An Empirical Performance Evaluation of a Parameter-free Genetic Algorithm for Job-Shop Scheduling Problem

Shouichi Matsui and Seiji Yamada

Abstract— **The Job-Shop Scheduling Problem (JSSP) is well known as one of the most difficult NP-hard combinatorial optimization problems. Several GA-based approaches have been reported for the JSSP. Among them, there is a parameter-free genetic algorithm (PfGA) for JSSP proposed by Matsui et al., based on an extended version of PfGA, which uses random keys for representing permutation of operations in jobs, and uses a hybrid scheduling for decoding a permutation into a schedule. They reported that their algorithm performs well for typical benchmark problems, but the experiments were limited to a small number of problem instances. This paper shows the results of an empirical performance evaluation of the GA for a wider range of problem instances. The results show that the GA performs well for many problem instances, and the performance can be improved greatly by increasing the number of subpopulations in the parallel distributed version.**

I. Introduction

In the Job-Shop Scheduling Problem (JSSP), n jobs have to be processed on m different machines. Each job J_i consists of a sequence of tasks $T_{i,1}, \cdots, T_{i,m}$ that have to be completed during an uninterrupted time period of length $p_{i,j} \in \mathbf{N}$ on a given machine ($M(T_{i,j}) \in \{1, \cdots, m\}$). A schedule is an allocation of tasks to time intervals on machines. The goal in the job-shop scheduling problem is to find the sequence of n jobs to be completed on m machines such that the makespan (finish time of the last operation) is minimized.

The job-shop scheduling problem is well known as one of the most difficult NP-hard combinatorial optimization problems. Several approaches have been reported to the JSSP for a few decades. Among them work on Genetic Algorithms (GAs) for solving the JSSP has a history of only two decades, but they perform well compared with other approaches.

Davis [7] was the first to use Genetic Algorithms to solve the JSSP, and many GAs have been proposed and analyzed (e.g., [1], [4], [8], [11], [12], [14]–[16], [20], [23]–[26]).

Other heuristics such as Simulated Annealing (SA) and Tabu Search (TS), and exact techniques as branch and bound have also been developed for solving the JSSP. Jain and Meeran provide a good description of these techniques [9].

However, the tuning of genetic parameters has to be performed by trial and error, making optimization by GA *ad hoc*. There have been many research projects for *self-adaptive* GA because such adaptation can tune the parameters while solving a given problem. Nevertheless, it is a very time-consuming task to design an optimal GA in an adaptive way because we have to perform computation many times by trial and error. To address this problem, Sawai et al. have proposed the Parameter-free Genetic Algorithm (PfGA), for which no control parameters for genetic operation need to be set in advance [10], [17], [18]. The performance of the PfGA has been reported to be high compared with other GAs [10], [17], [18].

Matsui et al. have proposed an extended parameter-free GA for the JSSP [13], and reported that the GA performed well without tedious parameter-tuning. Their GA is promising, but they only tested a small number of problem instances.

This paper reports the results of an empirical evaluation of the GA to a wider range of problem instances. The simulation results show that the GA performs well for many problem instances, and the performance can be improved greatly by increasing the number of subpopulations in the parallel distributed version.

S. Matsui is with System Engineering Research Laboratory (SERL), Central Research Institute of Electric Power Industry (CRIEPI), 2-11-1 Iwado-kita, Komae-shi, Tokyo 201-8511, Japan; email: matsui@criepi.denken.or.jp.

S. Yamada is with National Institute of Informatics (NII), 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan; email: seiji@nii.ac.jp.

II. The GA for Empirical Evaluation

This section gives a very brief overview of the GA [13] for the empirical evaluation. The proposed GA uses the random keys [15] for permutation representation, and hybrid scheduling for decoding the permutation.

A. Chromosome Representation

There are two ways of representing a schedule: indirect and direct. In *indirect* representation, the chromosome contains an encoded schedule. A schedule builder is used to transform the chromosome into a feasible schedule. The indirect representations range from traditional binary representations [14] to domain-specific knowledge representation [2].

In *direct* representation, the chromosome directly represents the production schedule. Bruns [6] shows many ways to deal with direct representations. Direct representation performs efficiently on production scheduling, incorporates domain-specific operations easily, but also requires domain-specific recombination operators.

B. Types of Feasible Schedules

There are four types of feasible schedules in JSSP as follows;

> **(1) inadmissible:** Inadmissible schedules contain excess idle time, and they can be improved by forward-shifting operations until no excess idle time exists.

