Evaluation of parallel metaheuristics

Enrique Alba and Gabriel Luque

Grupo GISUM, Departamento de LCC
E.T.S.I. Informática
Campus Teatinos, 29071 Málaga (Spain)
{eat,gabriel}@lcc.uma.es

Abstract. When evaluating algorithms a very important goal is to perform better than the state-of-the-art techniques.. This requires experimental tests to compare the new algorithm with respect to the rest. It is, in general, hard to make fair comparisons between algorithms such as metaheuristics. The reason is that we can infer different conclusions from the same results depending on the metrics we use and how they are applied. This is specially important for non-deterministic methods. This analysis becomes more complex if the study includes parallel metaheuristics, since many researchers are not aware of existing parallel metrics and their meanings, especially concerning the vast literature on parallel programming used well before metaheuristics were first introduced. In this paper, we focus on the evaluation of parallel algorithms. We give a clear definition of the main parallel performance metrics and we illustrate how they can be used.

1 Introduction

Most optimization tasks found in real world applications impose several constraints that usually do not allow the utilization of exact methods. The complexity of these problems (often they are NP-hard [1]) or the limited resources available to solve them (time, memory) have made the development of metaheuristics a major field in operations research, computer science, and other domains. In particular, parallel versions of metaheuristics have allowed to solve realistic scenarios for industrial problems, since parallelism is an approach not only to reduce the run time but also to improve the quality of the solutions [2].

Unlike exact methods, where time-efficiency is a main measure for success, there are two chief issues in evaluating parallel metaheuristics: how fast can solutions be obtained, and how far they are from the optimum. Besides, we can distinguish between two different approaches for analyzing metaheuristics: a theoretical analysis (worst-case analysis, average-case analysis, ...) or an experimental analysis. The difficulty of theoretical analysis on parallel metaheuristics makes hard to obtain results for most realistic problems and algorithms, and severely limits their range of application. As a consequence most of the metaheuristics are evaluated *empirically* in a *ad-hoc* manner.

In this paper, we focus on how the experiments should be performed, and how the results should be reported in order to make fair comparisons when dealing with parallel metaheuristics. We are interested in revising, proposing, and applying parallel performance metrics and guidelines to ensure the correctness of our conclusions.

This paper is organized as follows. The next section summarizes some parallel metrics such as speedup and related measures. Section 3 discusses some inadequate utilizations of parallel metaheuristics measures. In the following section, we perform several practical experiments to illustrate the importance of a metric in the conclusions obtained. Finally, some remarks are outlined in Section 5.

2 Parallel Performance Measures

There are different metrics to measure the performance of parallel algorithms. In the first subsection we discuss in detail the most common measure, i.e., the speedup, and address its meaningful utilization in parallel metaheuristics. Later, in a second subsection we summarize other metrics also found in the literature.

2.1 Speedup

The most important measure of a parallel algorithm is the *speedup* (and maybe its normalized version: the *efficiency*). This metric computes the ratio between sequential and parallel times. Therefore, the definition of time is the first aspect we must face. In a uni-processor system, a common performance is the *CPU time* to solve the problem; this is the time the processor spends executing algorithm instructions. In the parallel case, time is not a sum of CPU times on each processor, neither the largest among them. Since the objective of parallelism is the reduction of the real-time, time should definitely include any incurred overhead because it is the price of using a parallel algorithm. Hence the most prudent and spread choice for measuring the performance of a parallel code is the *wall-clock time* to solve the problem at hands, as has been largely reported for parallel programs in computer science.

