for our algorithm. The selection algorithm retained the best chromosome as well. It should be noted that reproduction occurs *locally*, within the same sequential process. However, migration among parallel sub-populations improves the population quality by injecting the best chromosome in the current sequential population. We describe the parallel population migration in section 2.5.

2.5 Parallel Computational Model

In order to parallelize the open shop scheduling algorithm, the population was partitioned into sub-populations that evolve independently using a sequential genetic algorithm. Interaction among sub-populations is allowed through *migration*. The parallel execution model is a more realistic

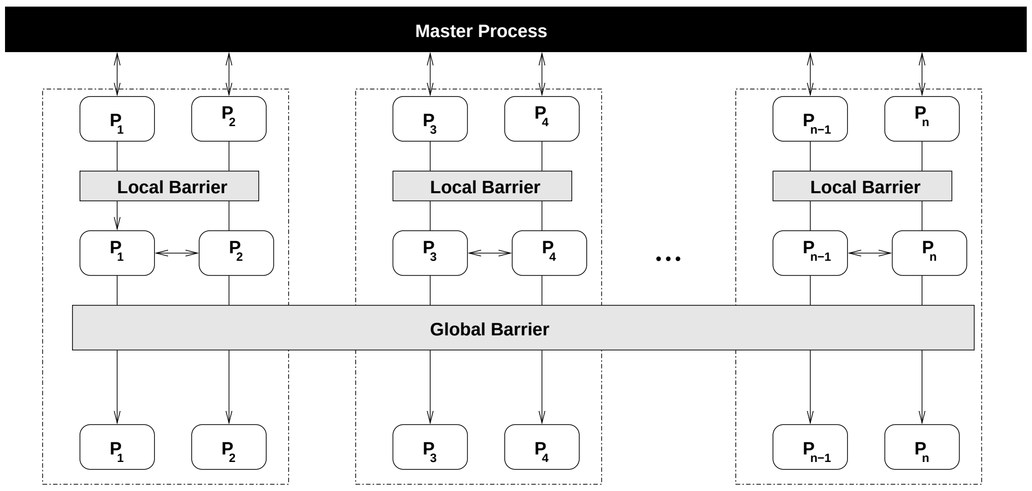


Figure 3: Parallel Model

simulation of natural evolution in which communities are isolated but occasionally interact through migration or cross-communities mating (Sadiq and Youssef, 2000).
Parallel migration implies that changes in a population come not only from inheriting portions of one's parents' genes, albeit with occasional random mutations, but also from the introduction of new species into the population. In nature, this movement between sub-populations is often a survival response that is responsible for several tasks including the selection and sending of emigrants in addition to the reception and integration of these immigrants. Note that population migration introduces communication overhead.
There are two common models for migration (Chipperfield and Fleming, 1996; Wilkinson and Allen, 1996), the *island model* where individuals are allowed to be sent to any other sub-populations and the *stepping stone model* that limits migration only to neighboring sub-populations. Both models have advantages and disadvantages. The obvious advantage is that, except for the communication of "best individuals" that occurs only once every k generations, both are embarrassingly parallel[1] in nature. Cohoon (Cohoon, Hedge, Martin and Richards, 1987) observed that when a substantial number of individuals migrated between isolated sub-populations, new solutions were found shortly after the migration occurred.
In order to reduce the communication overhead while ensuring a global migration of individuals, we have adopted a hybrid approach that is based on the following. We allocate the sequential GA (slaves) to groups where the number of processes within a group is set by the user. Next, a group of processes share their best chromosomes every G_N generation. On the other hand, every L_N generations, the best chromosome is broadcasted to all processes where $G_N \ll L_N$. This approach gives the genetic algorithm a chance to reconstruct different layouts starting from the best rather than from the initialized generations. Note that all processes are synchronized through barriers.