1-4244-1340-0/07/$25.00

(2) semi-active: Semi-active schedules contain no excess idle time, but they can be improved by shifting some operations to the front without delaying others.
(3) active: Active schedules contain no idle time and no operation can be finished earlier without delaying other operations. The optimal schedule is guaranteed to be an active schedule.
(4) non-delay: Non-delay schedules are active schedules, in which operations are placed into the schedule such that the machine idle time is minimized. No machine is kept idle if some operation can be processed.

The random keys representation [3], [15] encodes a solution with random numbers. These values are used as the *sort keys* to decode the permutation.

C. Permutation Decoding

Permutations can be decoded into *semi-active*, *active*, or *non-delay* schedules [5]. $\mu_i(k) = j\,(1 \leq k \leq m_i)$ specifies that M_j is the machine that processes the k-th operation of job J_i, where m_i is the number of operations of J_i. The k-th operation of job J_i processed on machine $M_{\mu_i(k)}$ is denoted as o_{ik}. The required processing time of o_{ik} is denoted by p_{ik}. Let t_{ik} be the starting time of operation o_{ik}, let t_{hl} and p_{hl} be the starting time and the required processing time of the operation of job J_h which precedes o_{ik} on its machine, then

$$t_{ik} = \max(t_{i,k-1} + p_{i,k-1}, t_{hl} + p_{hl}). \tag{1}$$

Hybrid schedules [5] can be produced in a flexible way by introducing a parameter $\delta \in [0, 1]$. The setting of δ can be thought of as defining a bound on the length of time a machine is allowed to remain idle. At the extreme $\delta = 0$ produces non-delay schedules while $\delta = 1$ produces active schedules. A schedule is constructed according to a given permutation as follows.

step 1: Build the set of all beginning operations $A := \{o_{i1} \,|\, 1 \leq i \leq n\}$.
step 2: Find an operation o' from A with the earliest completion time $t' + p'$.
step 3: Determine the machine M' of o' and build the set B from all operations in A which are processed on M'.
step 4: Find an operation o'' from B with the earliest possible starting time t''.
step 5: Delete operations in B in accordance with parameter δ such that
$B := \{o_{ik} \in B \,|\, t_{ik} \leq t'' + \delta((t' + p') - t'')\}$.
step 6: Select operation o^*_{ik} from B which occurs leftmost in the permutation and delete it from A.
step 7: Append operation o^*_{ik} to the schedule and calculate its starting time.
step 8: If o^*_{ik} has successor $o^*_{i,k+1}$, insert it into A.
step 9: If A is empty, terminate. Otherwise, go to step 2.

D. Parameter-free Genetic Algorithm

The Parameter-free Genetic Algorithm (PfGA) proposed by Sawai et al. [10], [17], [18] is a novel GA inspired by the "disparity theory of evolution." The idea of the theory is based on different mutation rate in double strands of DNA. The PfGA is a very compact and fast adaptive search algorithm based on the variable-size of population taking a dynamic but delicate balance between exploration, and exploitation, i.e., global search and local search. The PfGA is not only simple and robust, but also does not need to set almost all genetic parameters in advance that need to be set up in other GAs. The performance of PfGA is reported to be very high for function optimization problems [10], [17], [18].

The PfGA procedure is as follows [17].

step 0: Let S be the whole population, and S' be the subpopulation.
step 1: Select one individual randomly from S, and add it to S'.
step 2: Select one individual randomly from S, and add it to S'.
step 3: Select two individuals P_1, P_2 randomly from S' and perform multiple-point crossover to generate two children C_1, C_2. In the crossover n crossover points (n is a random integer $0 < n < L$, where L is the length of chromosome) are randomly selected.
step 4: For one randomly selected child, perform mutation. In the mutation, a randomly chosen portion of chromosome is inverted, i.e., bit-flipped, at random.
step 5: Select one to three individuals among P_1, P_2, C_1, C_2 depending on the cases (described later), and feed them back to S'.
step 6: If the terminating condition is satisfied, then terminate.
step 7: If $|S'| > 1$ then go to step 3, otherwise go to step 2.

For the selection operation in step 5, the fitness values f of P_1, P_2, C_1, C_2 are compared, and the selection is done according to the following rules.

case 1: If the fitness values of children $f(C_1)$ and $f(C_2)$ are better than those of the parents, then C_1 and C_2 and $\underset{P_i}{\operatorname{argmax}}(f(P_1), f(P_2))$ are left in S'.
case 2: If the fitness values of children $f(C_1)$ and $f(C_2)$ are worse than those of parents, then only $\underset{P_i}{\operatorname{argmax}}(f(P_1), f(P_2))$ is left in S'.
case 3: If the fitness values of either $f(P_1)$ or $f(P_2)$ is better than those of children, then $\underset{P_i}{\operatorname{argmax}}(f(P_1), f(P_2))$ and $\underset{C_i}{\operatorname{argmax}}(f(C_1), f(C_2))$ are left in S'.
case 4: In all other situations, $\underset{C_i}{\operatorname{argmax}}(f(C_1), f(C_2))$ is preserved and then one individual randomly chosen from S is added to S'.

