PARALLEL HYBRID HEURISTICS FOR THE PERMUTATION FLOW SHOP PROBLEM

M.G. RAVETTI, C. RIVEROS, A. MENDES, M.G.C. RESENDE, AND P.M. PARDALOS

Abstract. This paper addresses the Permutation Flowshop Problem with minimization of makespan, which is denoted by $F|prmu|Cmax$. In the permutational scenario, the sequence of jobs has to remain the same in all machines. The Flowshop Problem (FSP) is known to be NP-hard when more than three machines are considered. Thus, for medium and large scale instances, high-quality heuristics are needed to find good solutions in reasonable time. We propose and analyse parallel hybrid search methods that fully use the computational power of current multi-core machines. The parallel methods combine a memetic algorithm (MA) and several iterated greedy algorithms (IG) running concurrently. Two test scenarios were included, with short and long CPU times. The tests were conducted on the set of benchmark instances introduced by Taillard in 1993, commonly used to assess the performance of new methods. Results indicate that the use of the MA to manage a pool of solutions is highly effective, allowing the improvement of the best known upper bound for one of the instances.

1. Introduction

The Flowshop Problem (FSP) is a scheduling problem in which n jobs have to be processed by m machines. The problem is to find the sequence of jobs for each machine to minimize the maximum completion time, also known as makespan [1]. This problem is NP-Hard for $m > 3$ [2, 3]. Several papers in the literature address this problem, proposing models, heuristics, and bounds. Dannenbring [4] tested several heuristics. Nawaz, Enscore, and Ham [5] presented a polynomial time algorithm (NEH), finding very good results, considering the complexity of their heuristic. Indeed, NEH is still one of the best polynomial time heuristics for this problem. Another good constructive approach is the N&M algorithm, propose by Nagano and Moccelin in 2002 [6]. Taillard [7] presented an improvement in the complexity of the NEH algorithm, a heuristic based on tabu search, and a useful characterization of the distribution of the objective function. Taillard [8] proposed a series of test instances with good quality upper bounds. The running time required by Taillard's tabu search heuristic was not given and he focused on solution quality. The set of instances proposed in that work became a benchmark and is currently used to assess the performance of new methods for the FSP. Ben-Daya and Al-Fawzan [9] implemented and tested an improved variant of Taillard's tabu search, reporting times and comparing their performance with Ogbu and Smith's simulated

Date: October 8, 2010.

Key words and phrases. Metaheuristics, memetic algorithms, flowshop problem, combinatorial optimization, scheduling.

AT&T Labs Research Technical Report. Research supported by CNPq, NSF and Air Force Grants.

annealing algorithm [10]. However, they did not match all of Taillard's results for large instances. Stützle [11] presented and tested an Iterated Local Search (ILS) heuristic obtaining good results.

Ruiz and Maroto [12] compared 25 methods, from very basic ones, such as Johnson's algorithm [13], to the more sophisticated ones, such as tabu search and simulated annealing. The results of their study concluded that NEH was the best polynomial-time heuristic, while Stützle's ILS [11] and Reeves's genetic algorithm [14] were the best metaheuristic-based heuristics. Ruiz et al. [15] proposed a new memetic algorithm for this problem, obtaining improved results when compared with the ILS and tabu search. The same authors followed up with another paper in the same direction [16], proposing and testing two genetic algorithms and obtaining strong results. Agarwal, Collak, and Eryarsoy [17] implemented a heuristic improvement procedure based on adaptive learning and applied it to the NEH algorithm, leading to additional improvements. However, for larger instances, their results were of poor quality and their algorithm was computationally intensive. Then, Ruíz and Stützle [18] described a simple method called iterated greedy algorithm (IG) which produced good results, leading to six new upper bounds for the set of instances introduced by Taillard in 1993;

In [8, 9, 17] and the vast majority of papers dealing with the FSP, the problem of interest was a special case called Permutation Flowshop Problem (PFSP) in which the jobs have identical ordering sequences on all machines. This widespread approach is particularly useful because the space of possible solutions is considerably smaller than the non-permutational version of the FSP; and in general, good PFSP solutions represent good solutions for the non-permutation FSP.

Very few approaches consider a parallel or distributed scenario. More specifically, a recent publication [19] analyses the use of a computer cluster where each dual-core machine runs an IG algorithm. One core will be responsible for running the IG algorithm and the second core will deal with communication. At different stages of its execution each algorithm will broadcast its best solution.

In this paper, we consider the PFSP and propose a hybrid search method. It uses a Memetic Algorithm (MA) with an embedded IG algorithm. The MA collaborates with several independent IG methods, running with parameters that produce a more thorough search. MA and IGs run independently, in separate threads and in a collaborative fashion, exchanging solutions along the search. Even though the IG has already been presented in the literature, this work introduces a novel collaborative framework, and a specialized MA, with a structured population; an *ad-hoc* crossover operator; offspring acceptance policies; and a restart procedure triggered when populations lose diversity.

