On Asynchronous Non-Dominated Sorting for Steady-State Multiobjective Evolutionary Algorithms

Ilya Yakupov
ITMO University
Saint-Petersburg, Russia
iyakupov93@gmail.com

Maxim Buzdalov
ITMO University
Saint-Petersburg, Russia
mbuzdalov@gmail.com

ABSTRACT

In parallel and distributed environments, generational evolutionary algorithms often do not exploit the full potential of the computation system since they have to wait until the entire population is evaluated before starting selection procedures. Steady-state algorithms can perform fitness evaluations asynchronously, however, if the algorithm updates its state in a complicated way – which is common in multiobjective evolutionary algorithms – the threads will eventually have to wait until this update finishes.

The most expensive part of the update procedure in NSGA-II is non-dominated sorting. We turned the existing incremental non-dominated sorting algorithm into an asynchronous one using several concurrency techniques: a single entry-level lock, finer-grained locks on non-domination levels, and a non-blocking approach using compare-and-set operations. Our experimental results reveal the trade-off between the work-efficiency of the algorithm and the achieved amount of parallelism.

CCS CONCEPTS

• **Theory of computation → Divide and conquer**; *Sorting and searching*; • **Applied computing → Multi-criterion optimization and decision-making**;

KEYWORDS

Non-dominated sorting, steady-state algorithms, concurrency

ACM Reference Format:
Ilya Yakupov and Maxim Buzdalov. 2018. On Asynchronous Non-Dominated Sorting for Steady-State Multiobjective Evolutionary Algorithms. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3205651.3205802

1 INTRODUCTION

Without loss of generality, we consider multiobjective minimization problems. Due to the topic of the paper, we always treat individuals as points in the k-dimensional objective space.

A point p is said to *(strictly) dominate* a point q, denoted as $p \prec q$, if in every coordinate p is not greater than q, and there exists a coordinate where it is strictly smaller. *Non-dominated sorting* is a procedure that takes a set of points and assigns each point a *rank*. The points that are not dominated by any other points get rank 0. All points that are dominated only by points of rank $\le i$ get rank $i + 1$. A set of points with the same rank is called a *level*. The procedure was defined in [5], an $\Theta(n^2k)$ algorithm for n points was proposed in [2] along with NSGA-II. Our work is based on a divide-and-conquer algorithm [1].

Incremental non-dominated sorting is a procedure that updates ranks of a set of points when a new point is inserted or deleted. There are several algorithms to perform incremental non-dominated sorting [4, 6], of which the one from [6] currently has the best performance among the ones for arbitrary dimension k. This algorithm maintains the levels in separate lists. On insertion of a point p, first the maximum number of level ℓ is found which contains a point that dominates p. Then, a set of moving points M is formed, initially $M = \{p\}$. An algorithm from [1] is then run on $L_{\ell+1} \cup M$. Since both M and $L_{\ell+1}$ are both non-dominating sets, and no point from $L_{\ell+1}$ can dominate a point from M, the rank of each point will be either 0 or 1. The points of rank 0 form the new level $L_{\ell+1}$, rank 1 forms the new M, and then the process continues with $\ell \leftarrow \ell + 1$. The existence of only two ranks, 0 or 1, improves the performance of the algorithm from [1] by roughly $O(\log n)$. The fact that points from $L_{\ell+1}$ can never dominate points from M also saves some work.

We use *locks* and the *compare-and-set* concurrency primitive.

2 THE ALGORITHMS

In this section we describe the ways to introduce concurrency into the incremental non-dominated sorting (INDS).

The **baseline method**, which we denote as Sync, is to put a global lock on the entire state of the algorithm.

In the **basic compare-and-set approach**, which we denote as CAS1, we let the threads perform level updates locally and publish the results in the case no other thread had updated this level before. Each level is updated atomically. As we cannot ensure anymore that no point $p \in L_\ell$ can dominate any point $m \in M$ from the set of moving points M, we use the full-blown offline non-dominated sorting to determine the new contents of L_ℓ. Once the thread is done, it performs the compare-and-set with the current state of L_ℓ. If it fails, it performs insertion again, otherwise it continues.

The **time-stamping modification**, which we denote as CAS2, tries to use the benefits offered by a faster merging of levels in [6] whenever possible. We maintain a global atomic integer timer which is incremented at the beginning of each point insertion, and also on creation of a new version of a level. Each level contains a time-stamp of its last modification. Before inserting a point, we save the timer value τ. If the set M is about to be merged with the level L_ℓ, and the time-stamp $T(L_\ell)$ is still less than τ, then this level was not

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07.
https://doi.org/10.1145/3205651.3205802

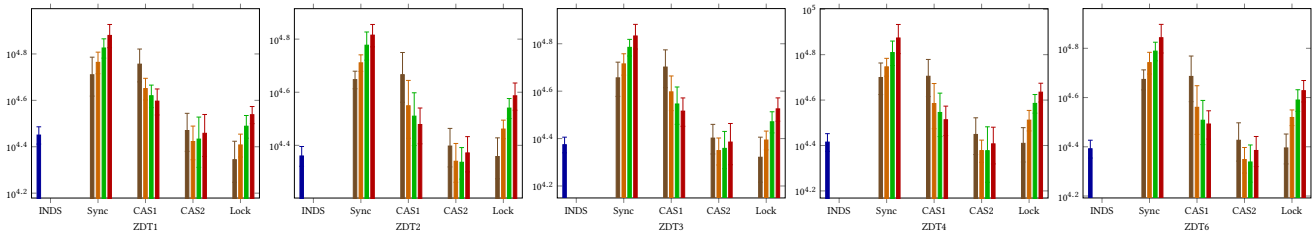

Figure 1: Experiments with ZDT problems. For all the problems, $k = 2$. For every asynchronous algorithm the columns correspond to the number of threads. They are, left-to-right, 3, 6, 12, 24. Times are given in microseconds.

