Parallelization of Enhanced Firework Algorithm using MapReduce

Simone A. Ludwig, Department of Computer Science, North Dakota State University, Fargo, ND, USA

Deepak Dawar, Department of Computer Science, North Dakota State University, Fargo, ND, USA

ABSTRACT

Swarm intelligence algorithms are inherently parallel since different individuals in the swarm perform independent computations at different positions simultaneously. Hence, these algorithms lend themselves well to parallel implementations thereby speeding up the optimization process. FireWorks Algorithm (FWA) is a recently proposed swarm intelligence algorithm for optimization. This work investigates the scalability of the parallelization of the Enhanced FireWorks Algorithm (EFWA), which is an improved version of FWA. The authors use the MapReduce platform for parallelizing EFWA, investigate its ability to scale, and report on the speedup obtained on different benchmark functions for increasing problem dimensions.

Keywords: Fireworks, Map, Reduce, Scalability, Swarm Intelligence

1. INTRODUCTION

Optimization is the process for searching for the best solution from a set of feasible solutions for a given problem with a set of constraints. Real world optimization problems can be challenging, apart from being highly complex and time consuming to solve, a given problem's objective function may also be non-continuous and non-differentiable, which adds another layer of complexity. These real world optimization problems are ubiquitously found in various scientific and engineering domains.

For solving complex real world problems, researchers have been looking into optimization techniques inspired by natural processes such as Darwinian evolution, social group behavior and foraging strategies. Over the past few decades a significant growth in the field of nature-inspired optimization algorithms has been seen. There are two main categories of algorithms: evolutionary computing methods and swarm intelligence algorithms.

DOI: 10.4018/IJSIR.2015040102

The common idea underlying evolutionary algorithms is that given a population of individuals, natural selection (biologically referred to as survival of the fittest) is used to improve the fitness of the overall population. Given a function to be maximized, a set of candidate solutions is randomly created and the fitness function used as a fitness measure (the higher the better) is applied. Based on this fitness measure, some of the better candidates are chosen to undergo recombination and mutation. Recombination is applied to two candidates and results in one or more new candidates, whereas mutation is only applied to one candidate and results in one new candidate. After recombination and mutation are applied a set of new candidates replace the old ones and the next generation begins. This process is iterated until a candidate with sufficient quality is found or a predefined number of iterations is reached (Eiben, 2003).

Swarm intelligence is characterized by the collective behavior of decentralized, self-organized systems that are typically made up of a population of simple agents interacting locally with one another and with their environment (Beni & Wang, 1989). The behavior of swarm optimization can be envisioned by comparing it to swarms searching for optimal food sources, where the direction in which each individual is influenced is its current movement, the best food source it ever experienced, and the best food source any individual in the swarm ever experienced.

There are several different algorithms that fall into the swarm intelligence category. The most famous algorithm is the Particle Swarm Optimization (PSO) algorithm (Eberhart & Kennedy, 1995). The PSO algorithm searches for the best solution by keeping track of the best solution of each particle and weighing this with the best solution of the swarm. Another well-known algorithm is the Ant Colony Optimization (ACO) algorithm (Dorigo, Maniezzo, & Colorni, 1996), which uses the interaction of ants. Similarly, the cuckoo search (Gandomi, Yang, & Alavi, 2013) uses the brooding parasitism of cuckoos, and the bat algorithm (Yang & Gandomi, 2012) uses the echolocation of foraging bats, whereas the bee algorithm (Yang, 2005) uses the foraging behavior of honeybees. Another algorithm that belongs to the swarm intelligence family is the Firefly algorithm (Yang, 2009). The algorithm is inspired by the flashing behavior of fireflies. The primary purpose for a firefly's flash is to act as a signal system to attract other fireflies.

FireWorks Algorithm (FWA) (Tan & Zhu, 2010) is a recently developed swarm intelligence algorithm. It tries to simulate the firework explosion process to find the optimal solution in the search space. A firework is an initial location, which produces sparks (adjacent locations) through a series of explosions. It was reported in (Tan & Zhu, 2010) that FWA significantly outperforms Standard Particle Swarm Optimization (SPSO) and Clonal PSO (Tan & Xiao, 2007). FWA and its variants (Zheng, Xu, & Ling, 2012) have been studied and evaluated on both single and multi-objective optimization problems (Zheng, Song, & Chen, 2013). FWA has also been successfully applied to many real life optimization problems (Janecek & Tan, 2011).

Its initial success notwithstanding, FWA has its own drawbacks. It tends to do poorly on shifted functions, and has a high computational cost relative to other swarm intelligence algorithms. Enhanced FireWorks Algorithm (EFWA) is an improved version of FWA proposed in (Zheng, Janecek, & Tang, 2013) that tries to overcome its shortcomings, and has shown promising results. EFWA brings down the computational cost of FWA and at the same time outperforms FWA significantly.

In this paper, we introduce a parallel version of the EFWA algorithm and evaluation the implementation with a scalability analysis using different benchmark functions and multiple dimensions. The implementation makes use of the MapReduce paradigm that allows the algorithm to scale to thousands of computers. We will evaluate the MapReduce implementation using an available infrastructure that provides us with the necessary computing power. Furthermore, a comparison with a MapReduce PSO implementation is conducted.