2.6 Deterministic Optimization Operation

Minimizing C_{max} is equivalent to minimizing the idle time on each machine. Idle time gaps are due either to moves that are caused by the genetic operators, or to conflicts since the m-processor

[1]An embarrassingly parallel computation is an ideal parallel computation that can be divided into completely independent tasks that can be executed simultaneously

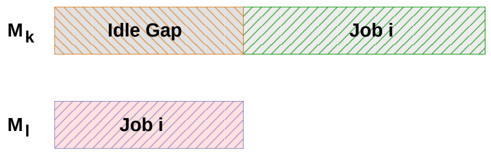


Figure 4: An example idle gap due to a scheduling conflict

schedules must be such that no job is processed simultaneously on two or more machines. The first problem is easily solved by scheduling the operations using an as soon as possible methodology, after a genetic operator move. Thus, the operation is inserted in the earliest available time slot. The second problem requires to locally deconstruct the schedule and reconstruct it again. In what follows, we describe the *ReduceGap* operation.

2.6.1 ReduceGap Operation

The *ReduceGap* operation improves the performance of the genetic algorithm by deterministically analyzing the schedule and passing hints to the genetic operators. Thus, if there is a ”large” enough idle time gap in a machine’s schedule, the operation minimizes the gap by recommending swapping the constraining operations. The *ReduceGap* operation results with pairs of values that indicate positions in the data area of the chromosome to be swapped. The *ReduceGap* operation communicates the information to the genetic operators via the *scratch area* as a pair consisting of an operation to adjust, and a corresponding constraining causal operation.

2.6.2 ReduceGap Measure

In order to guide the optimization process, the *ReduceGap* operation uses the lower bound $\mathcal{L}$ in equation 1.1. The rational is that if we are striving to find an ”optimum” finish time then $\mathcal{L}$ can give a tight measure. The operation checks the difference, δ, between the finish time for each machine, and the lower bound $\mathcal{L}$ in order to determine whether the idle time on each machine is acceptable. If a given schedule contains any machine that is exceeding the allowed idle scheduling time then the idle time gaps for that particular machine are analyzed, and the necessary moves are recommended in order to minimize one of those idle time gaps.

2.6.3 ReduceGap Algorithm

The *ReduceGap* operator selects a gap in the machine schedule where δ exceeds the allowed time. The algorithm uses two approaches based on a probability. The first approach traverses the specific schedule from right to left and selects the first problematic gap. The second approach selects randomly a gap in the machine. A problematic gap is gap that overlaps with a specific job on one machine, and that is followed by the same job but on a different machine (Figure 4). The position of the operation that is preceded by the gap as well as the position of another random gene are recorded in the scratch area. Finally, the *ReduceGap* operator determines the gene’s position at which the schedule makespan *degrades* causing the machine to exceed the current best finish time. This particular piece of information is stored in the last position in the scratch area.

Algorithm 1 Master Process

```
 1: function MASTER_PROCESS(JOBS, MACHINES)
 2:     Get the jobs, machines, and tasks
 3:     Get population size ($N$) and number of generations ($N_g$)
 4:     Get the number of parallel slave processes to be spawned
 5:     Spawn all slaves
 6:     Broadcast $N$, $N_g$, jobs, machines, and tasks
 7: end function
```

Algorithm 2 Slave Process

```
 1: function PARALLEL_GENETIC_OSSP()
 2:     Get the jobs, machines, and tasks from the master
 3:     Get population size ($N$) and number of generations ($N_g$)
 4:     Get the operator's probabilities from the master
 5:     Evaluate the fitness of chromosomes using equation 2.1
 6:     for ($i = 0$; $i <$ NumberOfGenerations; $i$++) do
 7:         ReduceGap()
 8:         Randomly select a chromosome from current_pop for mating
 9:             Apply mutation with probability $P_m$
10:             Select either a deterministic move or a random move subject to probability $P_d$
11:         Select two chromosomes from current_pop for mating
12:             Apply crossover with probability $P_{xover}$
13:             Select either a deterministic move or a random move subject to probability $P_d$
14:         Evaluate the cost and the fitness using equation 2.1
15:         Repeat the above forming a new population, new_pop.
16:         Perform local selection
17:         if ($i == L_N$) then
18:             local_migration()          ▷ Broadcast the best chromosomes to neighboring processes
19:         else if ($i == G_N$) then
20:             global_migration()         ▷ Broadcast the best chromosome
21:         end if
22:     end for
23: end function
```