The paper is organized as follows. In Section 2 we describe the IG algorithm; its multi-threaded version; and the memetic algorithm along with all its elements. In Section 3, we present the computational experiments as well as a discussion about the results. Finally, in Section 4 we make some concluding remarks.

2. Methods

2.1. **IG and Multi-threaded IG (MIG) algorithms.** The IG algorithm was first presented in a scheduling environment in [18]. It starts with a random solution and for a number of iterations it performs a partial destruction of the sequence, followed by a greedy reconstruction and a local search. The algorithm will change

```
Method IG(InitialSolution)
begin
   π = InitialSolution;
   π = LocalSearch(π);
   π_b = π;
   while(Termination criterion not satisfied)
      π' = π;
      for i = 1  to destructionParam do
         π'= remove one job at random from π' and insert in π_R;
      endfor
      for i = 1  to destructionParam do
         π'= GreedyConstruction(π', π_R);
      endfor
      π''= LocalSearch(π');
      if (C_max(π'') < C_max(π)) then
         π= π'';
         if (C_max(π) < C_max(π_b)) then
            π_b= π;
         endIf
      elseIf (rnd ≤ exp{-C_max(π'') − C_max(π)/Temperature}) then
         π= π'';
      endIf
   endWhile
   return π_b;
end
```

FIGURE 1. Pseudo-code for the IG algorithm as in [18]. *rnd* is a random number distributed uniformly in the interval [0,1].

the current solution and use a Simulated Annealing-like acceptance criterion [20]. The basic configuration of the IG algorithm is shown in Figure 1.

The Multi-threaded IG (MIG) method is an extension of the IG algorithm, developed to run in a multi-threaded environment. In this case, each thread will run an independent IG algorithm, which is reset at regular intervals. The *termination criterion* is fixed to a number of iterations, *IG steps*. At the beginning of each new cycle, the algorithm will work on the overall best solution found so far, considering all the threads. Finally, the overall best solution is reported by the end of the processing time, Figure 2 summarizes the approach.

2.2. **MA+MIG approach.** Memetic Algorithm (MA) [20, 21] is a population-based search method, which uses evolution-inspired operators to find high-quality solutions. Our MA implementation has several especially developed elements, such as structure population, new crossovers, acceptance policies for new individuals, restart procedures, and finally a local search based on the IG algorithm. The MIG is a Multi-Threaded IG which runs concurrently with the MA. The two methods collaborate by exchanging solutions using a *migration pool* that is populated by the MA. Next, we present a pseudocode of the MA+MIG algorithm and clarify its main components.

```
Method MIG(nthreads, IG_steps)
begin
   π = NEH_heuristic();
   π = LocalSearch(π);
   π_global = π;
   /* Starting nthreads threads */
   for i=1  to nthreads do
      π_b = startThreadIG(i, π_global);
      if (π_b < π_global) then π_global = π_b;
   endfor
   return π_global;
end
```

FIGURE 2. Pseudo-code for the IG algorithm as in [18]. *rnd* is a random number distributed uniformly in the interval [0,1].

```
Method MA+MIG
begin
   createThreadMA();
   createThreadsIG(numThreads);
   /* MA thread concurrent to IG threads */
   initializePopulation();
   while(cpuTime < limitCPU)
      updatePopStructure();
      while(populationNotConverged())
         newSolution = generateOffspring();
         newSolution = mutate(newSolution);
         newSolution = localSearchIG(newSolution);
         acceptNewSolution(newSolution);
         checkPopulationConvergence();
      endWhile
      migrateIndividualsPoolMA();
      restartPopulation();
   endWhile

   /* IG threads concurrent to MA thread */
   initSol = randomSolution()
   while(cpuTime < limitCPU)
      forEachIGThread thread do
         startIG(thread, initSol);
         while (numIterations(thread) < limitIter)
            if (incumbentFound)
               migrateIndividualToPoolMIG(thread);
            endIf
         endWhile
         initSol = migrateIndividualFromPool();
      endFor
   endWhile
end
```

FIGURE 3. Pseudo-code for the MA+MIG approach.

2.2.1. *MA+MIG pseudocode.* The pseudocode of the MA+MIG algorithm is shown in Figure 3. Initially, a thread is created for the MA, concurrently with multiple IG threads to be used by the Multi-threaded IG (MIG). The MA thread will run a memetic algorithm and the IG-related threads will run dedicated IG methods. In the pseudocode, those steps are represented by the functions **createThreadMA** and **createThreadsIG**. At this point, the pseudocode is divided into two concurrent parts: one for the MA and another for the MIG; both of which use CPU time as stop criterion. In the MA thread, at first, the population is initialized (**initializePopulation**) and its structure is updated (**updatePopStructure**). This structure is important to accelerate the evolution process towards better quality solutions.

After that, the MA enters the main loop, in which the population will evolve. While the population has not converged (according to some criteria), new solutions will be generated (**generateOffspring**), mutated (**mutate**), and go through a local search procedure (**localSearchIG**). After that, the resulting solution might be accepted or not, depending on an acceptance criterion (**acceptNewSolution**). Finally, population convergence is checked (**checkPopulationConvergence**). Population convergence triggers the migration of individuals to the migration pool (**migrateIndividualsPoolMA**), followed by a restart procedure (**restartPopulation**) that randomizes part of it, and the process starts again.