The rest of this paper is organized as follows. Section 2 describes related work done over the past few years that improved the FWA algorithm. In Section 3, we describe our MapReduce-based implementation of the EFWA algorithm. Section 4 describes the experimental setup and shows the results obtained, and Section 5 concludes with our findings.

2. RELATED WORK

2.1. Fireworks Algorithm

FWA (Tan & Zhu, 2010) steers the search towards the optimal solution through the generation of sparks (new locations) about a firework (initial location), similar to an actual firework explosion. Each randomly initialized firework is allowed to explode, generating a number of explosion sparks, which along with the parent firework, act as search agents. The explosion is characterized by an amplitude disturbance in selected dimensions for every firework. The amplitude is calculated for every i^{th} firework separately as:

$$A_i = \hat{A} \cdot \left(\frac{f(x_i) - y_{min} + \varepsilon}{\sum_{i=0}^{n} f(x_i) - y_{min} + \varepsilon} \right) \tag{1}$$

The number of sparks generated for each firework is given by:

$$s_i = \hat{m} \cdot \left(\frac{y_{max} - f(x_i) + \varepsilon}{\sum_{i=0}^{n} y_{max} - f(x_i) + \varepsilon} \right) \tag{2}$$

where $\hat{A}$ and $\hat{m}$ are predefined constants, $f(x_i)$ is the current fitness function of firework x_i, and y_{min} and y_{max} are the best and the worst fitness values so far, respectively. ε is a very small number used to avoid the division-by-zero error. It is quite explicit that a better firework will generate more sparks but with smaller amplitude disturbances. The number of sparks generated is bounded by certain restrictions. Afterwards, the sparks (new locations) are generated for each firework according to Algorithm 1.

Then, another type of spark, namely the Gaussian spark, is generated using the parent fireworks according to Algorithm 2.

All the sparks are then evaluated and the ones deemed fit (depends upon the selection model used), are promoted to generate the next round of sparks. FWA is summarized in Algorithm 3.

2.2. GPU-Based Fireworks Algorithm

Graphical Processing Unit (GPU), as the name characterizes, is an electronic unit originally built to accelerate the rendering of images on visual devices and acted as a helper for a CPU (Central Processing Unit). Recently, GPUs are beginning to be used for general purpose computing and the trend has shown rise in popularity (Owens et al., 2007). At the same time, programming

Algorithm 1. Explosion sparks generation in FWA

```
for every firework F_i
        calculate displacement: ΔF = A_i × rand(−1,1)
        get the number of dimensions to be displaced
        for each dimension of F_i to be displaced do
                F_i^k = F_i^k + ΔF
                if F_i^k is out of bounds then
                        F_i^k = F_min^k + |F_i^k| % (F_max^k − F_min^k)
                end if
        end for
end for
```

GPUs for general-purpose tasks requires learning new interfaces provided by the manufacturer. Programming GPUs for general purpose computing may become a challenge if suitable APIs are not provided concerning the task at hand. Recently, better APIs are being made available for easy programmability (Veronese & Krohling, 2009; Wong, Wong, & Fok, 2005).

Swarm intelligence algorithms, though inherently parallel, have to be modified if they are to leverage the full power of GPUs. Hence, to use the capabilities of GPUs, the algorithm has to be refashioned significantly. In (Ding, Zheng, & Tan, 2013), the authors introduced a GPU based parallel FWA algorithm and implemented it on NVIDIA's Computing Unified Device Architecture (CUDA) platform, which is a high level general purpose programming model. The authors refashioned the conventional FWA to suit the GPU architecture and called it GPU-FWA, and in addition, introduced and used the attract-repulse mutation for the generation of Gaussian sparks instead of the one used in conventional FWA:

$$F_i^k = F_i^k + \left(F_i^k - F_{best}^k\right) \times e \tag{3}$$

Algorithm 2. Gaussian sparks generation in FWA

```
for every firework F_i
        calculate displacement: e = gaussian(1,1)
        get the number of dimensions to be displaced
        for each dimension of F_i to be displaced do
                F_i^k = F_i^k × e
                if F_i^k is out of bounds then
                        F_i^k = F_min^k + |F_i^k| % (F_max^k − F_min^k)
                end if
        end for
end for
```

Algorithm 3. Pseudo-code for Fireworks Algorithm (FWA)

1. initialize *n* fireworks
2. evaluate *n* fireworks
3. calculate explosion amplitude A_i for all fireworks
4. while stopping criteria is not met : do
    a. calculate explosion amplitude and no. of sparks for each firework using Equation (1) and (2)
    b. generate explosion sparks for each firework using Algorithm 1
    c. generate Gaussian sparks using Algorithm 2
    d. evaluate the fitness of all sparks and select the next generation of fireworks
5. end while

where F_i and F_{best} are the current and best firework respectively, and e is a random number drawn from the Gaussian distribution with mean 0 and standard deviation 1. According to Ding, Zheng, & Tan (2013), this approach balances both exploration and exploitation. This approach is quite similar to the Gaussian mutation scheme of EFWA.