2.7 Genetic Operators

In order to explore the design space, we use two genetic operators, *mutation*, and *crossover*. The genetic operators are applied randomly as well as deterministically. It was observed that following a purely deterministic approach constrains the solution, and may result with a premature convergence to a local optima. Using a random approach slows down the convergence time, and the algorithm would require a substantial amount of time in order to converge to a good solution. Therefore, it is very important to find a balance between both approaches in order to obtain the best results. The genetic operators are applied iteratively with their corresponding probabilities as shown in Algorithm 2.

2.7.1 Mutation

The mutation operator selects two positions in a chromosome for possible swap. The positions are selected either *randomly* or *deterministically*, based on a probability. In the first case, the algorithm selects two random genes between 0 and $(m \times n) - 1$, and swaps their positions. In the second case, the mutation operator uses the information stored in the scratch area, and that was determined by the *ReduceGap* operator. The information consists of an operation that it affects the *makespan* of the schedule, and another operation that causes the first operation to be scheduled

Parameter	Value
Crossover	25%
Mutation	75%
Determinism	60%
Population Size	400
Number of Generation	1000

Table 2: Parameter Settings

as such. For each of these pairs the mutator chooses randomly one of the hinted operations, and moves it to another random position in a similar way to the random approach.

2.7.2 Crossover

We have implemented two different variations for the crossover operation. The first performs a uniform crossover while ensuring the solution feasibility. The second approach uses the information stored in the last gene of the scratch area. We use a parameter that can be tuned in order to determine how often to use each approach. The operator creates two new offspring chromosomes by swapping all the genes of the parents subject to the following steps:

1. Extract the value of the current gene in chromosome *one*.
2. If the value is not found in the new chromosome, then insert the value in the next available empty gene. Otherwise, repeat step 1 and select the next gene until a value that is not present in the new chromosome is found.
3. Repeat the same procedure for chromosome *two*.
4. Repeat the above procedure until the new offspring have all value permutations between 0 and $(n \times m) - 1$.

The second approach performs crossover using the position that was determined by the *Reduce-Gap* operator, and that was stored in the last gene of the scratch area. This information basically indicates the position at which the schedule loses a potential quality solution. To illustrate why this information is important, consider the case where a crossover point is randomly chosen such that it is less than or equal to the position where the schedule loses its quality solution. Applying crossover at that section would result with a low quality schedule. However, if we select a crosspoint that is before the critical position and we cross it over with the section from the second chromosome, we could be generating a potential good candidate solution. In this implementation we can also randomly decide from which of the two chromosomes to take the cross point, and which one to use as the start of the new chromosome.

3 Parallel Algorithm

The parallel algorithm was implemented using the C language, and is shown in Figure 3, based on a SPMD master-slave model. The master algorithm (Algorithm 1) reads the jobs and the machines

