ParallelGlobal with Low Thread Interactions

Dániel Zombori
Department of Computational Optimization
University of Szeged
zomborid@inf.u-szeged.hu

Dr. Balázs Bánhelyi
Department of Computational Optimization
University of Szeged
banhelyi@inf.u-szeged.hu

ABSTRACT

Global is an optimization algorithm conceived in the '80s. Since then several papers discussed improvements of the algorithm, but adapting it to a multi-thread execution environment is only a recent branch of development [1]. Our previous work focused on parallel implementation on a single machine but sometimes the use of distributed systems is inevitable. In this paper we introduce a new version of Global which is the first step towards a fully distributed algorithm. While the proposed implementation still works on a single machine, it is easy to see how gossip based information sharing can be built into and be utilized by the algorithm. We show that ParallelGlobal is a feasible way to implement Global on a distributed system. However, further improvements must be made to solve real world problems with the algorithm.

Categories and Subject Descriptors

[**Computing methodologies**]: Optimization algorithms; [**Computing methodologies**]: Parallel algorithms

1. INTRODUCTION

Global is an optimization algorithm built from multiple modules working in an ensemble. While older implementations viewed the algorithm as a whole, the most recent GlobalJ framework handles algorithms as a collection of interlocking modules. GlobalJ has several implementations for local search algorithms and variants of Global. Main characteristics of the single threaded version were established in [4]. In recent years Global was further developed [6] and it has several applications [5, 10] where it aids mostly other research works. To speed up optimization processes we developed an algorithm [1] that is capable of utilizing multiple computational threads of a single machine. It cannot be directly implemented for distributed systems as the millisecond order of magnitude latency in communication would significantly slow down the synchronization of threads. To mitigate this problem we propose ParallelGlobal, a parallel implementation suitable for distributed systems with high latency or even with unreliable communication channels. In this paper we introduce an experimental version whose main purpose is to test the feasibility of the proposed solution. It provides an algorithm skeleton for a real distributed implementation.

2. GLOBAL

Global is a global optimizer designed to solve black box unconstrained optimization problems with low number of function evaluations and probabilistic guarantees [1, 2, 3, 4, 6, 7, 8, 11]. It uses local search algorithms to refine multiple sample points hence Global is a multi-start method. Global also utilizes the *Single Linkage Clustering* algorithm to make an estimation about the value of samples from the aspect of optimization.

2.1 Updated Global Algorithm

While the updated Global algorithm has only minor changes and in a lot of cases performs equally to the original, it is superior in execution order, therefore we consider it as the basis for improvements.

Global has an iterative framework where samples in an iteration compete with samples of previous iterations. The original version contains four phases in every iteration consisting of sampling, reduction, clustering and local search. In the updated algorithm the clustering and local search phases are merged by an implementation alternating between the two.

Algorithm 1 describes the updated Global in detail. In lines 2-5 the algorithm performs the sampling phase. Selection of sample points is stochastic, using uniform distribution in the search space. The generated samples are placed in container S which is a list structure. To find the most promising samples, S is sorted and a reduced set of samples is acquired with the lowest function values. R contains the reduced set, which is removed from S.

When samples are ready to be processed, in lines 6-24 the algorithm alternates between clustering and local searches while there are unprocessed samples left. At 7-15 samples in R are tried against the clustered samples. To determine if $r_i \in R$ is part of cluster C we need the distance threshold d_c. d_c depends on the dimension of the objective function, the number of samples currently known in the clustering process and the $\alpha \in [0, 1]$ parameter. The latter controls the decrease speed of d_c while more samples are added, in order