To make FWA more suited to the GPU architecture, some changes were proposed by the authors. GPU-FWA was designed to make every firework generate a fixed number of sparks, *m*, differing from FWA, where the number of sparks generated by a firework is dependent on its fitness value. It is worth recalling that in FWA a better firework would generate more sparks than worse ones. The fixed number of sparks, *m*, is determined by the GPU architecture, and the authors argue that this scheme is more suitable for parallel implementation as it offsets the extra synchronization overhead. Authors refer to this overhead as the dynamic computational load for determining the number of sparks for each firework, which would require access to other fireworks and their respective sparks. In GPU-FWA, fireworks do not exchange information during every explosion. This exchange of information requires explicit synchronization and results in another computing overhead. Fireworks are allowed to explode without exchanging information for a certain number of iterations. Thus, by restricting global communications between the sparks and keeping the number of generated sparks fixed, GPU-FWA tries to avoid the related overheads and thereby increasing the speedup. GPU-FWA is summarized in Algorithm 4.

GPU-FWA was evaluated against FWA and PSO on its speed up capability and solution quality. The authors conducted the experiments on eight well known benchmark functions, utilizing multiple initial swarm sizes. The swarm sizes chosen were dependent on the GPU architecture. In the experimentation, GeForce 560 Ti GPU with 12 CUDA cores was used. Hence, the initial number of fireworks or swarm size chosen, to compare the solution quality, was 12×4=48, which was large enough to avoid compute cycle wastage. For a comparison of the speedup of GPU-FWA against FWA and PSO, swarm sizes were 48, 72, 96, and 144, respectively. The same number of function evaluations was executed for all three compared algorithms.

GPU-FWA was reported to have outperformed FWA and PSO in solution quality and obtained a speedup of 160 and 200 compared to conventional FWA and PSO, respectively. GPU-FWA also proved to be more scalable than its counterpart GPU-PSO.

Algorithm 4. GPU-FWA

1. initialize n fireworks
2. evaluate n fireworks
3. while stopping criteria is not met : do
    a. calculate explosion amplitude and no. of sparks for each firework
    b. for i =1 to n do
        for j = 1 to $\acute{L}$
        //avoids global synchronization compute overhead
        generate fixed no. of m sparks for each i
        evaluate each spark
        //avoids global synchronization compute overhead
        replace the best spark with current firework if better
        end for
    end for
    c. apply attract-repulse mutation
    d. evaluate the fitness of all sparks and select the next generation of fireworks
4. end while

$\acute{L}$ is the number of iterations for which fireworks are not allowed to exchange information globally thereby reducing synchronization overhead

2.3. MapReduce-based Nature-inspired Algorithms

Since the aim of this paper is to parallelize the EFWA algorithm using MapReduce, related work in which evolutionary computation or swarm intelligence approaches were parallelized using MapReduce are reviewed.

A Genetic Algorithm (GA) was first implemented with the MapReduce framework in (Jin, 2008) featuring a hierarchical reduction phase. This hierarchical reduction phase allows speeding up the computation by extending MapReduce. The extension is done by adding a second reduce phase and a special optimization on the merge phase. This allows the GA to be executed very efficiently.

Another MapReduce-enabled GA was introduced in (Verma, 2009). The author investigated the convergence and scalability of the implementation on the BitCounting problem. Good speedup results were achieved even on small problems investigated.

A practical application of GA modeled with MapReduce was proposed in (Huang, 2010), where the authors implemented GA for job shop scheduling problems running experiments with various population sizes and on clusters of various sizes. The authors comment on the speedup results obtained with an evaluation of different shop scheduling benchmark problems.

The MapReduce model was applied to PSO and evaluated on a radial basis function as the benchmark (McNabb, 2007). The authors describe the details of the communication and synchronization involved and confirmed that their approach scaled well for optimizing data-intensive functions. However, the authors comment that the MapReduce implementation applied

to easy benchmark functions is not appropriate since the parallelization overhead outweighs the speedup gain.

The authors proposed a MapReduce-based Ant Colony Optimization (ACO) approach in (Wu, 2012), and outlined the modeling with the MapReduce framework. The author argues that larger population sizes are necessary in order for the ACO approach to avoid local minima. The MapReduce implementation was evaluated on the Traveling Salesman Problem achieving good scalability.

The authors of this paper have experience in the parallelization of nature–inspired algorithms using the MapReduce framework. For example, in the clustering area three MapReduce-enabled algorithms were implemented, two using PSO (Aljarah and Ludwig, 2012) and (Aljarah and Ludwig, 2013), and the other using a Glowworm Swarm Optimization (GSO) approach (Al-Madi, Aljarah & Ludwig, 2014). Furthermore, an overlay network optimization approach was parallelized using MapReduce (Ludwig, 2014) showing good scalability.

3. PROPOSED APPROACH: MAPREDUCE-ENABLED ENHANCED FIREWORK ALGORITHM (MR-EFWA)

3.1. Enhanced Firework Algorithm

In (Zheng, Janecek, & Tang, 2013), the authors discuss important shortcomings of FWA and propose improvements to overcome those. They introduce their new improved algorithm as Enhanced FireWorks Algorithm (EFWA). This improved version attacks the primary inherent shortcomings of FWA, which are:

- Poor performance on shifted functions.
- High computational cost per iteration as compared to other metaheuristic optimization algorithms.