Concurrently to the MA thread, we have the MIG running. For the IG-related threads, the IG algorithm is started with a random solution and runs for a given number of iterations. Every time a new incumbent solution is found, it is sent to the migration pool (**migrateIndividualToPoolMIG**). Once the limit of IG iterations was reached, the method restarts using one of the solutions currently present in the migration pool (**migrateIndividualFromPool**). The idea behind the migration pool is to provide the IG with high-quality but diverse solutions, and to communicate incumbent solutions between different IG threads. Next, we will describe each of the steps of the MA+MIG algorithm, as well as some of its most important features.

2.2.2. *Procedures **createThreadMA** and **createThreadsIG**.* These procedures create and initialize the MA thread that will run the memetic algorithm; and the IG threads, respectively. In our experiments, given k threads, thread 0 receives the MA, and the other $k-1$ threads are used by the MIG.

2.2.3. *Procedures **initializePopulation** and **updatePopStructure**.* These procedures initialize the MA populations and maintain the population structure, respectively. As mentioned before, the populations have a special structure that follows a ternary tree. The main characteristic of this structure is that the objective function of a solution in a given node will be better than the solutions in all other nodes below it. That makes solutions with better objective function values be placed at the upper nodes, whereas worse solutions are placed at the bottom nodes. Moreover, the best solution in the population will always be located at the root node. An ordering procedure (**updatePopStructure**) is necessary to maintain this structure when a new population is created, and every time an existing solution is replaced by a new one. This structure will influence how parents are selected for crossover and offspring is created. This, coupled with a special offspring acceptance criterion, creates an evolutionary pressure towards better solutions.

This structure has been compared to non-structured populations and to other types of structured populations in previous works, related to several combinatorial optimization problems [22][23]. It has consistently outperformed those other types of populations and provided a good trade-off between population size and convergence rate.

The initialization (**initializePopulation**) uses the NEH algorithm (from Nawaz, Enscore, and Ham [5]) to create solutions which are then improved by the IG method running for 10 iterations. This procedure is used to populate the top four nodes of the ternary tree, i.e. root node plus the three nodes in the intermediate layer. The nine nodes in the bottom layer are initialized with random solutions (see Figure 4). This procedure aims at creating a balanced initial population with some high quality solutions (top layers) and some low quality, random ones (bottom layer).

2.2.4. *Procedure **generateOffspring** and the crossover operator.* The **generate-Offspring** procedure creates new individuals by iteratively selecting parents and applying crossovers. The *Optimal 1-Point Crossover* is an extension of the 1-point crossover which returns the optimal child that can be created for a given pair of parents. In the traditional 1-point crossover, a parent is chosen, and a random position of the its sequence is selected. Then all jobs between the first position and the position selected are copied to the child solution. The rest of its chromosome is completed by sequentially copying the jobs from the second parent, in the same order as they appear, into the child – jobs that are already present are ignored, as duplicates are not allowed. In the optimal version of this crossover used in this work, both parents are tested as the parent to have its initial jobs copied into the child. In addition, instead of selecting only one position at random, all positions are tested and the solution returned is that with the best makespan. The pseudocode for the optimal 1-point crossover is shown next in Figure 5:

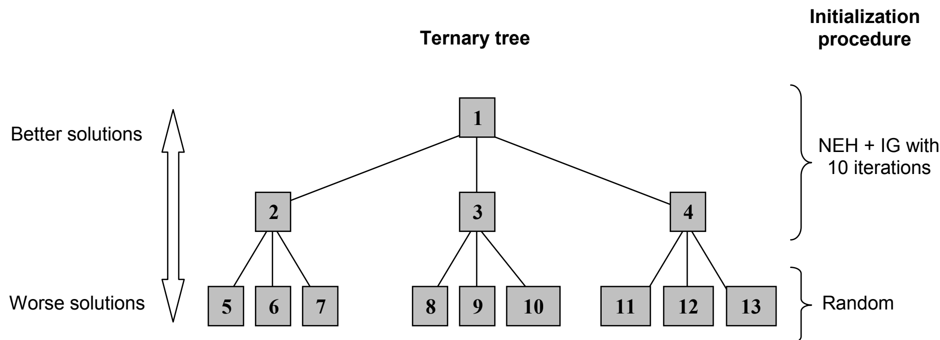

FIGURE 4. Population structure used in the memetic algorithm. The structure follows a ternary tree with 13 nodes. Higher nodes will contain higher quality individuals; bottom nodes will contain lower quality individuals (this structure is enforced by the procedure **updatePopStructure**). Initialization used the NEH algori followed by 10 iterations of the IG, for the top four individuals in the tree. The bottom layer individuals are all random.

```
Method Optimal1PointCrossover(parentA, parentB)
begin
   minimumMakespan = M; % M is an upper bound for the makespan
   for cutPosition = 1 to numJobs % checks all positions of ParentA's chromosome
      child = copy(1, cutPosition, parentA);
      child = completeChromosome(cutPosition+1, numJobs, parentB);
      if (makespan(child) < minimumMakespan) then
            {minimumMakespan = makespan(child); bestChild = child;}
   endFor
   for cutPosition = 1 to numJobs % checks all positions of ParentB's chromosome
      child = copy(1, cutPosition, parentB);
      child = completeChromosome(cutPosition+1, numJobs, parentA);
      if (makespan(child) < minimumMakespan) then
            {minimumMakespan = makespan(child); bestChild = child;}
   endFor
   return bestChild;
end
```