Algorithm 1 GLOBAL

1: **while** *termination-criteria*() **is not** *true* **do**
2: $S \leftarrow S \cup \{n_i = uniform(lb, ub) : i \in [1, new\ samples]\}$
3: $S \leftarrow sort(F(s_i) < F(s_{i+1})), s_i \in S$
4: $R \leftarrow \{s_i : i \in [1, reduced\ set\ size]\}$
5: $S \leftarrow S \setminus R$
6: **while** R **is not** $\emptyset$ **do**
7: **for** C **in** *clusters* **do**
8: $d_c \leftarrow \left(1 - \alpha^{\frac{1}{|clustered|+|R|-1}}\right)^{\frac{1}{dim(F)}}$
9: $N \leftarrow \left\{r_i : d_c > \|r_i - c_j\|_\infty \wedge F(r_i) > F(c_j)\right\}$
10: **if** N **is not** $\emptyset$ **then**
11: $C \leftarrow C \cup N$
12: $R \leftarrow R \setminus N$
13: **repeat iteration**
14: **end if**
15: **end for**
16: $l \leftarrow local\text{-}search(r_1 \in R)$
17: $C_l, d_{min} \leftarrow \underset{C \in clusters}{argmin} \left\| l - \underset{c_i \in C,}{argmin}\ F(c_i) \right\|_\infty$
18: **if** $d_{min} < d_c/10$ **then**
19: $C_l \leftarrow C_l \cup \{l, r_1\}$
20: **else**
21: $clusters \leftarrow clusters \cup \{\{l, r_1\}\}$
22: **end if**
23: $R \leftarrow R \setminus \{r_1\}$
24: **end while**
25: **end while**

to adapt to the expected decrease in distance between two random samples. With d_c set, sample pairs $(r_i \in R, c_j \in C)$ are evaluated to determine if r_i is part of C. The two criteria are having a clustered sample c_j with lower function value than r_i and it being closer with the infinity norm (Manhattan distance) than d_c. Samples in R satisfying both of them are moved to the current cluster C. When a sample is clustered, all samples in R can potentially be clustered too therefore $r_i \in R$ is rechecked against C. After the *for* cycle finished, samples in R cannot be the part of an existing cluster therefore performing a local search is inevitable.

Local searches are performed in lines 16-23, where l is the local optimum reached from r_1. To determine if l is a newly found local optimum a comparison with the cluster centers is needed. The center of a cluster is the sample in the cluster with the lowest function value. By finding the cluster with the closest center the algorithm can decide if the optimum is already found. If the distance d_{min} to the cluster C_l with the closest center is lower than a tenth of the d_c threshold, it is considered the same local optimum. In this case l and r_1 are added to C_l, otherwise they form a new cluster. Since r_1 is either in an already existing cluster or in a newly created one, we can remove it from R. Lines 6-24 are repeated until R becomes empty. With no unclustered samples left Global finished an iteration. The number of executed iterations is limited by the termination criteria.

3. PARALLEL GLOBAL

Our goal is to derive an implementation from the updated Global which is multi-threaded with low interactions between threads. The necessity for low thread interactions comes from the fact that on huge scale optimization tasks a single computer is not sufficient and in multi-computer environments the communication between machines is relatively slow compared to inter-thread communication. We address this problem by removing the synchronization of computational threads and replacing it with a message based information sharing scheme.

We can view ParallelGlobal as a naive parallelization of Global. The main idea lies in the parallel execution of Global iterations, while sharing information between computational threads. Consequently, inter-thread communication is necessary, however only a few selected data containers have to be shared. Also, the shared containers have independent data points and no deletions, therefore inconsistencies cannot arise from data insertions. These considerations make the algorithm for distributed systems viable.

3.1 ParallelGlobal Worker

Algorithm 2 describes the ParallelGlobal worker which is the implementation of a single computational thread. The worker might run on a machine by itself, or multiple workers can use the multi-threaded environment of a computer.

Algorithm 2 ParallelGlobal

1: **while** *termination-criteria*() **is not** *true* **do**
2: *exchange-data*()
3: $s \leftarrow uniform(lb, ub)$
4: $R \leftarrow reduce\,(\{s\})$
5: $d_c \leftarrow \left(1 - \alpha^{\frac{1}{|clustered|+1-1}}\right)^{\frac{1}{dim(F)}}$
6: **for** C **in** *clusters* **do**
7: $N \leftarrow \left\{r_i : d_c > \|r_i - c_j\|_\infty \wedge F(r_i) > F(c_j)\right\}$
8: $C \leftarrow C \cup N$
9: $R \leftarrow R \setminus N$
10: **end for**
11: $l \leftarrow local\text{-}search(r_1 \in R)$
12: $C_l, d_{min} \leftarrow \underset{C \in clusters}{argmin} \left\| l - \underset{c_i \in C}{argmin}\ F(c_i) \right\|_\infty$
13: **if** $d_{min} < d_c/10$ **then**
14: $C_l \leftarrow C_l \cup \{l, r_1\}$
15: **else**
16: $clusters \leftarrow clusters \cup \{\{l, r_1\}\}$
17: **end if**
18: **end while**