EFWA effectively introduces the following changes to FWA:

- A minimal explosion amplitude check: According to Equation 1, a better firework would have a lesser amplitude and the best firework's amplitude would almost be negligible limiting its search capability. Thus, a minimal amplitude was specified in EFWA that is reduced with increasing function evaluations. For every selected dimension k of firework i, the explosion amplitude is calculated as:

$$A_i^k(t) = \begin{cases} A_{min}^k \; if \; A_i^k < A_{min}^k \\ A_i^k \quad otherwise \end{cases} \tag{4}$$

and the minimum explosion amplitude is reduced with time linearly as:

$$A_{min}^k(t) = A_{init} - \frac{A_{init} - A_{final}}{Fev_{max}} \times Fev_{curr} \tag{5}$$

Algorithm 5. Explosion spark generation in EFWA with the new mapping operator

for every firework F_i
    calculate displacement: $\Delta F = A_i \times rand(-1,1)$
    get the number of dimensions to be displaced
    for each dimension of F_i to be displaced do
        $F_i^k = F_i^k + \Delta F$
        if F_i^k is out of bounds then
            $F_i^k = F_{min}^k + rand \times (F_{max}^k - F_{min}^k)$ //New mapping operator
        end if
    end for
end for

where Fev_{curr} and Fev_{max} are the current and maximum number of function evaluations, respectively.

- A new operator for explosion spark generation and a new mapping operator were introduced:

- A new operator for Gaussian spark generation was introduced, which is a new and relatively much less costly selection method:

This work primarily aims to leverage the inherent parallelism of EFWA as a swarm intelligence algorithm. We investigate the scalability of EFWA employing the MapReduce framework as the platform. We call our algorithm MR-EFWA. This has an advantage of simplicity over GPU-FWA as the algorithm does not have to be redesigned as per GPU architecture. More details are given in the following section.

3.2. Enhanced Firework Algorithm Implementation Using MapReduce

Google introduced the MapReduce programming paradigm (Dean & Ghemawat, 2004) that has become very popular in recent years. It is being seen as the alternative for parallel data programming compared to the Message Passing Methodology (MPI) (Snir, Otto, Huss-Lederman, Walker, & Dongarra, 1995). Besides Google's implementation of MapReduce, there are several open source implementations available such as Apache Hadoop (Apache software foundation, 2011), and Disco (Disco mapreduce framework, 2011). These implementation frameworks contain a very scalable model that can be used across many computing nodes. The concept that MapReduce employs is that it moves the processing to the data and processes data sequentially to avoid random access that requires expensive seek and disk throughput. Furthermore, the MapReduce model provides fault-tolerance and load balancing.

MapReduce technologies have been adopted by a number of groups in industry such as Facebook (Hadoop, 2011), Yahoo (Yahoo inc., 2011), etc. Researchers in academia are using

Algorithm 6. Explosion spark generation in EFWA with the new mapping operator

for every firework F_i
    calculate displacement: $e = gaussian(0,1)$
    get the number of dimensions to be displaced
    for each dimension of F_i to be displaced do
        $F_i^k = F_i^k + (F_{Best}^k - F_i^k) \times e$ where F_{Best} is the best firework location
        if F_i^k is out of bounds then
            $F_i^k = F_{min}^k + rand \times (F_{max}^k - F_{min}^k)$ //New mapping operator
        end if
    end for
end for

MapReduce for scientific computing in areas such as Bioinformatics (Gunarathne, Wu, Qiu, & Fox, 2010), and in the Geosciences (Krishnan, Baru, & Crosby, 2010).

The basic idea behind the MapReduce methodology is that the parallelization of a program is formulated as a functional abstraction using two main operations: *Map* and *Reduce*. The *Map* operation iterates over a large number of records and extracts interesting information from each record, and all values with the same key are sent to the same *Reduce* operation. The *Reduce* operation aggregates intermediate results with the same key that is generated by the *Map* operation and then generates the final results. The two operations are as follows:

Map Operation: Map(k,v) -> [(k',v')]
Reduce Operation: Reduce(k',[v']) -> [(k',v')]

Apache Hadoop [5] is a very popular open-source MapReduce implementation that supports the parallelization of applications. An implemented application can work with many thousands of computationally independent computers and petabytes of data. Apache Hadoop consists of the Hadoop Distributed File System (HDFS), which is the storage component, and MapReduce, which is the processing component. The HDFS provides high-throughput access to the data, while maintaining fault tolerance by creating multiple replicas of the target data blocks. MapReduce is designed to work with the HDFS to provide the ability to move computation to the data and not vice versa.

EFWA has been implemented with the *Map* and *Reduce* functions. The main EFWA code is run in the *Map* function, and the *Reduce* function aggregates the fitness values and emits the best value of the entire run. Algorithm 7 gives an account of the MapReduce implementation. The *Main* function first sets up the parameters necessary for executing the MapReduce job. The parameters include the number of mappers, number of reducers, input directory, and output directory. Then, the Hadoop framework starts the execution of the MapReduce job, which internally calls the *Map* and *Reduce* functions. In the *Map* function, the entire EFWA code is run and the best fitness is emitted. Depending on the *n* mappers specified, *n Map* functions emit the best fitness value of each run. In the *Reduce* function all fitness values are iterated over in order to identify the best fitness value of the entire run, which is then emitted.