FIGURE 5. Pseudo-code for the optimal 1-point crossover.

This procedure has an impact in the way that parents are selected for recombination. Since it returns the 1-point crossover optimal child for a given pair of parents, it becomes worthless to select the same pair of parents more than once, as the solution returned will always be the same. That opens the opportunity to create a more intelligent parent selection procedure. The selection of parents is as follows: *parentA* is selected from the first two levels of the tree and *parentB* is chosen from one of the children nodes of *parentA*. Every time a pair of parents is chosen, that pair is prohibited from being selected again until at least one of them has changed. Otherwise, they would return exactly the same child. This prohibition has an impact on the pairs of solutions available for recombination at each iteration, and can be used as an indirect measure of population diversity. This topic will be addressed again when we discuss the procedure **checkPopulationConvergence**.

2.2.5. *Procedure **mutate**.* The child returned by the crossover procedure goes through a mutation process with a certain probability. Different mutation rates were tested, but the best results were obtained with 10%, i.e. 10% of the solutions selected will have a mutation operator applied onto it. The mutation operator is based on the inversion of sequences of jobs. Two positions in the chromosome are chosen at random, and sequence of jobs inside the interval is inverted.

2.2.6. *Procedure **localSearchIG**.* The local search receives the solution after the mutation phase and then applies the IG algorithm for a number of iterations. After several tests, we opted for a very quick search, with only 10 iterations. Crossover and mutation operators are the main responsible for searching the solutions space. The IG is used simply to perform a very localized search which might help in the convergence of the algorithm.

2.2.7. *Procedure **acceptNewSolution**.* Once the new solution has gone through local search, the MA verifies it it satisfies the acceptance criteria to be inserted in the population. Several acceptance criteria were tested, with different impacts on

the MA performance. Next, we list some of the strategies tested and comment on the results:

(1) Accept *newSolution* if its makespan is better than at least one of its parents. This strategy forces a strong evolutionary pressure and leads to high quality solutions very quickly. However, it constrains the diversity of the population, inducing premature convergence.
(2) Accept *newSolution* if its makespan is better or equal to at least one of its parents. This strategy also forces a strong evolutionary pressure, but if the new solution has the same makespan of one of its parents, two possibilities arise. First, the new solution might be a copy of its parents, which does not help the search process at all. Second, the new solution is different from the parent, even though they have the same makespan. That is a very common situation in the PFSP – the occurrence of several different solutions, with the same makespan. Accepting such individual adds new information to the population and still keeps evolutionary pressure.
(3) Accept *newSolution* if (i)/(ii) is satisfied and the new solution differs from both parents by at least $x\%$ of the jobs. This strategy forces a diversity checking whenever a new solution is to be added to the population no matter its quality.

Each strategy has its pros and cons, and that was very clear in our tests. The acceptance criteria chosen to be used in the memetic algorithm was (ii). Tests have shown that there are several solutions with different sequences of jobs and the same makespan. Especially when the search is close to the upper bound for a given instance, the number of distinct local optimum solutions with the same makespan becomes an issue, and ignoring this fact clearly reduces the exploratory ability of the method. A final observation is that when a new solution is accepted into the population, it always replaces the worst parent, i.e. the parent with the largest makespan.

2.2.8. *Procedure* ***checkPopulationConvergence****.* Population convergence needs to be checked constantly, otherwise the MA might waste CPU time performing crossovers, mutations and local searches over solutions that are virtually the same. In our implementation, the acceptance of new solutions into the population was used as a proxy for convergence measure.

For the Optimal 1-point crossover, the same pair of parents will always yield the same child every time the function is called; thus the crossover just needs to be called once. In the beginning, all pairs of selectable positions in the population tree are marked as *selectable*. Then, every time two solutions are selected to be parents in a crossover, the corresponding pair of positions is marked as *not-selectable*. Pairs of positions will become *selectable* again only when one of the solutions has changed, either by a new child being accepted and replacing the parent, or by the **updatePopStructure** procedure to maintain the consistency of the ternary tree. The migration of solutions and restart are triggered when all pairs of positions in the ternary tree are marked as *not-selectable*. That indicates that all possible pairs of parents have already been examined and produced their optimal children. To continue to execute crossovers would simply produce the same children repeatedly, thus wasting CPU time.