The speedup compares the serial time against the parallel time to solve a particular problem. If we denote with T_m the execution time for an algorithm using m processors, the speedup is the ratio between the (greater) execution time on one processor T_1 and the (smaller) execution time on m processors T_m:

$$s_m = \frac{T_1}{T_m} \tag{1}$$

For non-deterministic algorithms we cannot use this metric directly. For this kind of methods, the speedup should instead compare the *mean* serial execution time against the *mean* parallel execution time:

$$s_m = \frac{E[T_1]}{E[T_m]} \tag{2}$$

With this definition we can distinguish among: *sublinear* speedup ($s_m < m$), *linear* speedup ($s_m = m$), and *superlinear* speedup ($s_m > m$). The main difficulty with that measure is that researchers do not agree on the meaning of T_1 and T_m. Several speedup taxonomies have been proposed in the literature [3,4] For example, Alba [3] distinguishes between several definitions of speedup depending of the meaning of these values (see Table 1).

Strong speedup (type I) compares the parallel run time against the best-so-far sequential algorithm. This is the most exact definition of speedup, but, due to the difficulty of finding the current most efficient algorithm in the world, most designers of parallel algorithms do not use it. *Weak speedup* (type II) compares the parallel algorithm developed by a researcher against his/her own serial version, a situation that is usually affordable in practice. In these two cases, two stopping criteria for the algorithms could be used: solution quality or maximum effort. In addition, two variants of the weak speed with solution stop are possible: to compare the parallel algorithm against the canonical sequential version (type II.A.1) or to compare the run time of the parallel algorithm on one processor against the time of the same algorithm on m processors (type II.A.2).

Table 1. Taxonomy of speedup measures proposed by Alba [3].

I. Strong speedup
II. Weak speedup
A. Speedup with solution stop
1. Versus panmixia
2. Orthodox
B. Speed with predefined effort

Once we have a taxonomy, the open problem is to select a fair way of computing speedup. Since it is supposed to measure the gains of using more processors, it is clear that the parallel metaheuristic should compute a similar accuracy as the sequential one. Only in this case we are allowed to compare times. Also, the used times are average run times: the parallel code on one machine versus the parallel code on m machines. All this define a sound way for comparisons, both practical (no best algorithm needed) and orthodox (same codes, same accuracy). Currently, there is not a common guidelines in performance measurements and researchers use different metrics or the same measures but different meanings, so we will further illustrate the drawbacks of non orthodox metrics in Section 3.

2.2 Other Parallel Metrics

Although the speedup is a widely used metric, there exist other measures of the performance of a parallel metaheuristic. The *efficiency* (Equation 3) is a normalization of the speedup ($e_m = 1$ means linear speedup).

$$e_m = \frac{s_m}{m} \tag{3}$$

Several variants of the efficiency metric exist. For example, the *incremental efficiency* (Equation 4) shows the fraction of time improvement got from adding another processor, and it is also much used when the uni-processor times are unknown. This metric has been later generalized (Equation 5) to measure the improvement attained by increasing the number of processors from n to m.

$$ie_m = \frac{(m-1) \cdot E[T_{m-1}]}{m \cdot E[T_m]} \tag{4}$$

$$gie_{n,m} = \frac{n \cdot E[T_n]}{m \cdot E[T_m]} \tag{5}$$

The previous metrics indicate the improvement coming from using additional processing elements, but they do not measure the utilization of the available memory. The *scaleup* addresses this issue and allows to measure the full utilization of the hardware resources:

$$su_{m,n} = \frac{E[T_{m,k}]}{E[T_{nm,nk}]} \tag{6}$$

where $E[T_{m,k}]$ is the mean run time needed by a platform with m processors to solve a task of size k and $E[T_{nm,nk}]$ is the mean run time needed by a platform with nm processors to solve a task of size nk. Therefore, this metric measures the ability of the algorithm to solve a n-times larger job on a n-times larger system in the same time as the original system. Therefore, linear speedup occurs when $su_{m,n} = 1$.