E. Distributed Parallel PfGA

The distributed parallel processing of PfGA is as follows [17]. Let us assume that there are N subpopulations $S_i (i = 1, 2, \cdots, N)$, and each subpopulation evolves as shown in the previous section. Migration among the subpopulations occurs when a better individual is produced in a subpopulation.

Sawai et al. proposed many migration methods, master-slave types and uniform-distributed types. The migration method tested in this paper, which is named as UD1, is as follows. If a better child is generated in a subpopulation, the best child C_i is copied to other randomly chosen subpopulations as an emigrant. When other subpopulations receive the immigrant, they add it as an individual and eliminate the worst individual among all individuals in the subpopulation.

F. Real-coded PfGA (Rc-PfGA)

Matsui et al. have extended the PfGA to a real coded version. A gene of the original PfGA takes the value of 0 or 1, i.e., a chromosome is a bit-string. The real coded PfGA is an extension where a chromosome is a string of real numbers.

The mutation operator of Rc-PfGA replaces the value of a gene by a random number uniformly distributed in the predefined range.

The random keys representation of permutation can be expressed naturally with the Rc-PfGA.

G. Chromosome

The chromosome of the GA is a sequence of real numbers, and its length is $2 \times n \times m$. The chromosome is divided into two parts, the first part represents the random keys, and the second part represents the δs of the hybrid schedule.

III. Computational Results

The GA is tested by the benchmark problems from ORLib [22]. In the paper [13], as the problem instances, we only used the the same set that was used by Norman and Bean [15]. The instances were limited to FT10, FT20, and LA01–LA40, therefore we tested a wider range of instances, namely 10 tough problems, ORB01–ORB10, SWV01–SWV20, and TA01–TA30.

The GA was run for each problem instance using 50 different random seeds. The maximum number of fitness evaluations was set to 1,000,000 as the base case.

A. 10 Tough Problems

1) Serial or Parallel: Matsui et al. showed that the performance of the GA improved as they increased the number of subpopulations (N) in distributed parallel version, because the total number of fitness evaluations increased [13]. But they did not show whether the GA converged or not. Therefore we tested the GA with different number of fitness evaluations using the serial version and the parallel version. We tested 10 tough problems as the problem instances, ABZ7, ABZ8, ABZ9, LA21, LA24, LA25, LA27, LA29, LA38, and LA40.

The results are shown in Table I. In Table I, S2, S4, and S8 correspond to the cases where the number of fitness evaluations was multiplied by 2, 4, and 8, and D2, D4, and D8 correspond to the cases where the number of subpopulations (N) was set to $N = \{2, 4, 8\}$. The μ, σ, and 'best' represents the average, standard deviation, and best makespan over 50 runs respectively. The numbers in boldface shows the optimal makespan.

Table I shows the followings.

- In most cases, as we increase the number of fitness evaluations, the average makespan is reduced, and also the standard deviation of makespan decreases in both the serial and the parallel version. But the best makespan does not always shorten as we increase the number of fitness evaluations.
- The difference of average makespan between the serial and the parallel version is small. We can verify that the difference is not statistically significant by statistical tests (t-test, Welch's test, and non-parametric tests).
- The best makespan by the serial version is smaller than that by the parallel version in LA21, LA24, LA25, and LA27 for all cases. In other problems, the best makespan by the parallel version is smaller when we compare S8 and D8, except for ABZ8 and ABZ9.

The average computation time of the serial version per run, for example, was 237 (sec.) for LA29 (20×10), 225 (sec.) for LA40 (15×15) on a PC with an AMD Athlon MP 1800+ (1.53GHz).

2) Number of Subpopulations: In the paper [13] they only reported the cases where the number of subpopulations (N) was set to $N = \{2, 4, 8\}$. We ran the cases $N = \{1, 4, 8, 16, 32, 64\}$ to see the performance improvement by increasing the number of subpopulations. Table II shows the results. In Table II 'blb' represents the best lower bound [9], and $n \times m$ represents the problem size.

Fig.1 and 2 show the relative error of the best span to the best upper bound and the average span to the best upper bound respectively. In the column 'Best bounds', the number in parentheses denotes the best lower bound, the number without parentheses denoted the best upper bound or the optimal makespan.

From Tables II and Fig.1, 2, we can conclude the followings.