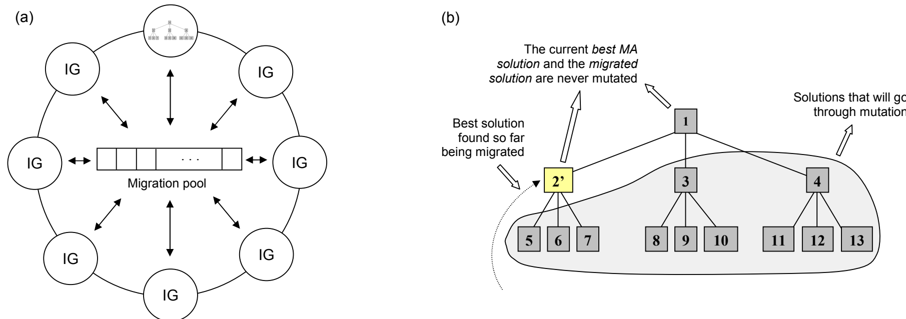

FIGURE 6. (a) MA+MIG algorithm. In the figure, there are eight threads; one running the MA and seven running IGs. In the centre we have the migration pool, which is used to communicate high quality individuals between the different threads. (b) Population restart applied after migration. The best individual and the individual received via migration are not changed. All other individuals in the population go through a special mutation procedure to add diversity to the population before the search process continues.

2.2.9. *Procedures* ***migrateIndividualsPoolMA*** *and* ***restartPopulation***. Migration of individuals from the MA population into the migration pool is represented by the procedure **migrateIndividualsPoolMA**. As mentioned before, when the MA population converges, it fills the migration pool with its best solutions. Insertions occur from best to worst individual and replaces all migration pool individuals. At the same time, the best solution found so far (either present in the migration pool, or in one of the IG threads) is copied into the MA population (i.e. if the MIG has found a new incumbent, it will be communicated to the MA in this step). Also important to mention, the migration pool has a fixed size (either 9 or 18 solutions), which we established during the parameter calibration experiments.

After migration, the population goes through a restart procedure, which aims at introducing diversity into the individuals. In Figure 6b we show how the restart occurs. The best individual of the population and the best solution found so far, which is now also part of the population, are not changed. All other individuals go through a mutation procedure. Mutation executes $numJobs/2$ random swaps in the sequence of jobs. The resulting sequences in those individuals will resemble random sequences, even though small sub-sequences might still be remain intact. Finally, the population goes through an ordering procedure to keep the structure.

2.2.10. *Procedure* ***migrateIndividualFromPool***. This procedure takes a random element from the migration pool and makes it the initial solution for the IG thread that has just reached the limit of iterations. Also, the procedure verifies if any solution in the pool has been copied more than X times, which triggers its removal from the pool.

Next, we present the computational results, which are divided into two parts. First, the tests performed to calibrate the algorithm, and then the tests on the

benchmark instances, which are compared to the current best results available in the literature.

3. Computational Results

Comprehensive tests with different configurations of the MA+MIG and MIG algorithms were executed to analyse their individual performances. The tests were conducted on all instances with 20 machines introduced in Taillard (1993), i.e. 50x20, 100x20, 200x20 and 500x20; 10 instances in each category. The parameters tested and the specific values were:

- *Size of migration pool*: 9, 18;
- *Size of MA population*: 13, 40, 121 individuals;
- *Number of IG steps*: 5, 50, 500, 5000; (Notice that these values refer to the IG-related threads only - the IG embedded in the MA uses IG_steps = 10, as described before.)

In Table 1, we show the results for the three overall best configurations in our tests, and compare them with the current upper bounds for each of the instances. In addition, we present the best makespan values for each of the instances, found by any of the three configurations. The column labelled 'Upper bound' shows the current upper bound for each instance, found in Taillard's webpage. The next three columns show the algorithms where (a) is the MA+MIG running with 5000 IG steps, population of 40 individuals and migration pool with 9 solutions; (b) is the MA+MIG running with 5000 IG steps, population of 121 individuals and migration pool with 18 solutions; and (c) is the MIG algorithm running with 5000 IG steps. Figures represent the average deviation from the upper bound and the corresponding standard deviation (between brackets) considering 5 runs with different seeds. The column 'Overall best' shows the best solution found by any of the three configurations. Between brackets we present the deviation of that value with respect to the lower bound, followed by which algorithms found that solution (indicated by (a), (b) or (c)).

All algorithms used in this study were implemented using the Intel C Compiler 10.1 with OpenMP and general optimization flags (-fast -O3). The machine had Dual Xeon E5405 (Quad core) 2.0GHz, 6Mb cache; 32Gb RAM. The operational system was Linux 64-bit. The code was written in standard C99. The random number generator is based on Mersenne Twister implementation.

The results in Table 1 indicate that for instances with 50 and 100 jobs, the two MA+MIG configurations and the MIG have similar results, with the three algorithms obtaining the best averages for nearly 1/3 of the instances. However, when the number of jobs grow beyond 200, there is a clear decline of the MIG algorithm, with the MA+MIG consistently obtaining the best averages, with a slight advantage for the configuration (a), which has less individuals and a smaller migration pool.