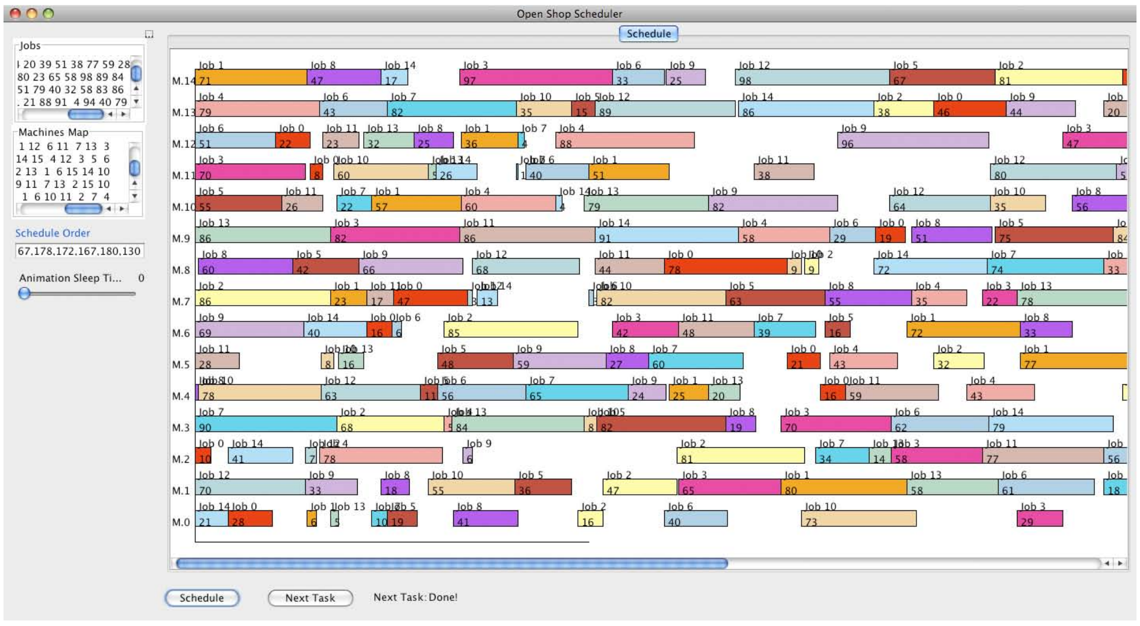


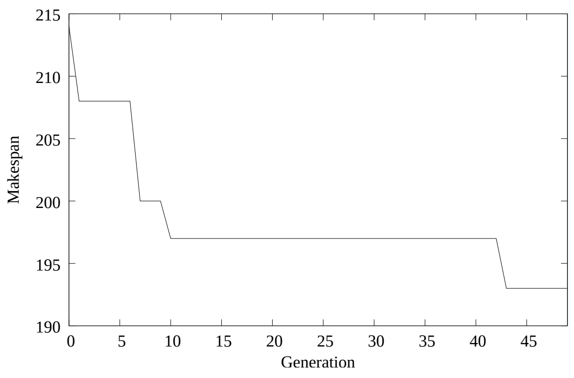
Figure 5: Schedule for an instance of problem 20×20 with a makespan of 1292

as well as the problem parameters. This includes the migration parameters as well as the number of slaves to be spawned. The master organizes the slave processes into groups and distributes the subproblems among the slaves which do the actual computational work. The master is also responsible for performing some analysis before the final results are produced.
The slave algorithm (Algorithm 2) receives the number of jobs, the number of machines, and the corresponding tasks from the master. Furthermore, the master sends to the slaves the genetic algorithm parameters such as the operators probabilities, the population size, and the number of generations. Every slave generates its own local initial population and executes its own sequential genetic algorithm. The GA operations are applied iteratively, and every operator has a probability associated with it. At the end of each generation, the slave program computes the fitness value for each chromosome, finds the best chromosome.
Every L_N iterations, local population stabilizes. The algorithm introduces a new competition chromosome from neighboring processes (processes that are running on the same machine). Every G_N iterations, all slaves broadcast their best chromosome to all other slaves. In both cases, the best chromosome migration simulates a change in the environment and helps the sub-population elements to rapidly evolve to adapt to this new change.