Algorithm 7. MapReduce-enabled EFWA (MR-EFWA)

```
Algorithm MR-EFWA
        Function Main()
                configureEFWA()
                initSystemConfiguration()
                setUpMapperNodes ()
                setJobConfiguration()
                timeStamp1()
                runMapReduceJob()
                timeStamp2()
                writeResultsToFile()
        End function

        Function Map(key,value)
                runEntireEFWA()
                emit(key',value')
        End function

        Function Reduce(key,value)
                iterateOverAllFitnessAndSelectBest()
                emit(key',value')
        End function
End Algorithm
```

4. EXPERIMENTATION AND RESULTS

The experiments were conducted on the Rustler Hadoop cluster hosted by the Texas Advanced Computing Center (TACC)[1]. The TACC cluster consists of 66 nodes computing cluster for data intensive computing. Each node has dual eight core Ivy Bridge CPUs (Intel(R) Xeon(R) CPU E5-2650) running at 2.60GHz, with 128 GB DDR3 memory and 16 1TB SATA drives providing the backing for the HDFS file system. All nodes are connected via a 10 Gbps network. Hadoop version 0.21 is used for the MapReduce framework, and Java runtime 1.7 for the system implementation.

Four sets of experiments were performed. The first and second set investigates the effect of dimensionality using the Rosenbrock benchmark function measuring the execution time and speedup. The difference between the two sets of experiments is that in the first experiment the number of independent optimization runs were equal to the number of mappers used, whereas in the second experiment a constant number of independent optimization runs were executed on varying numbers of mappers. The third set of experiments executes 6 benchmark functions and investigates the execution time and speedup for a problem dimension of 30. The fourth set of

Table 1. Benchmark functions

	Benchmark Function	Range	Dimension	Opt. f(x)
F1	Sphere	[±100]	30	0.0
F2	Schwefel	[±100]	30	0.0
F3	Gen. Rosenbrock	[±30]	30, 60, 90	0.0
F4	Ackley	[±32]	30	0.0
F5	Gen. Griewank	[±600]	30	0.0
F6	Gen. Rastrigin	[±5.12]	30	0.0

experiments shows the execution time and speedup of a MapReduce PSO (MR-PSO) algorithm for the six benchmark functions using 30 dimensions.

The overall settings of EFWA that are the same throughout the different sets of experiments are as follows:

- Number of locations = 100
- Number of maximum sparks = 60
- Number of minimum sparks = 2
- Number of maximum amplitude = 40
- Number of Gaussian sparks = 5
- Number of function evaluations = 300,000

The benchmark functions that are used are shown in Table 1.

The first experiment investigates the execution times of the Rosenbrock function for three different dimensions (30, 60, and 90) using different numbers of mappers. The number of reducers is set to 1 for all experiments. Basically, in this experiment one EFWA optimization run is executed in one mapper. What can be seen from by the results listed in Table 2 is that we observe a constant execution time for each of the three dimensions with dim = 90 having, as expected, the largest execution time. Running the optimization of the benchmark function on 450 and 500 nodes shows slightly shorter execution times. It seems that the utilization of the Hadoop framework improves with larger number of nodes used.

The results of the second set of experiments are shown in Figures 1-3 using Rosenbrock for dimensionality of 30, 60, and 90, respectively. For these experiments the number of EFWA optimization runs are 500, which are spread over the different number of mappers used. Figure 1 shows the execution time exponentially decreasing with increasing number of mappers used, and the speedup increases linearly. The speedup is calculated with the execution time of using 2 mappers and 1 reducer as the base. What we can see is a linear trend achieving a speedup of around 58 for 500 mappers used.

Figure 2 shows the execution time in seconds and the speedup for Rosenbrock using 60 dimensions. We observe similar time and speedup graphs, however, with the higher dimensionality we see larger execution times compared to dim=30 and slightly lesser speedup (achieving a speedup of 50 when 500 mappers are used).

Figure 3 shows the results for the optimization of Rosenbrock for dimension 90. Again, the execution time is longer and the speedup is smaller compared to 60 and 30 dimensions. A speedup of 45 is achieved when 500 mappers are used.

Table 2. Time results in seconds for Rosenbrock on dim = 30, 60, and 90 using different number of mappers (M) and equal number of optimization runs

M	Time in seconds		
	Dim = 30	Dim = 60	Dim = 90
50	94	151	218
100	93	153	215
150	95	154	218
200	96	155	212
250	98	159	210
300	97	168	208
350	97	159	222
400	92	155	209
450	86	125	211
500	88	123	202

The third set of experiments use the 6 benchmark functions outlined in Table 1. In these experiments, 300 independent EFWA optimization runs were executed on 100, 200, and 300 mappers, respectively. Figure 4 shows the results in terms of execution time in seconds (on the left) and speedup (on the right). Comparing the different benchmark functions we observe that F1 executes the fastest followed by F2, F3, and F5. The largest execution times are measured on F4 and F6. The speedup results show that F4 scales best achieving a speedup of 76, whereas F1 and F2 scale worst.