In terms of the best solutions found, the MIG is very consistent in reaching the best for instances with 50 and 100 jobs. Then, for 200 and 500 jobs, MA+MIG configuration (a) dominates the results, even though MIG continues to find a good proportion of best results. Our conclusion is that when the search space is smaller (50 and 100 jobs), the MIG can generate enough diversity to explore it thoroughly,

TABLE 1. Results for the three best configurations of the algorithms described in this work (configurations (a), (b) and (c)); plus the best results found for each of the instances. The columns referent to the three configurations present the average deviation from the current upper bounds, followed by the standard deviation (between brackets), calculated over 5 runs with different seeds. The column labelled "Overall best" presents the makespan of the best solutions found; and between brackets, it shows the respective deviation from the upper bound, followed by which algorithms reached that solution. Best averages for each instance are indicated in boldface. Configurations: (a) uses *IG steps = 5000*, *pop = 40* and *pool= 9*; (b) uses *IG steps = 5000*, *pop = 121* and *pool= 18* and (c) uses *IG steps = 5000*.

Jobs × Machines	Instance index	Upper bound	MA + MIG algorithm (a)	(b)	MIG algorithm (c)	Overall best
50 × 20	1	3847*	0.59 (0.12)	0.65 (0.13)	**0.58** (0.08)	3862$_{(0.39\|a)}$
	2	3704	0.24 (0.10)	0.27 (0.08)	**0.23** (0.06)	3708$_{(0.10\|a,b,c)}$
	3	3640	0.70 (0.22)	0.77 (0.15)	**0.64** (0.17)	3654$_{(0.38\|c)}$
	4	3719	**0.66** (0.20)	0.67 (0.20)	0.69 (0.20)	3732$_{(0.35\|c)}$
	5	3610	0.49 (0.14)	0.51 (0.10)	**0.48** (0.06)	3624$_{(0.39\|c)}$
	6	3679	**0.49** (0.20)	0.58 (0.17)	0.59 (0.15)	3690$_{(0.30\|c)}$
	7	3704	**0.54** (0.14)	0.60 (0.08)	0.64 (0.07)	3708$_{(0.46\|a)}$
	8	3691	0.81 (0.25)	**0.72** (0.28)	0.79 (0.32)	3703$_{(0.32\|c)}$
	9	3741	0.53 (0.11)	**0.52** (0.19)	0.56 (0.15)	3753$_{(0.32\|c)}$
	10	3756	**0.32** (0.03)	**0.32** (0.04)	0.36 (0.11)	3767$_{(0.29\|a,b,c)}$
100 × 20	1	6202	**0.82** (0.19)	**0.82** (0.18)	0.84 (0.11)	6246$_{(0.71\|c)}$
	2	6183	0.91 (0.14)	0.85 (0.24)	**0.80** (0.22)	6207$_{(0.39\|c)}$
	3	6271	**0.48** (0.13)	0.51 (0.20)	0.49 (0.11)	6294$_{(0.37\|a,b,c)}$
	4	6269	**0.48** (0.14)	0.49 (0.15)	0.54 (0.00)	6303$_{(0.54\|a,b,c)}$
	5	6314	0.51 (0.14)	0.52 (0.21)	**0.45** (0.15)	6330$_{(0.25\|c)}$
	6	6364	0.71 (0.26)	0.70 (0.31)	**0.48** (0.11)	6385$_{(0.33\|c)}$
	7	6268	**0.68** (0.16)	0.70 (0.16)	0.71 (0.12)	6304$_{(0.57\|a,b,c)}$
	8	6401	**0.81** (0.15)	**0.81** (0.28)	0.83 (0.12)	6444$_{(0.67\|a,c)}$
	9	6275	0.76 (0.17)	0.85 (0.22)	**0.66** (0.17)	6304$_{(0.46\|c)}$
	10	6434	0.62 (0.18)	**0.55** (0.24)	0.65 (0.06)	6465$_{(0.48\|a,b)}$
200 × 20	1	11181	**0.87** (0.20)	0.93 (0.29)	1.00 (0.15)	11254$_{(0.65\|a)}$
	2	11203	**1.11** (0.26)	1.15 (0.24)	1.25 (0.12)	11318$_{(1.03\|a)}$
	3	11281	**1.20** (0.14)	1.23 (0.15)	1.28 (0.11)	11407$_{(1.12\|a,b,c)}$
	4	11275	**0.74** (0.20)	**0.74** (0.20)	0.83 (0.07)	11359$_{(0.75\|a,b,c)}$
	5	11259	**0.50** (0.15)	0.51 (0.15)	0.57 (0.07)	11310$_{(0.45\|a,b,c)}$
	6	11176	**0.87** (0.15)	0.95 (0.26)	0.92 (0.11)	11262$_{(0.77\|a,c)}$
	7	11360	0.78 (0.08)	**0.77** (0.09)	0.80 (0.01)	11451$_{(0.80\|a,b,c)}$
	8	11334	0.77 (0.09)	**0.70** (0.11)	0.76 (0.09)	11402$_{(0.60\|b,c)}$
	9	11192	**0.90** (0.24)	0.93 (0.21)	1.04 (0.18)	11276$_{(0.75\|c)}$
	10	11288	1.04 (0.21)	**0.99** (0.24)	1.12 (0.06)	11404$_{(1.03\|b,c)}$
500 × 20	1	26059	**0.45** (0.09)	0.48 (0.04)	0.49 (0.04)	26175$_{(0.44\|a,b,c)}$
	2	26520	**0.72** (0.14)	0.76 (0.07)	0.79 (0.04)	26714$_{(0.73\|a,b,c)}$
	3	26371	**0.43** (0.06)	0.45 (0.03)	0.46 (0.03)	26481$_{(0.42\|a)}$
	4	26456	**0.35** (0.04)	**0.35** (0.04)	0.36 (0.02)	26549$_{(0.35\|a,b,c)}$
	5	26334	**0.26** (0.05)	**0.26** (0.05)	0.27 (0.04)	26390$_{(0.21\|a,b,c)}$
	6	26477	**0.42** (0.07)	0.43 (0.07)	0.46 (0.03)	26584$_{(0.40\|c)}$
	7	26389	0.27 (0.07)	**0.26** (0.08)	0.30 (0.03)	26461$_{(0.27\|a,b,c)}$
	8	26560	**0.37** (0.08)	0.40 (0.08)	0.39 (0.07)	26640$_{(0.30\|a,c)}$
	9	26005	**0.64** (0.16)	0.69 (0.14)	0.83 (0.13)	26144$_{(0.53\|a)}$
	10	26457	**0.47** (0.07)	**0.47** (0.07)	0.50 (0.03)	26572$_{(0.43\|a,b,c)}$