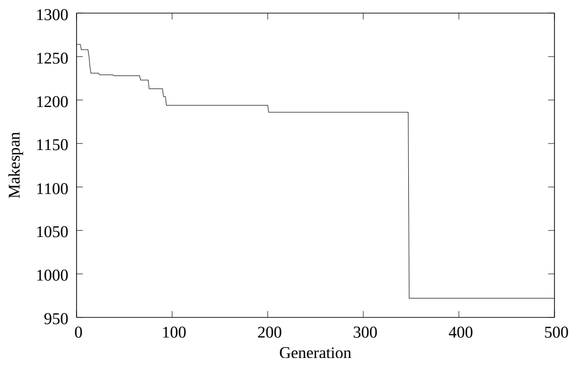
4 Experimental Results

4.1 Parameter Tuning

In order to apply the parallel genetic algorithm, several parameters have to be adjusted. Most important are the control parameters such as crossover rate, mutation rate, population size N, and the number of generations N_g. We have performed experiments on a set of problems with a crossover and a mutation rates that varied between 0.1 and 1. The population size was varied between 100 and 5000 while the number of generations was varied between 50 and 5000. It was determined experimentally that for the problems at hand, a population size of 400 and number of



(a) 4×4-1 Taillard Benchmark



(b) 15×15-1 Taillard Benchmark

Figure 6: Number of generations vs. *makespan* for the 4×4-1 (Figure 1), and for the 15×15-1 Taillard Benchmark

generation of 1000 was sufficient to achieve good solutions. Furthermore, it was observed that a crossover probability, P$_{xover}$, of 0.25 and a mutation probability, P$_{m}$, of 0.75 performed quite well. Within the mutation itself, it is also important to regulate the amount of deterministic moves. It was determined empirically that the best value for this parameter is a probability of 0.6.

4.2 Benchmark Results

The proposed method was implemented using C and MPI on an five-nodes *Beowulf* cluster, and was evaluated using the *Taillard* benchmarks (Taillard, 1994). The GUI tool was implemented using Java and includes a mechanism to trace through the schedule. The *Taillard* benchmarks consist of a set of problem instances that are easy to generate; however, their size is large and correspond to industrial problems. For the open shop, *Taillard* proposes problems where the number of machines and jobs is the same $(m \times m)$, and uses 6 different sizes (4, 5, 7, 10, 15, 20) where each problem has 10 instances.
Tables 3 and 4 show the results obtained after running each of the *Taillard* Benchmark instances using the determined parameters (Table 2). The GA was executed 30 times for each instance. We