- As we increase the number of subpopulations, the average makespan is reduced, but the best makespan does not always shorten. Except for the case that is relatively easy (LA21, LA24, and LA25) the best span does not shorten as we increase the number of populations.
- The relative error to the best upper bound is very small.
- The relative error of the best span to the best upper bound decreases roughly proportional to $1/N^{1/8}$. The relative error of the average makespan to the best upper bound decreases in a similar way.

TABLE I

COMPARISON OF SERIAL AND PARALLEL VERSION (10 TOUGH PROBLEMS)

	ABZ7(20×15)			ABZ8(20×15)			ABZ9(20×15)		
	S2	S4	S8	S2	S4	S8	S2	S4	S8
μ	692.7	688.2	685.7	713.8	710.8	708.1	725.2	719.0	718.5
σ	5.24	5.64	6.1	5.8	6.8	6.4	9.1	7.9	7.2
best	681	677	672	704	696	696	703	706	705
	D2	D4	D8	D2	D4	D8	D2	D4	D8
μ	691.1	687.3	685.4	714.7	709.1	703.9	721.8	718.9	716.4
σ	6.8	5.6	4.7	7.2	5.9	6.7	8.2	7.8	7.4
best	676	674	677	700	698	689	706	703	703

	LA21(15×10)			LA24(15×10)			LA25(15×10)			LA27(20×10)		
	S2	S4	S8	S2	S4	S8	S2	S4	S8	S2	S4	S8
μ	1069.0	1065.9	1069.4	958.1	952.9	950.2	990.2	989.0	986.5	1267.7	1264.2	1263.2
σ	12.0	9.2	11.5	10.2	10.1	10.2	6.9	7.2	6.4	7.0	7.2	6.4
best	**1046**	1047	1047	939	938	938	**977**	**977**	**977**	1253	1240	1245
	D2	D4	D8	D2	D4	D8	D2	D4	D8	D2	D4	D8
μ	1070.2	1066.5	1062.2	957.1	954.1	951.9	991.4	988.0	986.0	1267.8	1265.2	1263.5
σ	11.4	9.9	7.8	10.4	10.7	8.5	8.0	7.3	6.7	5.5	7.0	6.4
best	1047	1047	1050	939	938	941	**977**	**977**	**977**	1255	1250	1250

	LA29(20×10)			LA38(15×15)			LA40(15×15)		
	S2	S4	S8	S2	S4	S8	S2	S4	S8
μ	1203.7	1199.0	1196.6	1251.0	1248.5	1244.3	1255.5	1253.1	1247.1
σ	11.9	14.2	13.1	16.9	19.1	15.2	11.8	11.1	7.1
best	1184	1169	1167	1220	1207	1212	1233	1233	1228
	D2	D4	D8	D2	D4	D8	D2	D4	D8
μ	1203.5	1199.8	1191.0	1251.3	1244.1	1237.7	1254.9	1249.7	1245.8
σ	12.8	13.6	10.0	16.6	15.1	14.3	11.4	9.1	8.1
best	1171	1175	1167	1219	1203	1216	1228	1228	1231

Fig. 1. Relative error of the best span to the best upper bound (10 tough problems)

Fig. 2. Relative error of the average span to the best upper bound (10 tough problems)

3) Number of Migration: The number of migrations was very small for all problems, below 0.01% of the total number of fitness evaluations. Therefore, the overhead of communication by MPI (Message Passing Interface) [19] among subpopulations was negligible, and the computation time was almost equal to that of the serial version. Because there is no significant difference of average makespan between the serial and parallel version, we decided to test the parallel version for other problem instances.

