Decision Making in Manufacturing and Services
Vol. 8 • 2014 • No. 1–2 • pp. 13–23

Very Fast Non-Dominated Sorting

Czesław Smutnicki*, Jarosław Rudy*, Dominik Żelazny*

Abstract. A new and very efficient parallel algorithm for the Fast Non-dominated Sorting of Pareto fronts is proposed. By decreasing its computational complexity, the application of the proposed method allows us to increase the speedup of the best up to now Fast and Elitist Multi-Objective Genetic Algorithm (NSGA-II) more than two orders of magnitude. Formal proofs of time complexities of basic as well as improved versions of the procedure are presented. The provided experimental results fully confirm theoretical findings.

Keywords: parallel algorithms, Pareto sorting, computational complexity, GPU computing, multiple-criteria decision analysis, NSGA-II

Mathematics Subject Classification: 90

Revised: September 27, 2014

1. INTRODUCTION

The field of Multi-Objective Optimization (MOO) has received considerable attention in the last few years, since many classic single-criterion problems (like scheduling, container loading, vehicle routing, etc.) have been modified towards bi- and multi-criteria cases. Such alterations usually tune well with practical situations as real-life problems require the decision makers to view them from more than one perspective. While practical, such problems are also extremely hard to solve and existing general-purpose algorithms may prove unefficient or infeasible when applied to their multi-criteria variants.

Therefore, considerable effort was put into the development of optimization methods aimed solely at MOO. One of the commonly used notion in such methods is the concept of Pareto-efficiency, where the domination relation is used to obtain a set of non-dominated solutions from all considered solutions, called the Pareto frontier. In result a number of approaches to MOO is aimed at obtaining the Pareto frontier or its approximation, see Minella *et al.* (2008) for more detail, especially when the decision maker's preferences are unknown in advance. Because of NP-hardness of majority of problems, chiefly methods approximating Pareto frontier were proposed.

* Wrocław University of Technology, Institute of Computer Engineering, Control and Robotics, Poland, e-mail: czeslaw.smutnicki@pwr.edu.pl, corresponding author, jaroslaw.rudy@pwr.edu.pl, dominik.zelazny@pwr.edu.pl

DOI: http://dx.doi.org/10.7494/dmms.2014.8.1.13

Among known metaheuristic approaches used for the approximate solving of MOO problems, Fast and Elitist Multi-objective Genetic Algorithm (NSGA-II) of Deb *et al.* (2002) is commonly considered the best one, see also the conclusions from Minella *et al.* (2008), Yijie and Gongzhang (2008) and Rudy and Żelazny (2012). This algorithm relies on a procedure, called Fast Non-dominated Sorting (FNS), with the computational complexity $O(KN^2)$, which divides the given set of N solutions, evaluated through K criteria, into so called Pareto fronties (layers), defined as follows: solutions from the first layer are not dominated by any solution, the solutions from the second layer are only dominated by solutions from the first layer, and so on. Adjective "fast" follows from the reduction of complexity from the originally proposed $O(KN^3)$ to the improved $O(KN^2)$. Layers are used in the selection phase of the genetic algorithm, when a certain number of solutions (individuals) from the current offspring must be chosen to create the next population. Solutions from the first layer are chosen first and, if more solutions are needed, successive layers are considered. Solutions belonging to one layer are also ranked using the notion of solution distance, which keeps the selection process from choosing many similar solutions.

Simultaneously, with the development of solution methodology, parallel versions of methods have appeared in the literature, in order to improve the numerical properties of sequential algorithms for a wide range of problems, see e.g. Bożejko *et al.* (2013) and Bożejko *et al.* (2014) in case of scheduling problems. This refers also to MOO, in particular NSGA-II and its variants. Interestingly, for NSGA-II all published parallelizations deal with offspring creation (which is evident), but not with the FNS prcedure. A distributed computing approach to NSGA-II proposed in the paper Deb *et al.* (2003) introduced a modified domination criterion. Another approach to parallel NSGA-II was presented in paper Jozefowiez *et al.* (2006) and introduced the Elitist Diversification Mechanism for the purpose of parallel application to Vehicle Routing Problem with Route Balancing. A study of master-slave approaches to NSGA-II can be found in paper Durillo *et al.* (2008). Finally, a survey of different parallel approaches, including NSGA-II, to multi-objective optimization was presented in paper Talbi *et al.* (2008).