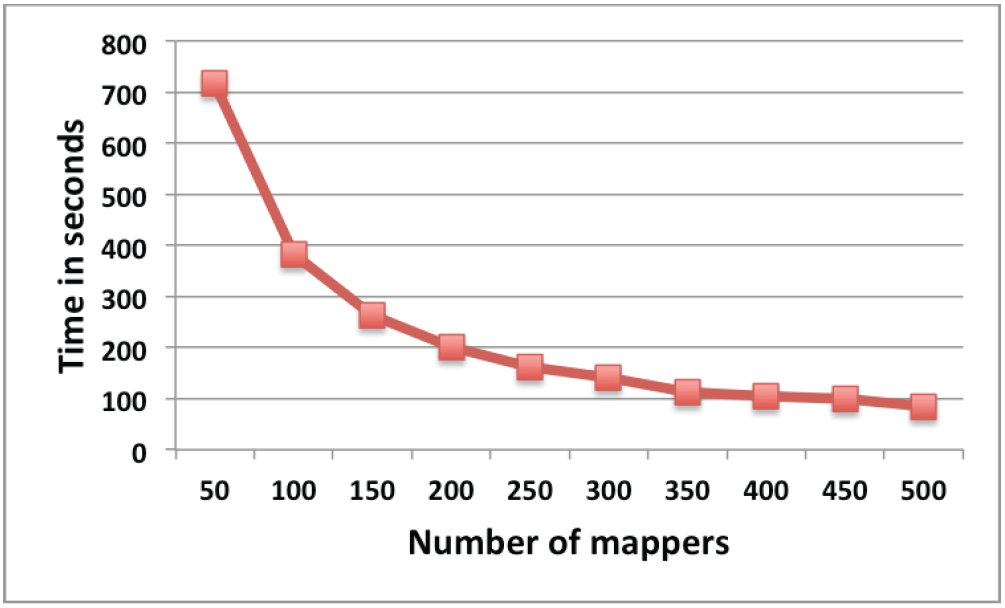
In (Ding, Zheng, & Tan, 2013), the authors compared their GPU-EFWA with a GPU-enabled PSO algorithm, and therefore, this paper has also implemented a MapReduce-based PSO (MR-PSO) algorithm in order to compare MR-EFWA with. Thus, the fourth set of experiments use the 6 benchmark functions outlined in Table 1 evaluating the MR-PSO algorithm. Similar to the third set of experiments, 300 independent PSO optimization runs were executed on 100, 200, and 300 mappers, respectively.

Figure 5 shows the execution time in seconds (on the left) and the speedup results (on the right). Comparing the different benchmark functions we observe that F1 executes the fastest followed by F6, F3, and F4. The largest execution times are measured on F5. F3 shows the best scalability results achieving a speedup of 63, whereas F1 and F2 scale worst.

Comparing MR-PSO (Figure 5) with MR-EFWA (Figure 4), both similarities but also differences can be observed. Overall, even though the execution times of F1 and F2 are comparable, however those for F3-F6 are quite different. This implies that the optimization for both, EFWA and PSO, operate differently and the parallelization is used differently.

Furthermore, MR-EFWA achieves better speedup results (best: 76) compared to MR-PSO (best: 63). This implies that the utilization of the MapReduce framework is better for the MR-EFWA algorithm. Thus, this concurs with the findings in (Ding, Zheng, & Tan, 2013) where the same result was found.

Figure 1. Time and Speedup versus number of mappers for dim = 30



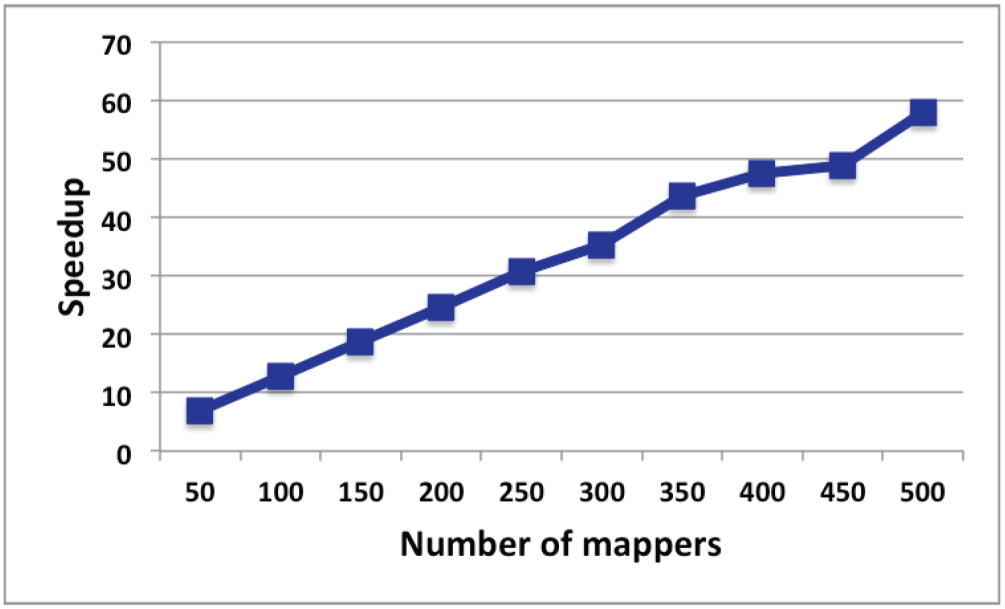
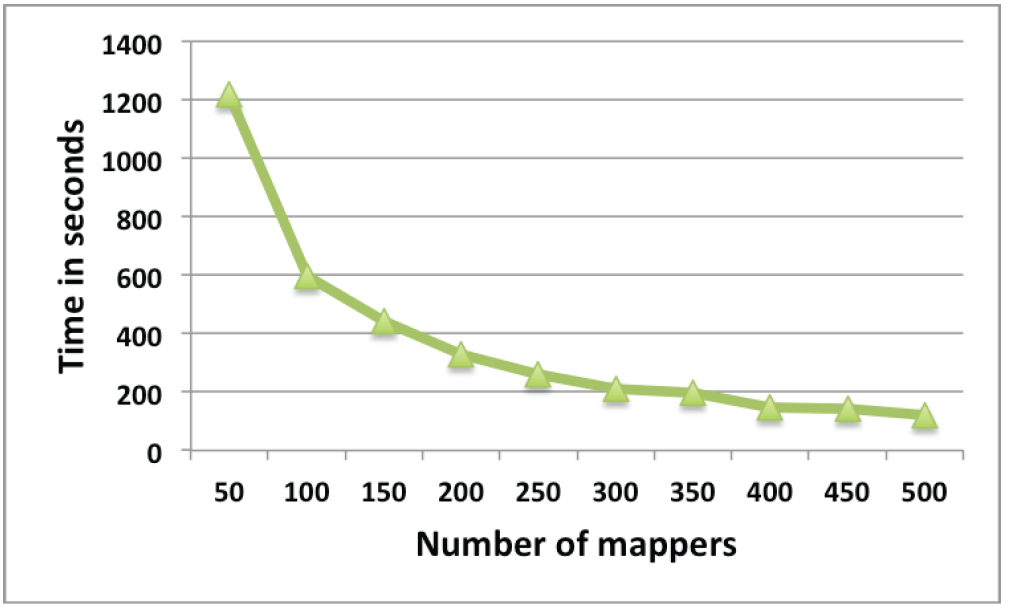