Similarly to Global, ParallelGlobal also runs in a loop to complete iterations until a termination criterion is met. Unlike Global, the new algorithm needs a data exchange step (line 2). At the start of every iteration, received messages can be processed and new messages can be sent according to a suitable policy. The messages contain evaluated data points arranged into clusters. These clusters can be handled as if they were evaluated locally by clustering the center point (minimum) of the cluster. If the center point corresponds to an existing cluster, the two clusters should be merged while duplicate points are filtered out. Otherwise, the received cluster describes a previously unknown local optimum and it can be added to the existing clusters without modifications.

In lines 3 and 4 happens the sampling and reduction. In previous Global versions sampling and reduction was performed by taking a randomized sample set, then using a sorted sample pool and taking the best samples out. ParallelGlobal cannot utilize a common pool efficiently due to the distributed nature of the system. In this version, for sim-

plicity we envisioned taking a single sample every iteration and using stochastic sample reduction, possibly aided with spatial measures on the samples information value. A more complex but possible solution would be a distributed sample pool. Samples could be transferred between local pools over reliable data connection. This would ensure that a sample is only evaluated by a single worker and would create a bigger variety of samples to choose from.

In lines 5-10 occurs the clustering. It is very similar to the original clustering algorithm. The only change is that we know that no more than one sample is in R. This is also true for the local search (lines 11-17) which is identical with the original local search part.

3.2 Current implementation

The current implementation of ParallelGlobal only simulates the described functionality with some simplification. First, it runs on a single machine with multiple threads as a single program. Second, messaging is simulated by synchronization on the given containers while they are written, but reading operations happen simultaneously. During clusterization, the cluster list is only read to a point determined before the process starts, hence new clusters will be excluded from already started searches. This also resembles the effects of messaging, like delays and losses in information spread. Because no real messaging is present, the *exchange-data*() function is only a placeholder for now. The *reduce*() function is also a placeholder and the subject of further development. Currently, every sample is evaluated by the clustering and local search steps.

4. RESULTS

The algorithm was examined from two aspects; comparison with the updated Global in the number of function evaluations and scaling of run time with additional threads. Numerical results were obtained on the following functions, definitions can be found in [9]. *Ackley*, *Discus*, *Easom*, *Griewank*, *Levy*, *Rastrigin*, *Schaffer*, *Schwefel*, *Shekel-5*, *Shekel-7*, *Shekel-10*, *Shubert*, *Spikes*[1] and *Zakharov*. For the evaluations we used two termination criteria, the maximum number of function evaluations is 10^5 which is a soft condition therefore overshoot is possible. To check whether an optimum point is reached we use the following expression

$$|F(x^*) - F(x)| < 10^{-8} + |F(x^*)| \cdot 10^{-6}$$

where x^* is a known global optimum point and x is the point in question. To emulate computationally more expensive functions we defined the hardness level. A hardness level of h means that the function will be evaluated 10^h times at the requested point. Global is a stochastic optimizer, moreover ParallelGlobal is also affected by the operating systems thread scheduling, consequently run times and the number of function evaluations can differ largely from one optimization process to the other. To reduce the noise induced by this, we obtained data points by averaging the results of 100 runs with every configuration. The algorithm parameterizations were identical except for the number of threads.