Table 1. *Possible speedups for NSGA-II without parallelization of the FNS procedure*

L	running time [s]				p	S_p	S_∞
	FNS	GA	NSGA-II	FNS/NSGA [%]			
50	2.34	2.69	5.03	47	1	1	1
100	10.64	5.52	16.16	66	1	1	1
200	37.08	10.73	47.81	78	1	1	1
50	2.34	0.05	2.39	98	50	2.10	2.15
100	10.64	0.05	10.7	99	100	1.51	1.52
200	37.08	0.05	37.13	100	200	1.29	1.29

While developing the parallel version of the NSGA-II we observed the significant influence of the FNS procedure on the final computational complexity of NSGA-II, thus its running time and possible speedup. Let as assume the population size $L = 100$. Then, the number of solutions to sort is equal to $N = 2L = 200$ (since the FNS procedure works on both the parent and the children populations, both of size L). Indeed, for $N = 200$, the procedure FNS consumes nearly 80% of the total computation time, see Table 1 for sigle-processor run. The FNS procedure takes up so much time because of its time complexity which was described by Deb *et al.* (2002) to be $O(KN^2)$, while the complexity of the remaining parts (*i.e.* general GA framework) is $O(KL)$, where K is the number of criteria, L is the population size and N is the number of solutions to sort ($N = 2L$). This means that for sufficiently large N FNS will always take the majority of the computation time. Assuming that FNS cannot be realized as a parallel procedure, we can obtain via Amdahl's law from Amdahl (1967) the following surprising result. Let $B < 1$ be the fraction representing the running time of the FNS procedure compared to the entire running time of the NSGA-II algorithm in a single-processor environment. Then, the speedup of NSGA-II using p processors (in comparison to single processor run) is given by:

$$S_p = \frac{1}{B + \frac{1}{p}(1 - B)} \tag{1}$$

which limits the theoretical possible speedup to

$$S_\infty = \lim_{p \to \infty} S_p = \frac{1}{B} \tag{2}$$

For $N = 200$ and $p = 1$ we have $B = FNS/NSGA \approx 78\%$ which yields $S_\infty = 1.29$, far away from our expectations. This means that even if we can easily make the other parts of the NSGA-II (crossover and mutation process, goal function evaluation) to work in parallel, the overall speed up capability of the algorithm remains severely limited as shown in the last column of Table 1. Therefore, the key success to improve the parallel NSGA-II lies in the efficient parallelization of the FNS procedure, which is the primary purpose of this work. The remainder of this paper is organized as follows: in Section 2 sequential algorithm for FNS procedure is described, along with proof of its computational complexity. In Section 3 two parallel approaches to FNS are presented, including time complexity proofs. The results of the computer experiment using GPU and CUDA technologies is presented in Section 4. Finally, Section 5 provides the conclusions.

2. SEQUENTIAL ALGORITHM

We start the analysis with the sequential version of the FNS method. Our implementation differs slightly from the original work by Deb *et al.*, although it has the same time complexity of $O(KN^2)$. In particular, for N solutions we create a matrix of size $N \times N$ called the *domination matrix* D. The element $d_{i,j}$ of this matrix indicates whether solution i dominates solution j. Thus, the sequential algorithm is constructed as a part of the proof of Theorem 1.

Theorem 1. *Let $S = \{S_1, S_2, \ldots, S_N\}$ be a set of N solutions and let $C = \{c_1, c_2, \ldots, c_K\}$ be a set of K criteria. Let s_i^c be the objective function value for solution i on criterion c. Then, the FNS of Pareto fronts can be done in time $O(KN^2)$ on a sequential processor.*

Proof. First, we compute a domination matrix: the time needed for this is $KN(N-1)$ since for every pair of solutions (j_1, j_2) we need to check the K criteria and the number of such pairs is $N(N-1)$. For each solution, we also compute the number (sum) of solutions that it dominates. This can be done while computing the matrix itself (each domination simply adds 1 to one of the sums) and thus can be done in $KN(N-1)$ steps as well. A solution never dominates itself, so the remaining N elements of the matrix are filled with 0 s (no domination relation) in N steps.

Next, we have a loop that assigns solutions to fronts. Let us assume the number of fronts is $F \leqslant N$. Let us consider front i and let f_i be the number of solutions that will be assigned to this front. The assignment works as follows: for each solution we check whether its previously computed sum equals 0 (this means the solution is not dominated) in constant time. We do this for every non-assigned solution, so for at most N solutions. Next, we need to remove the solutions added from the future calculations. We do this by substracting 1 from every sum representing a solution is dominated by any solution from the current front (one solution can be dominated many times, requiring multiple subtractions). This is done in Nf_i steps as every solution requires at most N substractions and the number of solutions is f_i.