TABLE II

SIMULATION RESULTS FOR 10 TOUGH PROBLEMS

			$N=1$			$N=2$			$N=4$			$N=8$		
Prob.	$n \times m$	blb [9]	best	μ	σ	best	μ	σ	best	μ	σ	best	μ	σ
ABZ7	20×15	656	677	696.1	7.5	676	691.1	6.8	674	687.3	5.6	677	685.4	4.7
ABZ8	20×15	645	700	716.4	7.5	700	714.7	7.2	698	709.1	5.9	689	703.9	6.7
ABZ9	20×15	661	708	730.0	9.1	706	721.8	8.2	703	718.9	7.8	703	716.4	7.4
LA21	15×10	1046	1053	1071.8	11.9	1047	1070.2	11.4	1047	1066.5	9.9	1050	1062.2	7.8
LA24	15×10	935	938	959.8	12.7	939	957.1	10.4	938	954.1	10.7	941	951.9	8.5
LA25	15×10	977	978	996.8	10.2	**977**	991.4	8.0	**977**	988.0	7.3	**977**	986.0	6.7
LA27	20×10	1235	1260	1273.7	9.6	1255	1267.8	5.5	1250	1265.2	7.0	1250	1263.5	6.4
LA29	20×10	1152	1184	1207.6	12.6	1171	1203.5	12.8	1175	1199.8	13.6	1167	1191.0	10.0
LA38	15×15	1196	1208	1260.9	21.2	1219	1251.3	16.6	1203	1244.1	15.1	1216	1237.7	14.3
LA40	15×15	1222	1240	1258.5	9.3	1228	1254.9	11.4	1228	1249.7	9.1	1231	1245.8	8.1
			$N=16$			$N=32$			$N=64$					
Prob.	$n \times m$	blb [9]	best	μ	σ	best	μ	σ	best	μ	σ			
ABZ7	20×15	656	669	681.4	4.2	669	677.6	4.8	667	675.6	3.5			
ABZ8	20×15	645	680	700.3	7.0	688	696.9	5.7	680	693.0	5.3			
ABZ9	20×15	661	692	710.0	6.3	696	707.2	5.7	692	703.9	4.5			
LA21	15×10	1046	1047	1058.0	6.9	**1046**	1054.8	6.0	**1046**	1051.5	3.8			
LA24	15×10	935	938	948.8	7.5	937	943.1	4.9	935	940.3	3.1			
LA25	15×10	977	**977**	983.6	4.9	**977**	982.6	5.1	**977**	979.7	3.2			
LA27	20×10	1235	1244	1260.0	6.7	1241	1256.0	6.9	1241	1254.2	6.2			
LA29	20×10	1152	1164	1186.4	10.4	1164	1185.0	8.9	1163	1175.2	9.0			
LA38	15×15	1196	1205	1228.3	10.6	1202	1223.9	11.5	1202	1215.8	7.6			
LA40	15×15	1222	1224	1240.7	7.1	1226	1240.6	4.6	1228	1238.2	5.2			

B. ORB Problem Instances

Table III and Fig. 3 show the results for problem instances ORB01–ORB10. Due to space limitation, we only show the cases for $N = \{1, 16, 32, 64\}$ in Table III, but the relative error to the optimal makespan is shown for all cases $N = \{1, 2, 4, 8, 16, 32, 64\}$ in Fig. 3.

Table III and Fig. 3 show the followings.

- Because the size of the ORB problems is small, 10×10, the GA can find the optimal makespan with $N = 1$ except for ORB02, ORB05, and ORB06. The best makespan can be found in all problem instances when $N \geq 8$.
- The relative error of the average makespan to the best makespan decreases as we increase N. It is very small, below 1.0%, when $N \geq 16$. The relative error decreases in a similar way as of the 10 tough problems.
- The GA can always find the optimal makespan when $N \geq 2$ in ORB10, when $N \geq 32$ in ORB09, and when $N = 64$ in ORB08.

Fig. 3. Average relative error to the optimal makespan (ORB problems)

C. TA and SWV Problem Instances

Table IV and Fig. 4 show the results for problem instances TA01–TA30, and Table V and Fig. 5 show the results for problem instances SWV01–SWV15 (hard SWV problems). Due to space limitation, we only show the cases for $N = \{1, 16, 32, 64\}$ in Table IV and Fig. 5, but the relative error to the best upper bound is shown for all cases $N = \{1, 2, 4, 8, 16, 32, 64\}$ in Fig. 4 and Fig. 5.

- The GA can find the makespan that is equal to the optimal or the best upper bound in TA02 and TA03 when we increase the number of subpopulations.
- As we increase the number of subpopulations, the average makespan is reduced, but the best makespan does not always shorten. The relative error decreases in a similar way as of the 10 tough problems and ORB problem instances.
- The average relative error to the best upper bound is small enough in TA01–TA10, below 1.5%, when $N =$

TABLE III

SIMULATION RESULTS FOR ORB PROBLEMS

			$N=1$		$N=16$		$N=32$		$N=64$	
Prob.	$n \times m$	Opt [9]	Best	μ	Best	μ	Best	μ	Best	μ
ORB01	10×10	1059	**1059**	1087.0	**1059**	1066.3	**1059**	1062.3	**1059**	1059.6
ORB02	10×10	888	889	892.7	**888**	889.0	**888**	889.0	**888**	888.8
ORB03	10×10	1005	**1005**	1027.7	**1005**	1014.9	**1005**	1011.4	**1005**	1007.3
ORB04	10×10	1005	**1005**	1016.8	**1005**	1011.1	**1005**	1009.9	**1005**	1007.9
ORB05	10×10	887	889	893.0	**887**	889.1	**887**	888.8	**887**	888.5
ORB06	10×10	1010	1012	1029.3	**1010**	1019.6	**1010**	1016.7	**1010**	1014.5
ORB07	10×10	397	**397**	401.2	**397**	397.7	**397**	397.2	**397**	**397.0**
ORB08	10×10	899	**899**	920.3	**899**	906.2	**899**	902.8	**899**	901.8
ORB09	10×10	934	**934**	939.9	**934**	934.7	**934**	**934.0**	**934**	**934.0**
ORB10	10×10	944	**944**	944.5	**944**	**944.0**	**944**	**944.0**	**944**	**944.0**