[1]Spikes function definition:

$$f(x) = \begin{cases} 1002 + \Pi_{x_i} sin(2\pi x_i), & \text{if } \|x - (15.25, 15.75)\|_2 > \frac{1}{4} \\ 1000, & \text{otherwise} \end{cases}$$

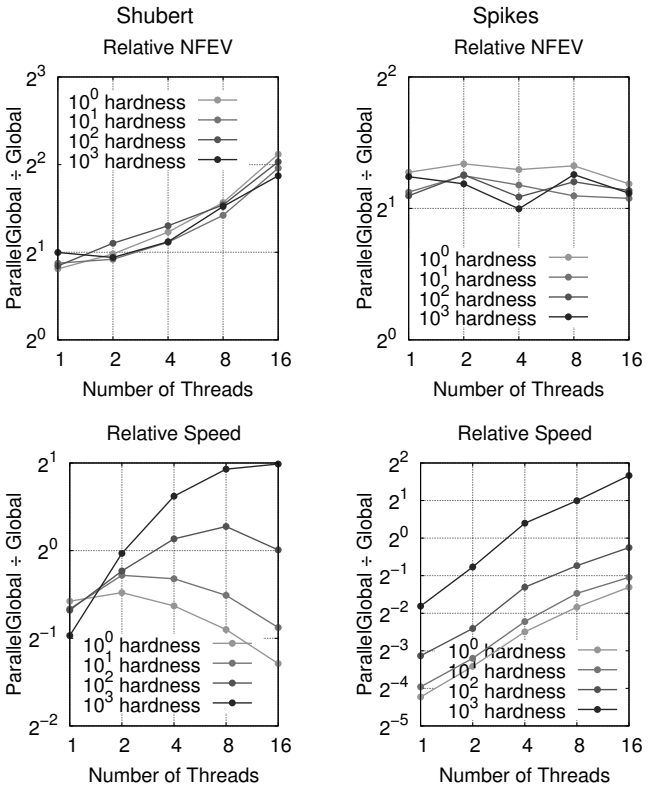

Figure 1: Numeric results on Shubert (left) and Spikes (right) test functions.

On the left side of Figure 1 we show results for the *Shubert* test function, namely the number of function evaluations (NFEV) and the speed of the optimization process, both relative to Global. On the horizontal axes we see the number of threads. The vertical axes show the number of function evaluations and optimization processes run in unit time respectively, both divided by the result of Global on a single core. *Shubert* is a function with many local optima and a flat global trend. In case of Global, NFEV is mostly in the $[500, 2000]$ range with an average of 900. On the top-left graph relative NFEV shows that we have an increase with a factor of two. On a single thread the multiplier of 2 shows that the algorithm is by itself inferior to Global. This static multiplier is explained by the lack of a sample pool which reduces the necessary number of local searches. They create the bulk of the NFEV and while Global uses 1.5 local searches on average ParallelGlobal needs much more. The dynamic growth is also explained by the local searches, combined with multi-threading. Finding the global optimum with local search takes several function evaluations in sequence. Since multiple threads start local searches independently, more evaluations can happen until one of them reaches the global optimum. Moreover when the optimum is found, the program does not terminate immediately, all local searches have to finish. This phenomenon increases the NFEV due to the intrinsic usage of multi-threading and local searches.

The bottom-left graph of Figure 1 shows the speedup with additional threads and different hardness values. While for

hardness 0 and 1 the additional threads caused a slowdown due to synchronization time and increased NFEV, on computationally more demanding versions we achieved a significant speedup. The results are promising because for the hardness value of 3 on a single thread a function evaluation took only 650μs on average. With higher evaluation times, the addition of computational power would have more effect.

On the right side of Figure 1, we show the results for the *Spikes* test function which also has many local optima and a flat global trend. ParallelGlobal suffers from the lack of a sample pool on the *Spikes* function too. On the other hand, no dynamic change in NFEV is experienced. Without a sample pool, ParallelGlobal had a much harder time finding the global optimum, which would often exceed the 10^5 NFEV limit. This resulted in close to constant NFEV and no saturation of threads. Based on the relative speed graph, we gain speed linearly with additional CPU power in every hardness level. Since the function is very cheap to evaluate and ParallelGlobal has to do much more evaluations, only hardness 3 gives an advantage to the multi-threaded implementation.