Therefore, the single iteration of the loop requires $N + Nf_i$ steps. Thus, the number of steps needed to assign all the fronts (entire loop) is equal to:

$$\sum_{i=1}^{F}(N + f_i N) = FN + N\sum_{i=1}^{F} f_i \tag{3}$$

However, we notice that all solutions need to be assigned to fronts, so $\sum_{i=1}^{F} f_i = N$. Thus, the above equation takes the form of $FN + N^2$. Moreover, $F \leqslant N$, so we simply get $N^2 + N^2$ for all the iterations. Thus, the total complexity for the non-dominated sorting procedure is:

$$2KN(N-1) + 2N^2 = O(KN^2 + N^2) = O(KN^2) \tag{4}$$

■

3. PARALLEL ALGORITHM

Now, let us consider a parallel version of this algorithm using p procesors. We have N solutions and K values for every solution, so the size of our input is $O(NK)$. From this, it would seem that NK parallel processors should be enough to reduce the computation time significantly. However, we notice that most of the processing is not done directly on the input, but on the domination matrix, which has size $N \times N$. From this, we conclude that the logical number of processors for the parallel algorithm

should be $O(N^2)$. We assumed the PRAM CREW (Concurrent Read, Exclusive Write) model. Thus, p processors can write to one memory location only sequentially in time $O(p)$. Below in Theorem 2 the computational complexity of $O(K+N\log N)$ is claimed and the algorithm is constructed in the proof.

Theorem 2. *Let $S = \{S_1, S_2, \ldots, S_N\}$ be a set of N solutions and let $C = \{c_1, c_2, \ldots, c_K\}$ be a set of K criteria. Let s_i^c be objective function value for solution i on criterion c. Then, the FNS of Pareto frontiers can be done in time $O(K+N\log N)$ on $P = N^2$ parallel processors, assuming the PRAM CREW model.*

Proof. First, we compute a domination matrix: the time to compute element i, j (whether solution i dominates j) requires K steps (comparison of values of at most all K criteria for i and j). Since the number of processors is equal to the size of the matrix ($N \times N$) and calculation of each element is independent from others (concurrent read is allowed, data is written to different matrix elements), then the matrix can be computed in $O(K)$.

Next, we need to assign the solutions to the first front. We start by calculating the sum of values in each row of the matrix – if this sum for row i equals 0, then the solution j_i is not dominated by any other solution and has to be assigned to the current front. We have N rows and N^2 parallel processors, so each row can be sumed independently by the N processors.

The sum of a vector of X values can be done in $O(\log X)$ using $\frac{X}{2}$ parallel processors: in the first step we reduce X values into $\frac{X}{2}$ values (each value is a sum of two elements). In next step we reduce $\frac{X}{2}$ values into $\frac{X}{4}$ and so on, until we get a single value *i.e.* $\frac{X}{X} = 1$. The needed number of such steps is $\log X$. Let us notice that this method assumes that the vector size is a power of 2 *i.e.* $X = 2^a$, $a \in \mathbb{N}$. In our case the vector size N is not necessarily a power of 2. Fortunately, we can extend our matrix (the extended part is filled with 0s) to be of size $N \times R$, where R is the smallest power of 2 such that $N \leqslant R$. It is trivial to show that $R \leqslant 2N$. Thus, we need a sum of matrix rows of size at most $2N$. As shown above this can be done in time $O(\log 2N) = O(\log N)$ on $\frac{2N}{2} = N$ parallel processors – just the number of processors we have for each row. After this we have N sums computed in $O(\log N)$ and we can assign solutions to the current front.

Let us consider front i and, as before, let f_i be the number of solutions assigned to that front. We need to check N solutions and assign to the front all that have the sum equal to 0. With N parallel processors (we have much more than that) this can be done in $O(1)$. Next, however, each solution assigned to front i forces us to increase the global counter of assigned solutions (our halting condition). This means concurrent write and with f_i writes this can take time $O(f_i)$. Single front i can therefore be assigned in $O(\log N + 1 + f_i) = O(\log N + f_i)$.