TABLE IV

SIMULATION RESULTS FOR TA PROBLEMS

		Best	$N=1$		$N=16$		$N=32$		$N=64$	
Prob.	$n \times m$	bounds [21]	Best	μ	Best	μ	Best	μ	Best	μ
TA01	15×15	1231	1265	1288.2	1249	1263.1	1248	1259.6	1241	1254.3
TA02	15×15	1244	1252	1280.9	**1244**	1264.3	**1244**	1261.0	**1244**	1255.8
TA03	15×15	1218	1228	1256.8	1219	1232.4	**1218**	1229.3	**1218**	1226.4
TA04	15×15	1175	1191	1222.6	1181	1195.0	1181	1191.3	1181	1188.4
TA05	15×15	1224	1253	1272.4	1236	1252.6	1236	1246.4	1228	1244.9
TA06	15×15	1238	1249	1286.8	1248	1263.7	1248	1258.6	1246	1255.4
TA07	15×15	1227	1250	1264.0	1236	1247.3	1233	1242.6	1228	1240.6
TA08	15×15	1217	1233	1266.8	1221	1242.0	1224	1238.2	1223	1233.1
TA09	15×15	1274	1314	1342.9	1296	1316.4	1292	1310.2	1284	1305.6
TA10	15×15	1241	1244	1299.0	1244	1265.1	1244	1256.3	1244	1246.5
TA11	20×15	1361(1323)	1416	1464.0	1395	1430.0	1408	1426.8	1398	1419.9
TA12	20×15	1367(1351)	1428	1471.6	1406	1435.4	1408	1430.8	1396	1421.8
TA13	20×15	1342(1282)	1407	1443.1	1378	1408.1	1361	1399.6	1371	1393.5
TA14	20×15	1345	1374	1405.5	1352	1378.0	1355	1373.6	1351	1365.8
TA15	20×15	1340(1304)	1412	1461.3	1388	1416.1	1379	1412.6	1384	1402.1
TA16	20×15	1360(1302)	1417	1462.8	1397	1426.9	1400	1420.2	1380	1413.8
TA17	20×15	1462	1498	1546.5	1496	1519.9	1494	1514.1	1490	1508.3
TA18	20×15	1396(1369)	1468	1519.1	1461	1489.1	1449	1477.6	1443	1468.9
TA19	20×15	1335(1297)	1419	1454.2	1398	1419.9	1390	1415.2	1382	1405.8
TA20	20×15	1351(1318)	1414	1453.3	1399	1420.0	1389	1414.1	1380	1406.8
TA21	20×20	1644(1539)	1729	1773.6	1687	1728.7	1688	1723.1	1685	1716.1
TA22	20×20	1600(1511)	1699	1736.2	1677	1697.3	1665	1689.7	1646	1683.3
TA23	20×20	1557(1472)	1635	1683.9	1595	1641.9	1598	1636.9	1589	1625.1
TA24	20×20	1647(1602)	1712	1767.8	1695	1730.3	1698	1722.0	1693	1716.2
TA25	20×20	1595(1504)	1665	1715.6	1651	1682.8	1644	1671.3	1650	1668.3
TA26	20×20	1645(1539)	1728	1766.3	1700	1728.7	1704	1724.7	1699	1719.7
TA27	20×20	1680(1616)	1787	1827.3	1737	1777.9	1725	1761.3	1741	1760.1
TA28	20×20	1614(1614)	1698	1739.4	1658	1691.5	1653	1685.9	1654	1676.4
TA29	20×20	1625(1514)	1689	1730.5	1655	1693.3	1656	1685.9	1653	1678.1
TA30	20×20	1584(1473)	1656	1717.8	1639	1679.0	1638	1667.7	1638	1661.5

64. But in TA11–TA30 it varies from 1.5% to 5.3% even in $N = 64$. The best relative error to the best upper bound is from 0.4% to 3.6% when $N = 64$.