* The MA+MIG algorithm found a solution with makespan equal to 3846.
This finding will be discussed next in Section 3.1.

matching the performance of the MA+MIG. However, when the search space increases too much (200 and 500 jobs), the exploratory characteristics of the MA start to show up, and the method consistently obtains better results than the MIG.

TABLE 2. Comparison between Vallada and Ruiz (2009) [19] method Cooperative Iterated Greedy (CIG), and the MA+MIG (a)(b) and MIG (c). Results are averaged for each size of instance.

Jobs *times* Machines	Cooperative IG (CIG) $p = 4$	$p = 6$	$p = 8$	MA + MIG algorithm (a)	(b)	MIG algorithm (c)
50 × 20	0.71	0.59	0.54	0.54	0.56	0.56
100 × 20	0.76	0.74	0.63	0.68	0.68	0.65
200 × 20	0.93	0.84	0.81	0.88	0.89	0.96
500 × 20	0.43	0.42	0.38	0.44	0.46	0.49

3.1. New upper bound for 50x20 instance #1.

In one of the runs of the MA+MIG algorithm, with the configuration IG steps = 5000; pop = 40; pool = 9; the method managed to obtain a solution with makespan equal to **3846**, which improves the current best makespan of 3847. The sequence of the jobs is as follows (jobs are numbered from 0 to 49): 19 30 38 26 42 14 43 10 7 44 34 36 5 16 33 27 6 13 41 32 39 23 4 28 9 1 17 46 47 20 45 0 15 48 11 22 21 35 31 37 18 8 25 24 12 40 29 3 49 2. This was a surprising result that further demonstrates the capabilities of the MA+MIG algorithm.

3.2. Comparison with Cooperative Iterated Greedy (CIG) – Vallada and Ruiz (2009) [19].

The intent of this study was to show alternate ways of exploring the full power of multi-core machines. However, we believe it is worth to compare the results with the work of Vallada and Ruiz (2009), which uses distributed processing. In Table 2, we compare our results from Table 1 with those from the Cooperative Iterated Greedy (CIG) method, introduced in Vallada and Ruiz (2009) [19]. The CIG is similar to the MIG used in this work, but it has a fundamental architectural difference. The CIG uses the concept of islands; each island corresponding to a computer with a dual-core processor. For instance, the CIG configuration of $p = 4$ contains four islands and 8 threads in total. Moreover, in each island/processor, one core is responsible for the running the IG algorithm, whereas the other is responsible for the communication.

In our implementation of the MIG, when 8 threads are used, we actually have 8 IG algorithms running concurrently, but each thread's is responsible for its communication. Because of that, MIG threads typically incur in significant amounts of waiting time, whereas communication in CIG is much faster. Comparing the two algorithms thready-by-thread might be misleading in this case. In Table 2, we chose to include the results for the CIG with $p = 4$ (8 threads); $p = 6$ (12 threads) and $p = 8$ (16 threads). Note that, on the other hand, our tests with MA+MIG and MIG use 8 threads only. We leave to the reader draw their own conclusions in terms of performance, reiterating that it is not a main aspect of this study.