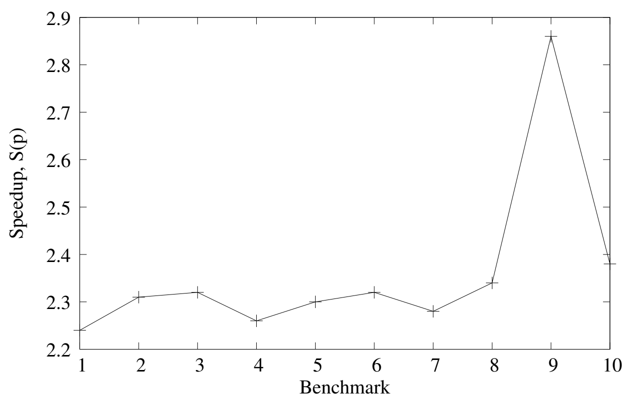


Figure 7: Speedup for the $20 \times 20 - 1$ benchmark

Instance	Opt. (L_B)	Ours	Difference (%)	Average
4 x 4 - 1	193	193	0	193
4 x 4 - 2	236	236	0	236
4 x 4 - 3	271	271	0	271
4 x 4 - 4	250	250	0	250
4 x 4 - 5	295	295	0	295
4 x 4 - 6	189	189	0	189
4 x 4 - 7	201	201	0	201
4 x 4 - 8	217	217	0	217
4 x 4 - 9	261	261	0	261
4 x 4 - 10	217	217	0	217
5 x 5 - 1	300	300	0	300
5 x 5 - 2	262	262	0	262
5 x 5 - 3	323	323	0	323
5 x 5 - 4	310	310	0	310
5 x 5 - 5	326	326	0	326
5 x 5 - 6	312	312	0	312
5 x 5 - 7	303	303	0	303
5 x 5 - 8	300	300	0	300
5 x 5 - 9	353	353	0	353
5 x 5 - 10	326	326	0	326
7 x 7 - 1	435	447	2.75	452
7 x 7 - 2	443	456	2.93	468
7 x 7 - 3	468	470	0.42	496
7 x 7 - 4	463	468	1.08	482
7 x 7 - 5	416	419	0.72	435
7 x 7 - 6	451	461	2.21	475
7 x 7 - 7	422	430	1.89	455
7 x 7 - 8	424	426	0.47	441
7 x 7 - 9	458	465	1.52	477
7 x 7 - 10	398	408	2.51	420

Table 3: Taillard Benchmark Results

present the best solution found, and the average of all solutions found through the 30 runs. We show the solution for an instance of the 20×20 benchmark in Figure 5 using our GUI interface. Figure 7 illustrates the speedup factor for all 20×20 instances, and varied in this case between 2.28 and 2.89 using *five* processors. It should be noted that the setup time as well as the communication time were considered when computing the speedup factor. Figures 6a and 6b depict the *makespan* of the best chromosome in a population of 400. It is clear how quickly our method converges to a good solution before it saturates. Furthermore, the sudden improvement in solution quality is noticeable and this is due to migration.

5 Conclusion

This paper presented a parallel genetic algorithm to solve the *open-shop scheduling problem*. The method is based on an interesting implementation of genetic operators that combines the use of deterministic and random moves. The method was implemented using MPI on a *Beowulf* cluster. Favorable results using the *Taillard* benchmarks were reported.

Instance	Opt. (L_B)	Ours	Difference (%)	Average
10 x 10 - 1	637	673	5.65	686
10 x 10 - 2	588	601	2.21	619
10 x 10 - 3	598	610	2.00	632
10 x 10 - 4	577	586	1.55	605
10 x 10 - 5	640	659	2.96	668
10 x 10 - 6	538	556	3.34	577
10 x 10 - 7	616	632	2.59	639
10 x 10 - 8	595	610	2.52	625
10 x 10 - 9	595	615	3.36	635
10 x 10 - 10	596	617	3.52	633
15 x 15 - 1	937	972	3.73	1159
15 x 15 - 2	918	992	8.06	1197
15 x 15 - 3	871	914	4.93	1131
15 x 15 - 4	934	945	0.96	1167
15 x 15 - 5	946	1023	8.13	1192
15 x 15 - 6	933	986	5.68	1165
15 x 15 - 7	891	957	7.41	1143
15 x 15 - 8	893	938	5.03	1136
15 x 15 - 9	899	978	8.79	1119
15 x 15 - 10	902	962	6.65	1157
20 x 20 - 1	1155	1244	7.70	1268
20 x 20 - 2	1241	1344	8.29	1379
20 x 20 - 3	1257	1315	4.61	1347
20 x 20 - 4	1248	1330	6.57	1392
20 x 20 - 5	1256	1342	6.84	1352
20 x 20 - 6	1204	1292	7.30	1301
20 x 20 - 7	1294	1384	6.95	1404
20 x 20 - 8	1169	1292	10.52	1316
20 x 20 - 9	1289	1326	2.87	1383
20 x 20 - 10	1241	1317	6.12	1345

Table 4: Taillard Benchmark Results (Continued)

References

Blum, C. 2005. Beam-aco-hybridizing ant colony optimization with beam search: an application to open shop scheduling, *Computers and Operations Research* **32**(6): 1565–1591.

Brucker, P., Huring, J. and Wostmann, B. 1997. A branch and bound algorithm for the open-shop problem, *Discrete Applied Mathematics* **76**: 43–59.