- The relative error to the best upper bound is larger than that of ORB cases. This is due to the size of the problems. The size of ORB problems is 10×10 and it is 15×15, 20×15, or 20×20 in TA problems. The length of the chromosomes used in the GA is proportional to

Fig. 4. Average relative error to the best upper bound (TA problems)

TABLE V
SIMULATION RESULTS FOR HARD SWV PROBLEMS

		Best	$N=1$		$N=16$		$N=32$		$N=64$	
Prob.	$n \times m$	bounds [9]	Best	μ	Best	μ	Best	μ	Best	μ
SWV01	20×10	1407	1489	1545.2	1456	1490.5	1442	1478.5	1459	1479.9
SWV02	20×10	1475	1549	1588.7	1499	1551.9	1496	1543.6	1503	1537.8
SWV03	20×10	1398(1396)	1480	1522.4	1433	1475.6	1436	1468.8	1433	1465.2
SWV04	20×10	1483(1450)	1559	1601.0	1509	1555.0	1519	1545.7	1509	1535.4
SWV05	20×10	1424	1532	1576.5	1475	1523.7	1469	1514.0	1470	1506.7
SWV06	20×15	1678(1591)	1817	1879.5	1782	1825.6	1778	1809.6	1752	1797.9
SWV07	20×15	1620(1446)	1737	1788.1	1711	1742.1	1688	1731.4	1687	1720.3
SWV08	20×15	1763(1640)	1939	2006.3	1865	1934.7	1881	1919.6	1851	1904.3
SWV09	20×15	1663(1604)	1809	1872.1	1762	1812.1	1741	1795.7	1742	1783.8
SWV10	20×15	1767(1631)	1902	1940.0	1849	1886.1	1830	1870.4	1809	1862.4
SWV11	50×10	2991(2983)	3327	3468.5	3240	3355.3	3241	3323.4	3212	3300.5
SWV12	50×10	3003(2972)	3429	3509.6	3330	3397.0	3297	3369.4	3274	3349.8
SWV13	50×10	3104	3462	3567.0	3348	3450.1	3374	3423.6	3310	3386.5
SWV14	50×10	2968	3274	3385.9	3162	3259.9	3150	3236.9	3115	3189.0
SWV15	50×10	2904(2885)	3327	3426.7	3210	3318.0	3190	3275.8	3189	3254.9

Fig. 5. Average relative error to the optimal makespan (Hard SWV problems)

the size of problems, $2 \times n \times m$, therefore the length (L) is 200 in ORB problems, and 450, 600, or 800 in TA problems. More detailed analysis is shown in the discussion section.

D. Discussions

The relative error to the best upper bound varies among problem instances. The size of the problems is a reason of the difference. The length of the chromosomes in the GA is proportional to the size of problem instances, $2 \times n \times m$, therefore the length (L) varies from 200 (10×10 problems) to 1000 (50×10 problems).

The size of search space is approximately proportional to Z^L where Z is a constant determined by the floating-point number representation format. Because we used the IEEE 32-bit representation of floating-point number format in these experiments, $Z = 2^{24}$. Therefore the size of search space varies from $(2^{24})^{200} \simeq 10^{1444}$ to $(2^{24})^{1000} \simeq 10^{7224}$.

The difference of search space is huge, but the performance does not vary so greatly, and this supports the promising applicability of the PfGA to wider class of JSSP instances.

As the experiments show, we can easily improve the solution by just increasing the number of subpopulations. This is also a good feature of the PfGA when applied to the real-world applications. There are dual-/quad-core CPUs that are not expensive, it is very easy to have PC-based cluster system, and it is fairly easy to have a multi-processor system today. Therefore it is very easy to increase the number of subpopulations, and have a better solutions using distributed version of PfGA.

IV. Conclusions and Future Work

Our empirical performance evaluation showed that the parameter-free GA for JSSP can attain high quality solutions for a wide range of benchmark problems without tedious parameter tuning. Though the decreasing rate of the average relative error to the best upper bound is not high, it is roughly proportional to $1/N^{1/8}$ (N is the number of subpopulations), the parallel version can attain better results as we increase the number of subpopulations. The average makespan is reduced as we increase the number of fitness evaluations. If the total number of fitness evaluations is the same for the serial and parallel version, there is no statistically significant difference in both versions, therefore we can speed-up by using the parallel version. The performance increases as we increase the number of fitness evaluations in subpopulations, so we will test it for a wider class of benchmark problems.

References

[1] K.S. Amirthagadeswaran and V.P. Arunachalam: "Improved solutions for job shop scheduling problems through genetic algorithm with a different method of schedule deduction," *The International Journal of Advanced Manufacturing Technology*, vol.28, no.5-6, pp.1433–3015, 2006

[2] S. Bagchi, S. Uckun, Y. Miyabe, and K. Kawarnura: " Exploring problem-specific recombination operators for job shop scheduling," *International Conf. Genetic Algorithms (ICGA-91)*, pp.10–17, 1991.

[3] J.C. Bean: "Genetics and random keys for sequencing and optimization," *ORSA Journal of Computing*, vol.6, no.2, pp.154–160, 1994.