Finally, Karp and Flatt [5] have devised an interesting metric for measuring the performance of any parallel algorithm that can help us to identify much more subtle effects than using speedup alone. They call it the *serial fraction* of the algorithm (Equation 7).

$$f_m = \frac{1/s_m - 1/m}{1 - 1/m} \tag{7}$$

Ideally, the serial fraction should stay constant for an algorithm. If a speedup value is small we can still say that the result is good if f_m remains constant for different values of m, since the loss of efficiency is due to the limited parallelism of the program. On the other side, smooth increments of f_m are a warning that the granularity of the parallel tasks is too fine. A third scenario is possible in which a significant reduction in f_m occurs as m enlarges, indicating something akin to superlinear speedup. If this occurs, then f_m would take a negative value.

3 Inadequate Utilization of Parallel Metrics

The objective of a parallel metaheuristic is to find a *good* solution in a *short* time. Therefore, the choice of performance measures for these algorithms necessarily involves both solution quality and computational effort. In this section, we discuss some scenarios, where these metrics are incorrectly used, and we propose a solution to these situations.
Computational effort evaluation: Many researchers prefer the number of evaluations as a way to measure the computational effort since it eliminates the effects of particular implementations,

software, and hardware, thus making comparisons independent from such details. But this measure can be misleading in several cases in the field of parallel methods. Whenever the standard deviation of the average fitness computation is high, for example, if some evaluations take longer than others (parallel genetic programming [6]) or if the evaluation time is not constant, then the number of evaluations does not reflect the algorithm's speed correctly. Also, the traditional goal of parallelism is not only the reduction of the number of evaluations but the reduction of time. Therefore, a researcher must often use the two metrics to measure this effort.

Comparing means/medians: In practice, a simple comparison between two averages or medians (of time, of solutions quality, ...) might not give the same result as a comparison between two statistical distributions. In general, it is necessary to offer additional statistical values such as the variance, and to perform a global statistical analysis to ensure that the conclusions are meaningful and not just random noise. The main steps are the following: first, a normality test (e.g., Kolmogorov-Smirnov) should be performed in order to check whether the variables follow a normal distribution or not. If so, an Student t-test (two set of data) or ANOVA test (two or more set of data) should be done, otherwise we should perform a non parametric test such as Kruskal-Wallis. Therefore, the calculation of speedup is only adequate when the execution times of the algorithms are *statistically different*. This two-step procedure also allows to control the I type error (the probability of incorrectly rejecting the null hypothesis when it is true), since the two phases are independent (they test for different null hypotheses).

Comparing algorithms with different accuracy: Although this scenario can be interesting in some specific applications, the calculation of the speedup or a related metric is not correct because it is against the aim of the speedup to compare algorithms not yielding results of equal accuracy, since the two algorithms are actually solving two different problems (i.e., it is nonsense, e.g., to compare a sequential metaheuristic solving TSP of 100 cities against its parallel version solving a TSP of 50 cities). Solving two different problems is what we have in speedup if the final accuracy is different in sequential and parallel.

Comparing parallel versions vs. canonical serial one: Several works compare the canonical sequential version of an algorithm (e.g., a GA) against a parallel version (e.g., a distributed [2]). But these algorithms have a different behavior and therefore, we are comparing clearly different methods (as meaningless as comparing times of a sequential SA versus a parallel GA).

Using a predefined effort: Imposing a predefined time/effort and then comparing the solution quality of the algorithms is an interesting and correct metric in general; what it is incorrect is to use it to measure speedup or efficiency (although works making this can be found in literature). On the contrary, these metrics can be used when we compare the average time to a given solution, defined over those runs that end in a solution (with a predefined quality maybe different from the optimal one). Sometimes, the average evaluations/time to termination is used instead of the average evaluations/time to a solution of a given accuracy. This practice has clear disadvantages, i.e., for runs finding solutions of different accuracy, using the total execution effort to compare algorithms becomes hard to interpret from the point of view of parallelism.

4 Illustrating the Influence of Measures

In this section we perform several experimental tests to show the importance of the reported performances in the conclusions. We use two parallel methods to solve the well-known MAXSAT problem [1]. This problem consists in finding an assignment to a set of boolean variables such that all the clauses of a given formula are satisfied. In the experiments we use several instances generated by De Jong [7]. These instances are composed of 100 variables and 430 clauses ($f(opt) = 430$, where f is the number of satisfied clauses).