Figure 2. Time and Speedup versus number of mappers for dim = 60



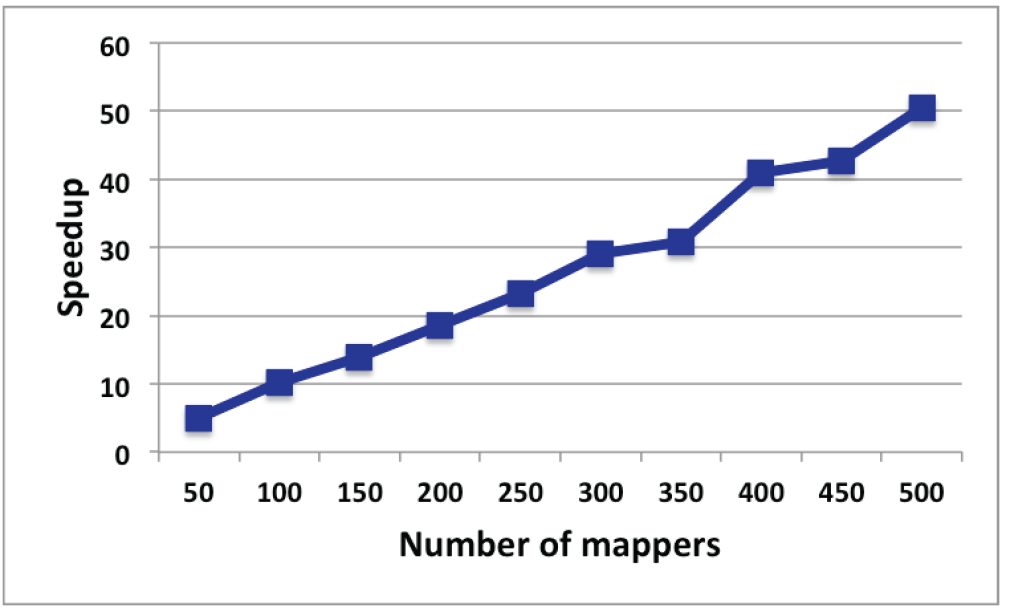
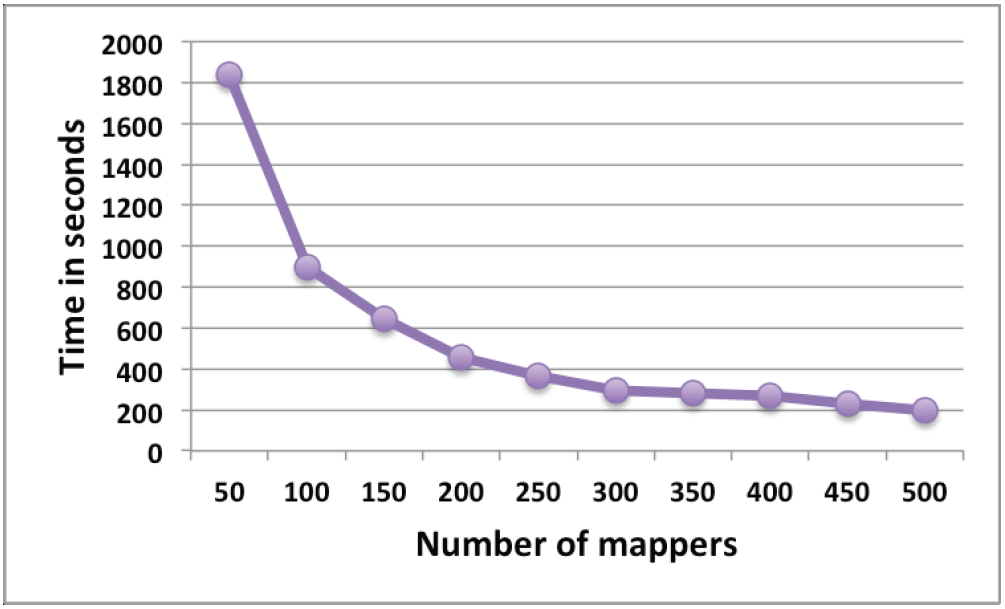