[4] C. Bierwirth, D. Mattfeld, and H. Kopfer: "On permutation representations for scheduling problems," *Proc. 4th Parallel Problem Solving from Nature – PPSN IV*, pp.310–318, 1996.

[5] C. Bierwirth and D.C. Mattfeld: "Production scheduling and rescheduling with generic algorithms," *Evolutionary Computation*, vol.7, no.1, pp.1–17, 1999.

[6] R. Bruns: "Direct chromosome representation and advanced genetic operations for production scheduling," *International Conf. Genetic Algorithms (ICGA-93)*, pp.352–359, 1993.

[7] L. Davis: "Job shop scheduling with genetic algorithms," *International Conf. Genetic Algorithms (ICGA-85)*, pp.136–140, 1985.

[8] P. Fenton and P. Walsh: "A comparison of messy GA and permutation based GA for job shop scheduling," *Genetic And Evolutionary Computation Conference 2005*, pp.1593–1594, 2005.

[9] A.S. Jain and S. Meeran: "Deterministic job-shop scheduling: past, present and future," *European Journal of Operational Research*, vol.113, pp.390–434, 1999.

[10] S. Kizu, H. Sawai and H. Endo: "Parameter-free genetic algorithm: GA without setting genetic parameters," *Proc. 1997 International Symposium on Nonlinear Theory and its Application*, vol.2 of 2, pp.1273–1276, 1997.

[11] S. Kobayashi,I. Ono and M. Yamamura: "An efficient genetic algorithm for job shop scheduling problems," *Proc. Sixth International Conference on Genetic Algorithms*, pp.506-511, 1995.

[12] T.-K. Liu, J.-T. Tsai, and J.-H. Chou: "Improved genetic algorithm for the job-shop scheduling problem," *The International Journal of Advanced Manufacturing Technology*, vol.27, no.9-10, pp.1021–1029, 2006.

[13] S. Matsui, I. Watanabe, and K. Tokoro: "Real-coded parameter-free genetic algorithm for job-shop scheduling problems," *Proc. 7th Parallel Problem Solving from Nature – PPSN VII*, pp.800–810, 2002.

[14] R. Nakano and T. Yamada: "Conventional genetic algorithm for job shop scheduling," *Proc. 3rd International Conference on Genetic Algorithms*, pp.474–479, 1991.

[15] B. Norman and J.C. Bean: "Random keys genetic algorithm for job shop scheduling," Engineering Design & Automation, vol.3, no.2, pp.145–156, 1997.

[16] B.M. Ombuki and M. Ventresca: "Local search genetic algorithms for the job shop scheduling problem," vol.21, no.1, pp.99–109, 2004.

[17] H. Sawai and S. Kizu: Parameter-free genetic algorithm inspired by "disparity theory "of evolution"," *Proc. 5th Parallel Problem Solving from Nature – PPSN V*, pp.702–711, 1998.

[18] H. Sawai, S. Kizu, and T. Endo: "Parameter-free genetic algorithm (PfGA)," *Trans. IEICE, Japan*, vol.J81-D-II, no.2, pp.450–452, 1998 (*in Japanese*).

[19] M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra: *MPI: The Complete Reference*, The MIT Press (1997)

[20] R. Storer, S. Wu, and R. Vaccari: "New search spaces for sequencing problems with application to job shop scheduling," *Management Science*, vol.38, pp.1495–1509, 1992.

[21] É. Taillard: "Summary of best known lower and upper bound of Taillard's instances," http://ina.eivd.ch/collaborateurs/etd/problemes.dir/ordonnancement.dir/ jobshop.dir/best_lb_up.txt, last updated Nov. 17, 2005 (2005).

[22] R.J.M. Vaessens: "Operations Research Library of Problems," Management School, Imperial College London, ftp://mscmga.ms.ic.ac.uk/pub/jobshop1.txt, 1996.

[23] M. Vázquez and D. Whitley: "A comparison of genetic algorithms for the static job shop scheduling problem," *Proc. 6th Parallel Problem Solving from Nature – PPSN VI*, pp.303–312, 2000.

[24] T. Yamada and R. Nakano: "A genetic algorithm applicable to large-scale job-shop problems," *Proc. 2nd Parallel Problem Solving from Nature – PPSN II*, pp.281–290, 1992.

[25] L. Wang and D.-Z. Zheng: "A modified genetic algorithm for job shop scheduling," *The International Journal of Advanced Manufacturing Technology*, vol.20, no.1, 2002.

[26] C. Wu, W. Xiang, Y. Liang, H. Pueh Lee, and C. Zhou: "A modified integer-coding genetic algorithm for job shop scheduling problem," *PRICAI 2004: Trends in Artificial Intelligence*, pp.373–380, 2004.