The algorithms used are a parallel distributed genetic algorithm and a parallel simulated annealing. Genetic algorithms (GA) [8] iteratively enhance a population of tentative solutions through a natural evolution process. In the experiments, we use a parallel distributed GA (dGA)[9]. In this parallel model, the population is structured into smaller subpopulations relatively isolated from the others that occasionally exchange solutions. A simulated annealing (SA) [10] is a stochastic technique that can be seen as a hill-climber with an internal mechanism to escape from local optima. In our parallel SA (pSA) there exists multiple asynchronous component SAs. Each component SA periodically exchanges the best solution found (cooperation phase) with its neighbor SA.

4.1 Experiment Results

In our tests, we focus on the quality of found solutions and the execution time of a dGA and a pSA to solve one MAXSAT instance [7]. The values shown in tables are the number of executions that found the optimal value (**% hit**), the average fitness (**avg**), the number of evaluations (**# evals**), and the running time (**time**). To assess the statistical significance of the results we performed 100 independent runs and we have computed statistical analyses so that we could be able to distinguish meaningful differences in the average values.

Panmictic vs. orthodox speedup: In our first experiment (Table 2), we show an example on speedup. We show the results for a sequential SA against a parallel SA with different number of processors. We focus on two definitions of speedup: orthodox (**orthodox**) and panmictic (**panmixia**), which were explained in Section 2.1. First, we can observe that we obtain a very good speedup (even superlinear for two and four processors) using the panmictic definition of speedup. But this comparison is not fair, since we are comparing two clearly different methods. In fact, the mean solution quality and the number of points of the search space explored by the canonical sequential SA and the parallel ones are statistically different (all the $p-$values smaller than 0.05). Hence, we compare the same algorithm (orthodox speedup) both in sequential and in parallel (pSAn on 1 versus n processors). The orthodox values are slightly worse than those on the panmictic ones but fair and realistic. The trends in both cases are similar (in some other algorithms the trends could even be contradictory); the speedup is quite good but it is always sublinear and it slightly moves away from the linear speedup as the number of CPUs increases which is a confirmation of what happens in parallel programming usually. From this experiment, it is clear that panmictic definition of speedup is not adequate since it compares two different methods, and in the following examples, we will only focus on the orthodox speedup.

Table 2. Panmixia vs. orthodox speedup.

Alg.	% hit	avg	# evals	time	s_m panmixia	s_m orthodox
Seq.	66%	425.76	894313	18.26	-	-
pSA2	88%	427.98	649251	8.89	2.06	1.89
pSA4	85%	427.82	648914	4.52	4.03	3.78
pSA8	87%	428.06	651725	2.55	7.16	6.94

Predefined effort: In our second example (Table 3), the termination condition is based on a predefined effort (200,000 evaluations) and not in finding a given solution in parallel and sequential. In this example, algorithms (pSA and dGA) did not find any optimal solution in any execution. Then we cannot use the percentage of hits to compare them: we must stick to another metric to compare the quality of solutions. We could use the best solution found, but that single value does not represent the actual behavior of the algorithms. In this case the best measure to compare the quality of results is the mean fitness, but in these experiments the differences in this value are negligible. Therefore, we can not state the superiority of any of them. The calculation of the speedup is not always a good idea due to two main issues. Firstly, accuracy of the solutions found is different; and secondly, we have set a predefined number of evaluation, and therefore, we have also fixed the execution time ($time = c \cdot t_{eval} \cdot eval$). In fact, we can calculate these constants and, for example, the algorithm **dGA8** executed on 1 processor has $c_1 = 3.4 \cdot 10^{-5}$ and the same method executed on a parallel platform with 8 machines has $c_8 = 4.3 \cdot 10^{-6}$. Using this information, we can choose the configurations of the methods to obtain any speedup that we happen to like, thus biasing the result. In addition, we can obtain a theoretical linear speedup ($s_m = 8$) if we compare a **dGA8** (8 processors) with 200,000 evaluations and a **dGA8** (1 processor) with 202,350 evaluations. This prediction is quite accurate since these configurations has a experimental speedup equal to 7.993.