Chipperfield, A. and Fleming, P. 1996. *Parallel and Distributed Computing Handbook*, McGraw Hill, chapter Parallel Genetic Algorithms, pp. 1118–1193.

Cohoon, J., Hedge, S., Martin, M. and Richards, D. 1987. Punctuated equilibria: A parallel genetic algorithm, *Proceedings of the Second International Conference on Genetic Algorithm*.

Garey, M. and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman.

Gonzalez, T. and Sahni, S. 1976. Open shop scheduling to minimize finish time, *Journal of the Association for Computing Machinery* **23**(4): 665–679.

Gueret, C. and Prins, C. 1998. Classical and new heuristics for the open-shop problem: a computational evaluation, *European Journal of Operational Research* **107**: 306–314.

H.-L.Fang, Ross, P. and Corne, D. 1993. A promising genetic algorithm approach to job shop scheduling, rescheduling and open-shop scheduling problems, *Proceedings of the Fifth International Conference in Genetic Algorithms*, pp. 375–382.

H.-L.Fang, Ross, P. and Corne, D. 1994. A promising hybrid GA/heuristic approach for open-shop scheduling problems, *ECAI Proceedings of the 11th European Conference on Artificial Intelligence, Amsterdam, The Netherlands*, John Wiley & Sons, Ltd, pp. 590–594.

Jiao, B. and Yan, S. 2011. A cooperative co–evolutionary quantum particle swarm optimizer based on simulated annealing for job shop scheduling problem, *International Journal of Artificial Intelligence* **7**(11): 232–247.

Khmelev, A. and Kochetov, Y. 2015. A hybrid local search for the split delivery vehicle routing problem, *International Journal of Artificial Intelligence* **13**(1): 147–164.

Khuri, S. and Miryala, S. 1999. Genetic algorithms for solving open shop scheduling problems, *Proceedings of the 9th Portuguese Conference on Artificial Intelligence, EPIA '99 Évora, Portugal, September 21–24.*, Springer, pp. 357–368.

Lawler, E. L., Lenstra, J., Kan, A. R. and Shmoys, D. 1993. Sequencing and scheduling: Algorithms and complexity, *Handbooks in Operations Research and Management Science: Logistics of Production and Recovery* **4**: 445–522.

Liaw, C.-F. 1998. An iterative improvement approach for the nonpreemptive open shop scheduling problem, *European Journal of Operational Research* **111**: 509–517.

Liaw, C.-F. 1999a. Applying simulated annealing to the open shop scheduling problem, *IIE Transactions* **31**(5): 457–465.

Liaw, C.-F. 1999b. A tabu search algorithm for the open shop scheduling problem, *Computers and Operations Research* **26**(2): 109–126.

Liaw, C.-F. 2000. A hybrid genetic algorithm for the open shop scheduling problem, *European Journal of Operational Research* **124**: 28–42.

Pinedo, M. 1995. *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall.

Precup, R.-E., David, R.-C., Petriu, E., Preitl, S. and Rădac, M.-B. 2013. Fuzzy logic–based adaptive gravitational search algorithm for optimal tuning of fuzzy controlled servo systems, *IET Control Theory & Applications* **7**(1): 99–107.

Sadiq, S. and Youssef, H. 2000. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*, Wiley-IEEE Computer Society Press.

Taillard, E. 1994. Benchmarks for basic scheduling problems, *European Journal of Operational Research* **64**: 278–285.

Wilkinson, B. and Allen, M. 1996. *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers*, Prentice Hall.

Zăvoianu, A.-C., Bramerdorfer, G., Lughofer, E., Silber, S., Amrhein, W. and Klement, E. P. 2013. Hybridization of multi-objective evolutionary algorithms and artificial neural networks for optimizing the performance of electrical drives, *Engineering Applications of Artificial Intelligence* **26**(8): 1781–1794.