This process has to be repeated for every front. If we assume F fronts, then the total computational complexity for this procedure is:

$$O(K + \sum_{i=1}^{F}(\log N + f_i)) \tag{5}$$

However, from Theorem 1 we know that:

$$\sum_{i=1}^{F} f_i = N \tag{6}$$

also:

$$\sum_{i=1}^{F} \log N = F \log N \tag{7}$$

Lastly, $F \leqslant N$. Thus, the final complexity is:

$$O(K + N \log N + N) = O(K + N \log N) \tag{8}$$

■

The obtained time complexity is better that the sequential FNS, but we would like to show that the same number of parallel processors can be used to achieve even better result. This procedure will be called Very Fast Non-dominated Sorting (VFNS) and has the time complexity as stated in Theorem 3.

Theorem 3. *Let $J = \{J_1, J_2, \ldots, J_N\}$ be a set of N solutions and let $C = \{c_1, c_2, \ldots, c_K\}$ be a set of K criteria. Let j_i^c be objective function value for solution i and criterion c. Then, the VFNS of the Pareto frontiers can be done in time $O(K + N)$ on $P = N^2$ parallel processors, assuming the PRAM CREW model.*

Proof. First, we compute a domination matrix and, as in Theorem 3 this takes $O(K)$.

Next, we need to calculate the sum of values in each row of the matrix, but we will do it only once – later we will just substract the values from those sums. Thus, we calculate N vectors of sizes up to $2N$. This can be done in time $O(\log N)$ as in Theorem 2.

Now, we can assign solutions to the fronts. Let us consider front i which will be assigned f_i solutions as before. With over N processors we can check each solution independently. Unfortunately, each front assignment will force as to increase the global counter of solutions assigned so far and substract from (update) one of the previous sums. Since this requires a concurrent write, it can be done in $O(f_i)$ steps. Thus, for a single front we have $O(f_i)$. For all F fronts, however, we get:

$$\sum_{i=1}^{F} f_i = N = O(N) \tag{9}$$

since the sum of solutions in all fronts must equal N. Thus, the final computational complexity on N^2 parallel processors with the CREW model restrictions is:

$$O(K + logN + N) = O(K + N) \tag{10}$$

■

The last theorem allows us to perform the VFNS in linear time with regard to both the number of solutions N and the number of criteria K. If $K = N$, then we managed to reduce the complexity from $O(N^3)$ to $O(N)$, thus the number of parallel processors equal to N^2 is justified. We assume that the number of criteria will never be larger than the number of solutions.

4. COMPUTER EXPERIMENT

In order to test the theoretical results obtained above, we decided to perform a series of computer simulations using the Compute Unified Device Architecture (CUDA) platform for parallel computing on GPU devices. The chosen GPU was nVidia^{TM} Tesla K20, reported with 2946 cores, managed with its own strategy regarding the balancing of processor usage and sharing memory. In order to compare the sequential and parallel algorithm, we implemented them both to run in CUDA, i.e. the sequential algorithm runs in only one thread, while the parallel algorithm launches N threads in N blocks, for N^2 threads in total. We focused on the parameter N, since it is rare to consider a number of criteria higher than 3 or 4, thus K can be treated as a constant. The time results for both algorithms are shown in Figure 1.