4. CONCLUSION

This study presented new parallel hybrid search methods for the permutation flow shop problem. The methods are based on the use of Memetic Algorithms (MA) and the Iterated Greedy Algorithm (IG) in a multi-threaded environment (eight threads in total). The configurations tested use either one thread for the MA and seven for the IG (MA+MIG algorithm), or all eight threads dedicated to the IG (MIG algorithm). Different values for parameters controlling migration of

individuals between MA and IG; size of the MA population; and the number of IG_steps were also tested. Results indicate that two of the MA+MIG configurations had a superior performance compared to the MIG alone, especially for larger instances (200 and 500 jobs). For 50 and 100 jobs, the performance was similar between all methods. During our tests, we managed to find a new upper bound for instance 50x20 #1, reducing the makespan from 3847 to 3846. The sequence obtained is presented as well.

Further on, we compared the results from our methods with those from Vallada and Ruiz (2009) [19], which is the most recent study on multi-threading techniques applied to the PFSP problem found in the literature. That comparison was undermined by the architectural difference between the approaches, but it seems that our methods had a similar performance to the best one in Valladas' study (the Cooperative IG - or CIG). If you consider Vallada's CIG with eight threads ($p = 4$ islands), our approach (which also has eight threads) performed consistently better. If you consider the CIG with 16 threads ($p = 8$ islands), our methods' performance is still comparable for the 50x20 instances, but they lose to the CIG for the larger instances by around 10%.

References

[1] K. R. Baker.Handbooks in Operations Research and Management Science, Vol. 4, 571–627. Elsevier Science Publishers, New York (1993).

[2] E. Coffman. Computer and Job-Shop Scheduling Theory. Wiley, New York (1976).

[3] A.H.G. Rinnooy Kan. Machine Scheduling Problems: Classification, Complexity, and Computations. Martinus Nijhoff, The Hagues (1976).

[4] D. G. Dannenbring. Management Science, 23, 1174–1182 (1977).

[5] M. Nawaz, E.E. Enscore, and I. Ham. Omega, 11, 91–95 (1983).

[6] M.S Nagano, J.V. Moccellin. The Journal of the Operational Research Society, 53, 1374-1379 (2002).

[7] É. Taillard. European Journal of Operational Research, 47, 65–74 (1990).

[8] É. Taillard. European Journal of Operational Research, 64, 278–285 (1993).

[9] M. Ben-Daya and M. Al-Fawzan. European Journal of Operational Research, 109, 88–95 (1998).

[10] F.A. Ogbu and D.K. Smith. Computers & Operations Research, 17, 243–253 (1990).

[11] T. Stützle. Technical report, TU Darmstadt, AIDA-98-04, FG Intellektik, (1998).

[12] R. Ruiz and C. Maroto.European Journal of Operational Research, 165, 479–494 (2005).

[13] S. M. Johnson.Nav. Res. Log. Quarterly, 1, 61–68 (1954).

[14] C.R. Reeves. Computers & Operations Research, 22, 5–13 (1995).

[15] R. Ruiz, C. Maroto, and J. Alcaraz.MIC2003: The Fifth Metaheuristics International Conference, 63.1–63.8 (2003).

[16] R. Ruiz, C. Maroto, and J. Alcaraz. OMEGA, The International Journal of Management Science, 34, 461–476 (2006).

[17] A. Agarwal, S. Colak, and E. Eryarsoy, European Journal of Operational Research, 169, 801–815 (2006).

[18] R. Ruiz and T. Stützle. European Journal of Operational Research, 77, 2033-2049 (2006).

[19] E. Vallada and R. Ruiz. European Journal of Operational Research, 192, 365–376 (2009).

[20] F. Glover and G. Kochenberger. Handbook of Metaheuristics, Springer, USA (2003).

[21] D. Goldberg and K. Sastry. Genetic Algorithms: The Design of Innovation, 2nd ed., Springer, USA (2010).

[22] L. Buriol, P. Franca and P. Moscato. Journal of Heuristics, 10, 483–506 (2004).

[23] P. Moscato, A. Mendes and R. Berretta. Biosystems, 88, 56–75 (2007).

(Martín Gómez Ravetti) Departamento de Engenharia de Produção, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, MG Brazil
E-mail address: martin.ravetti@dep.ufmg.br

(Carlos Riveros) School of Electrical Engineering and Computer Science, The University of Newcastle, Newcastle, NSW, Australia.
E-mail address: carlos.riveros@newcastle.edu.au

(Alexandre Mendes) School of Electrical Engineering and Computer Science, The University of Newcastle, Newcastle, NSW, Australia.
E-mail address: alexandre.mendes@newcastle.edu.au

(Mauricio G.C. Resende) Algorithms and Optimization Research Department, AT&T Labs Research, Florham Park, NJ USA.
E-mail address, M.G.C. Resende: mgcr@research.att.com

(Panos M. Pardalos) Department of Industrial and Systems Engineering, University of Florida, Gaineville, FL USA.
E-mail address: pardalos@ufl.edu