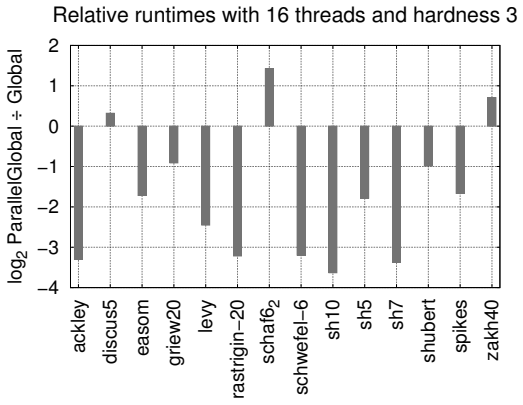

Figure 2: Relative runtimes on all test functions with 16 threads and hardness 3.

On Figure 2 we show relative runtimes for the configuration of 16 threads and hardness 3 on every test function. Since the plot is logarithmic, 0 and values below mean similar and better results compared to Global. On the functions which experienced slowdown either the lack of a sample pool or the intrinsic properties of ParallelGlobal prevented gains in speed. About 50% of the functions with speedup were solved successfully where the NFEV limit had no effects.

5. CONCLUSION

During our work we came to multiple important conclusions about the ParallelGlobal algorithm. The most needed change is the implementation of a distributed sample pool with sample sharing between threads. Having a set of probe points in the search space would ensure that local searches only start from promising regions. This change would probably move the algorithm much closer to the NFEV values of Global.

Many of our results show slowdown with ParallelGlobal, but huge improvements as hardness values increase, *Shubert* function is a good example. To keep our run times manageable we kept the hardness value relatively low. By going up from the current millisecond order to the second or 10 second order in function evaluations we would have a clearer image on how much speedup can we achieve. This would still undershoot the evaluation time of many practical problems, however it would be sufficient for proper testing on distributed systems.

To achieve these changes, first the addition of a distributed framework is needed. Both the sharing of probe samples and cluster information would rely on it. It is also a key for testing on computationally expensive problems.

6. ACKNOWLEDGMENTS

This work was supported by the Hungarian Government under the grant number EFOP-3.6.1-16-2016-00008. The project has been supported by the European Union, co-funded by the European Social Fund, and by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

7. REFERENCES

[1] B. Bánhelyi, T. Csendes, B. Lévai, L. Pál, and D. Zombori. *The GLOBAL Optimization Algorithm.* Springer, 2018.

[2] B. Betró and F. Schoen. Optimal and sub-optimal stopping rules for the multistart algorithm in global optimization. *Mathematical Programming*, 57:445–458, 1992.

[3] C. Boender and A. Rinnooy Kan. On when to stop sampling for the maximum. *Global Optimization*, 1:331–340, 1991.

[4] C. Boender, A. Rinnooy Kan, G. Timmer, and L. Stougie. A stochastic method for global optimization. *Mathematical Programming*, 22:125–140, 1982.

[5] T. Csendes, B. Garay, and B. Bánhelyi. A verified optimization technique to locate chaotic regions of hénon systems. *Journal of Global Optimization*, 35:145–160, 2006.

[6] T. Csendes, L. Pál, J. Sendin, and J. Banga. The global optimization method revisited. *Optimization Letters*, 2:445–454, 2008.

[7] I. Lagaris and I. Tsoulos. Stopping rules for box-constrained stochastic global optimization. *In Applied Mathematics and Computation*, 197:622–632, 2008.

[8] J. Sendín, J. Banga, and T. Csendes. Extensions of a multistart clustering algorithm for constrained global optimization problems. *Industrial & Engineering Chemistry Research*, 48:3014–3023, 2009.

[9] S. Surjanovic and D. Bingham. http://www.sfu.ca/%7essurjano/optimization.html.

[10] A. Szenes, B. Bánhelyi, L. Z. Szabó, G. Szabó, T. Csendes, and M. Csete. Improved emission of siv diamond color centers embedded into concave plasmonic core-shell nanoresonators. *Scientific Reports*, 7:an:13845, 2017.

[11] A. Törn. *A search clustering approach to global optimization*, pages 49–62. Elsevier, North-Holland, 1978.