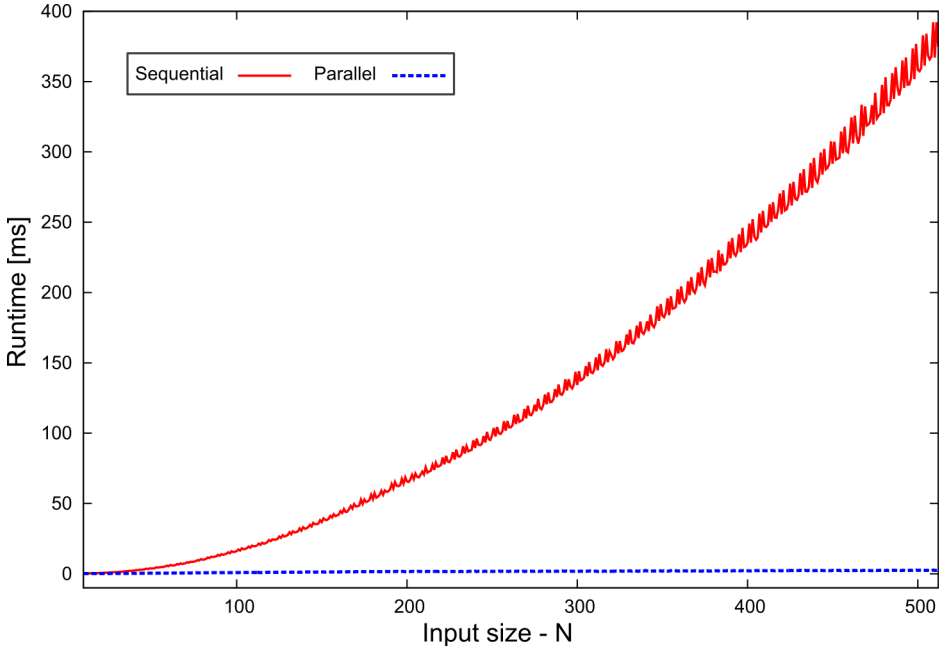

Fig. 1. *Runs of FNS (sequential) and VFNS (parallel) methods*

We observe that while the computation time of the sequential algorithm grows quickly, the parallel algorithm remains almost constant in comparison, never exceeding over 3 ms. The plot of the sequential algorithm is ragged. Limitations of the GPU

device used are partially responsible for this, but the other cause may be that the running time of the algorithm is dependent on the numbers of fronts F present in the data set. The same feature is true for the parallel algorithm when magnified.

Since our purpose was to state and reduce the time complexity of the FNS method, we would like to compare the obtained empirical results with the theoretical computations model of $O(KN^2)$ and $O(K+N)$ for the sequential and parallel algorithm respectively. With K treated as a constant these complexities become $O(N^2)$ and $O(N)$ respectively. We used the Curve Fitting Tool to try and match our data to these models. For the sequential algorithm the obtained curve was Ax^B with coefficient A in interval $[0.002993, 0.003035]$ and B in $[1.879, 1.881]$. The coefficient of determination (R^2) was 1 and Root-Mean-Square Error (RMSE) was 0.49. For the parallel algorithm model $Ax + B$, we obtained coefficient A in interval $[0.004017, 0.004256]$ and B in $[0.5214, 0.5927]$. $R^2 = 0.9123$ with RMSE $= 0.1979$. All coefficients were obtained with 95% confidence bounds. From this we conclude, that the empirical results match well with the obtained theoretical models.

Next, we considered the speedup achievable due to the use of the parallel algorithm. Both algorithms were executed on the same CUDA device, making speedup analysis easier. Since the plots of the algorithms are not smooth, it is possible for the speedup to vary for similar values of N. Thus, we decided to present the average, minimum and maximum values of the speedup on different interval of N. The results of our computer experiment are shown in Table 2.

Table 2. *Speedups for different input sizes for parallel FNS method,* $p = 2496$

Instance size	Minimum	Maximum	Average
1–32	1.37	8.46	4.43
33–64	7.35	15.32	10.34
65–96	12.65	20.16	16.11
97–128	19.58	28.13	23.23
129–160	24.26	34.15	29.47
161–192	31.67	44.29	36.41
193–224	40.10	55.68	46.26
225–256	46.76	64.21	55.69
257–288	55.47	76.17	66.23
289–320	70.85	88.78	80.14
321–352	73.09	102.18	88.97
353–384	92.42	114.14	102.16
385–416	94.36	121.33	111.71
417–448	112.00	143.15	123.48
449–480	113.63	148.21	135.52
481–512	138.28	162.85	146.03

In all cases the minimum speedup of one interval is higher than in the previous one and very often even higher than the average speedup of the previous interval. Moreover, the relative range *i.e.* $(\max - \min)/\text{avg}$ gets lower with the increase of N. The obtained average speedup for the interval around $N = 500$ is close to 150 and the maximum value is 162. We have also performed some tests for higher instance sizes that 512. The results strongly suggest that the maximal value of speedup for $N = 650$ can exceed even 200.

We can use the obtained speedup values to show the new paralellization limits for the NSGA-II algorithm with the parallel VFNS procedure. First, we remember that the number of solutions to sort is twice the number of population size (because sorting is performed on L children and L parents). Thus, we look up the speedup values for $N = 100$, $N = 200$ and $N = 400$ to correspond to population sizes $L = 50$, $L = 100$ and $L = 200$ and compute new speedup values for the NSGA-II algorithm, which are shown in Table 3.