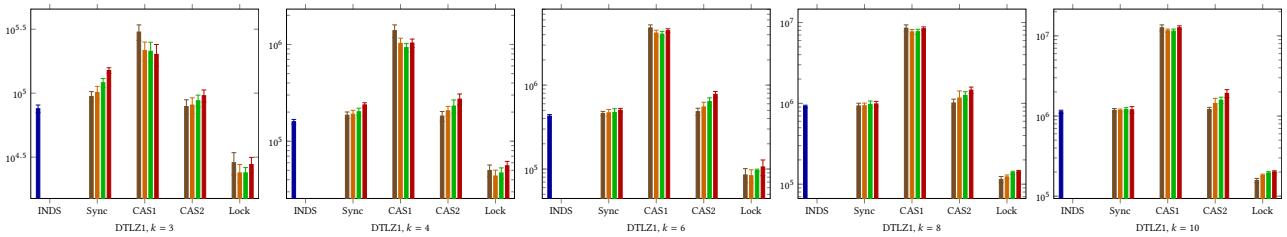

Figure 2: Experiments with DTLZ1 problem with varying k.

modified by any thread, so no point $p \in L_\ell$ can dominate any point $m \in M$ and the faster merge is possible.

Finally, the **fine-grained lock approach**, which we denote as Lock, associates a lock with each level. When performing an update of the level L_ℓ by a set of moving points M, the thread acquires a lock K_ℓ associated with the updated level, so no other thread can modify the level L_ℓ. Just before the lock K_ℓ is released, the thread acquires a lock $K_{\ell+1}$ associated with the next level $L_{\ell+1}$ if the new set of moving points M is not empty. This ensures that no thread can surpass another one. This also ensures that points from $L_{\ell+1}$ cannot dominate points from the new version of M.

3 EXPERIMENTS

We evaluated the algorithms on the well-known benchmark problems DTLZ1 [3], ZDT1–ZDT4 and ZDT6 [7]. For the ZDT problems, we kept $k = 2$. For the DTLZ1 problem, $k \in \{3, 4, 6, 8, 10\}$.

The datasets were synthesized for each problem by creating a random population of size 5000, running a steady-state NSGA-II for 1000 iterations and recording the points to be inserted. Each of the algorithms was run on these datasets, and times for point insertions, each followed by deletion of the worst point, were measured. For all algorithms except the sequential INDS, t threads, $t \in \{3, 6, 12, 24\}$, were used to insert the points.

The results on the ZDT problems (Figure 1) reveal that there is no benefit in asynchronous algorithms when the insertion time is small. Thread contention in the Sync algorithm introduces slowdowns of orders of magnitude. The algorithms based on the compare-and-set mechanism scale rather well in these conditions, as CAS1 performs better with more threads. CAS2 is initially rather fast and is competitive with INDS. The Lock algorithm degrades with more threads, but it is better than Sync and can be on par with INDS.

With more than two dimensions, the cost of a single insertion increases (Figure 2). In these settings, the performance of Sync, is still much worse compared to INDS, but not to the scale observed on the ZDT problems. The performance of CAS1 becomes much worse even compared to Sync. We explain this by increased complexity of non-dominated sorting, the results of which are often wasted due to compare-and-sets. CAS2 is relatively efficient, however, still worse than INDS. The biggest surprise is the Lock algorithm, which demonstrates roughly the same performance as in two dimensions and thus overcomes INDS in the performance. A possible explanation for such a good behavior of Lock can be that, after a short initial phase, the threads follow each other in a fixed order.

This research was supported by the Russian Scientific Foundation, agreement No. 17-71-20178. The full text is available at arXiv[1].

REFERENCES

[1] Maxim Buzdalov and Anatoly Shalyto. 2014. A Provably Asymptotically Fast Version of the Generalized Jensen Algorithm for Non-Dominated Sorting. In *Parallel Problem Solving from Nature – PPSN XIII*. Number 8672 in Lecture Notes in Computer Science. Springer, 528–537.

[2] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multi-Objective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[3] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. 2005. Scalable Test Problems for Evolutionary Multiobjective Optimization. In *Evolutionary Multiobjective Optimization. Theoretical Advances and Applications*. Springer, 105–145.

[4] Ke Li, Kalyanmoy Deb, Qingfu Zhang, and Qiang Zhang. 2017. Efficient Non-domination Level Update Method for Steady-State Evolutionary Multiobjective Optimization. *IEEE Transactions on Cybernetics* 47, 9 (2017), 2838–2849.

[5] N. Srinivas and Kalyanmoy Deb. 1994. Multiobjective Optimization Using Non-dominated Sorting in Genetic Algorithms. *Evolutionary Computation* 2, 3 (1994), 221–248.

[6] Ilya Yakupov and Maxim Buzdalov. 2017. Improved Incremental Non-dominated Sorting for Steady-State Evolutionary Multiobjective Optimization. In *Proceedings of Genetic and Evolutionary Computation Conference*. 649–656.

[7] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. 2000. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation* 8, 2 (2000), 173–195.

[1] http://arxiv.org/abs/1804.05208