Table 3. Predefined effort.

Alg.		% hit	avg	# evals	time	s_m
pSA8	$proc = 1$	0%	424.13	200000	6.89	-
	$proc = 8$	0%	424.09	200000	0.87	7.92
dGA8	$proc = 1$	0%	424.82	200000	9.79	-
	$proc = 8$	0%	424.86	200000	1.24	7.90

Other parallel metrics: We conclude this section showing (in Table 4) the results when the stopping condition is to find the global optimum (or when the global population has converged). In this experiment, we have also included in our analysis the efficiency (e_m column) and the serial fraction (f_m column) using the times of the runs finding a solution. First, we notice that the parallel versions improve the solutions found by the serial one and located them in a smaller number of evaluations. Once we have analyzed the numerical performance, we can focus on execution times and parallel metrics. We observed that the parallel methods obtain quite good speedups with a moderate loss of efficiency when the number of processor is increased. Here, the serial fraction metric plays an important role since the variation in this measure is negligible, indicating that this loss of efficiency is mainly due to the limited parallelism (intrinsic sequential part) of the program and not a result of a (own) poor implementation.

Table 4. Other parallel metrics.

Alg.	% hit	avg	# evals	time	s_m	e_m	f_m
Seq.	73%	427.23	429723	11.48	-	-	-
dGA2	90%	428.10	397756	5.87	1.93	0.96	0.036
dGA4	94%	428.54	401505	3.18	3.65	0.91	0.032
dGA8	94%	428.31	380192	1.89	6.43	0.80	0.035

5 Conclusions

This paper considered the issue of reporting experimental research with parallel metaheuristics. Since this is a difficult task, the main issues of an experimental design are highlighted.

As it could be expected, we have focused on parallel performance metrics that allow to compare parallel approaches against other techniques in the literature. We have given some ideas to guide researchers in their work.

Finally, we have performed several experimental tests to illustrate the influence and utilization of the many metrics described in the work. We have analyzed different scenarios and we have observed that the importance of a fair metric is a key factor to ensure the correctness of the conclusions.

Acknowledgments

The authors acknowledge partial funding by Spanish Ministry of Education and European FEDER under contract TIN2005-08818-C04-01 (the OPLINK project, `http://oplink.lcc.uma.es`).

References

1. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, San Francisco (1979)

2. Alba, E., ed.: Parallel Metaheuristics: A New Class of Algorithms. Wiley (2005)
3. Alba, E.: Parallel evolutionary algorithms can achieve super-linear performace. Information Processing Letters **82** (2002) 7–13
4. Barr, R., Hickman, B.: Reporting Computational Experiments with Parallel Algorithms: Issues, Measures, and Experts' Opinions. ORSA Journal on Computing **5**(1) (1993) 2–18
5. Karp, A., Flatt, H.: Measuring parallel processor performance. Communications of the ACM **33**(5) (1990) 539–543
6. Koza, J.R.: Genetic Programming. The MIT Press, Cambridge (1992)
7. Jong, K.A.D., Potter, M.A., Spears, W.M.: Using problem generators to explore the effects of epistasis. In: 7th ICGA, Kaufman (1997) 338–345
8. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley (1989)
9. Tanese, R.: Distributed genetic algorithms. In Schaffer, J.D., ed.: Proceedings of the Third ICGA, Morgan Kaufmann (1989) 434–439
10. Kirkpatrick, S., Gelatt, C., Vecchi, M.: Optimization by Simulated Annealing. Science **220**(4598) (1983) 671–680