Table 3. *Speedups for NSGA-II with parallelization of the FNS,* $p = 2496$

L	running time [s]				S_p
	FNS	GA	NSGA-II	FNS/NSGA [%]	
50	0.467	0.054	0.521	90	9.664
100	0.790	0.055	0.845	93	19.119
200	1.079	0.054	1.133	95	42.202

We observe that the reduction of the running time thanks to the VFNS procedure allowed to speedup the entire NSGA-II algorithm 5 to 30 times higher that the theoretical limit without the standard FNS procedure (Tab. 1). Let us also note that those are empirical results obtained with the available number of processors severely limited (GPU allowed for up to 2496 CUDA cores). We conclude that the paralellel VFNS procedure allowed to speedup the NSGA-II algorithm by at least an order of magnitude for the population size $L \geqslant 100$.

Finally, in order to approximate the potential speedup, we can make an interpolation as follows. It is well known that an algorithm of complexity $O(c)$ performing on p processors can be processed on $p' < p$ processors in the time $O(cp/p')$. Since results in Table 3 are obtained for $p' = 2496$ processors, thus for $L = 200$, we can reach the theoretical speedup of over 650 by using $p = N^2 = 400^2$ processors.

5. CONCLUSIONS

In this paper we proposed a method of speeding up the Fast Non-dominated Sorting procedure called the Very Fast Non-dominated Sorting that can be used for various purposes in Multi-Criteria Optimization and similar fields, particularly for the NSGA-II algorithm. We managed to reduce the complexity of the procedure from polynomial to linear. We also presented formal proofs and computer simulations supporting

the complexities of both the sequential and parallel version of the algorithm. The obtained speedup values are significant and exceed 100 with N over 400. The resulting empirical speedup of the NSGA-II is 30 times its original limit number for $N = 400$ and the possible speedup with sufficient number of processors exceeds 650 (compared to the single-processor NSGA-II run).

ACKNOWLEDGEMENTS

This work is co-financed by the European Union as part of the European Social Fund.

REFERENCES

Amdahl, G.M., 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485. ACM.

Bożejko, W., Pempera, J., and Smutnicki, C., 2013. Parallel tabu search algorithm for the hybrid flow shop problem. In *Computers and Industrial Engineering*, **65**(3), pp. 466–474.

Bożejko, W., Uchroński, M., and Wodecki, M., 2014. Multi-gpu tabu search metaheuristic for the flexible job shop scheduling problem. In *Advanced Methods and Applications in Computational Intelligence*, volume 6 of *Topics in Intelligent Engineering and Informatics*, pp. 43–60.

Deb, K., Pratap, A., Agarwal, S. and Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. In *IEEE Transactions on Evolutionary Computation*, **6**(2), pp. 182–197.

Deb, K., Zope, P., and Jain, A., 2003. Distributed computing of pareto-optimal solutions with evolutionary algorithms. In *Evolutionary Multi-Criterion Optimization*, Vol. 2632 of *Lecture Notes in Computer Science*, pp. 534–549.

Durillo, J.J., Nebro, A.J., Luna, F. and Alba, E., 2008. A study of master-slave approaches to parallelize NSGA-II. In *Proceedings of International Symposium on Parallel and Distributed Processing*, pp. 1–8.

Jozefowiez, N., Semet, F. and Talbi, E., 2006. Enhancements of NSGA-II and its application to the vehicle routing problem with route balancing. In *Artificial Evolution*, Vol. 3871 of *Lecture Notes in Computer Science*, pp. 131–142.

Minella, G., Ruiz, R. and Ciavotta, M., 2008. A review and evaluation of multiobjective algorithms for the flowshop scheduling problem. *INFORMS Journal on Computing*, **20**(3), pp. 451–471.

Rudy, J. and Żelazny, D., 2012. Memetic algorithm approach for multi-criteria network scheduling. In *Proceedings of the International Conference On ICT Management for Global Competitiveness And Economic Growth In Emerging Economies*, pp. 247–261.

Talbi, E., Mostaghim, S., Okabe, T., Ishibuchi, H., Rudolph, G., and Coello, C.A., 2008. Parallel approaches for multiobjective optimization. In *Multiobjective Optimization*, Vol. 5252 of *Lecture Notes in Computer Science*, pp. 349–372.

Yijie, S., Gongzhang, S., 2008. Improved NSGA-II multi-objective genetic algorithm based on hybridization-encouraged mechanism. *Chinese Journal of Aeronautics*, **21**(6), pp. 540–549.