Figure 3. Time and speedup versus number of mappers for dim = 90



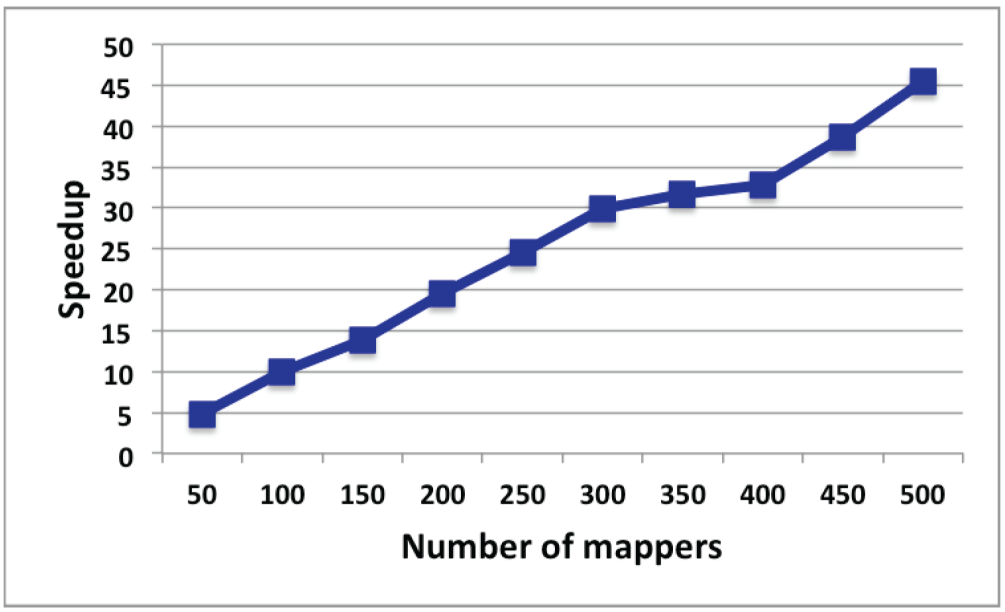
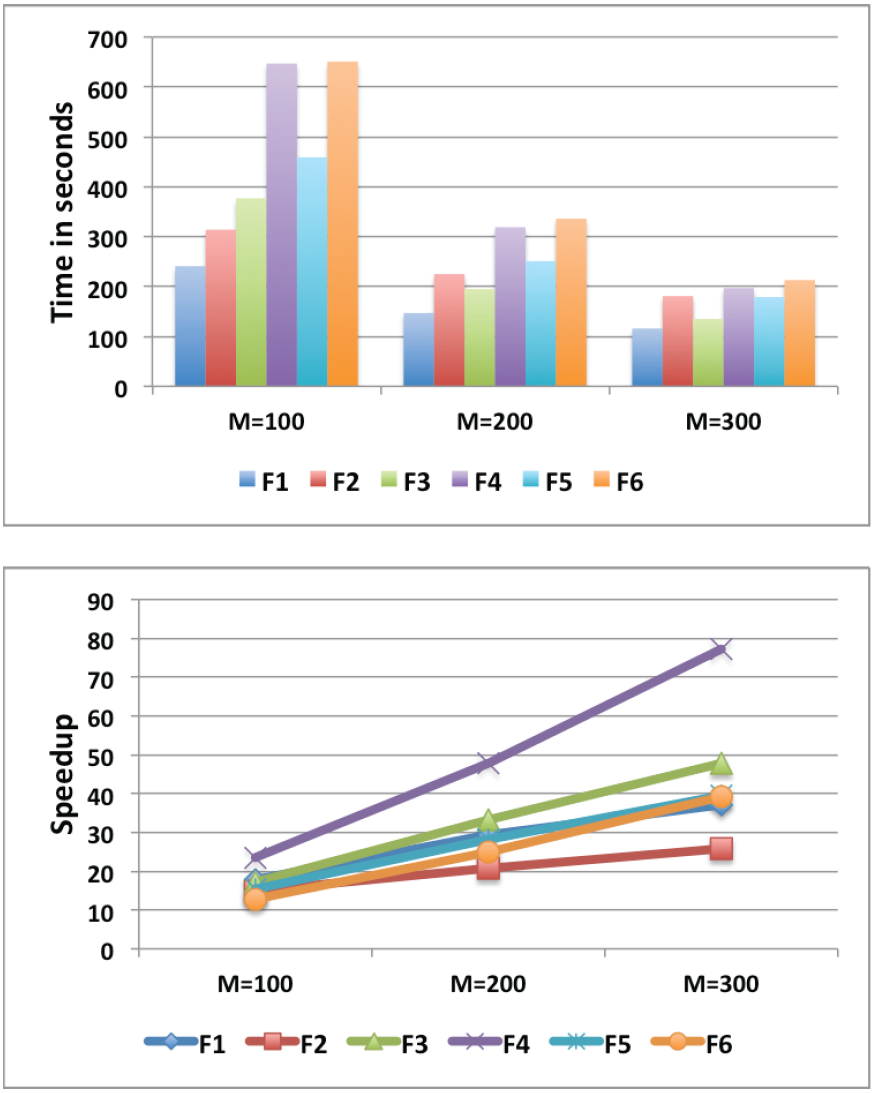


Figure 4. Time and speedup for six functions for different numbers of mappers

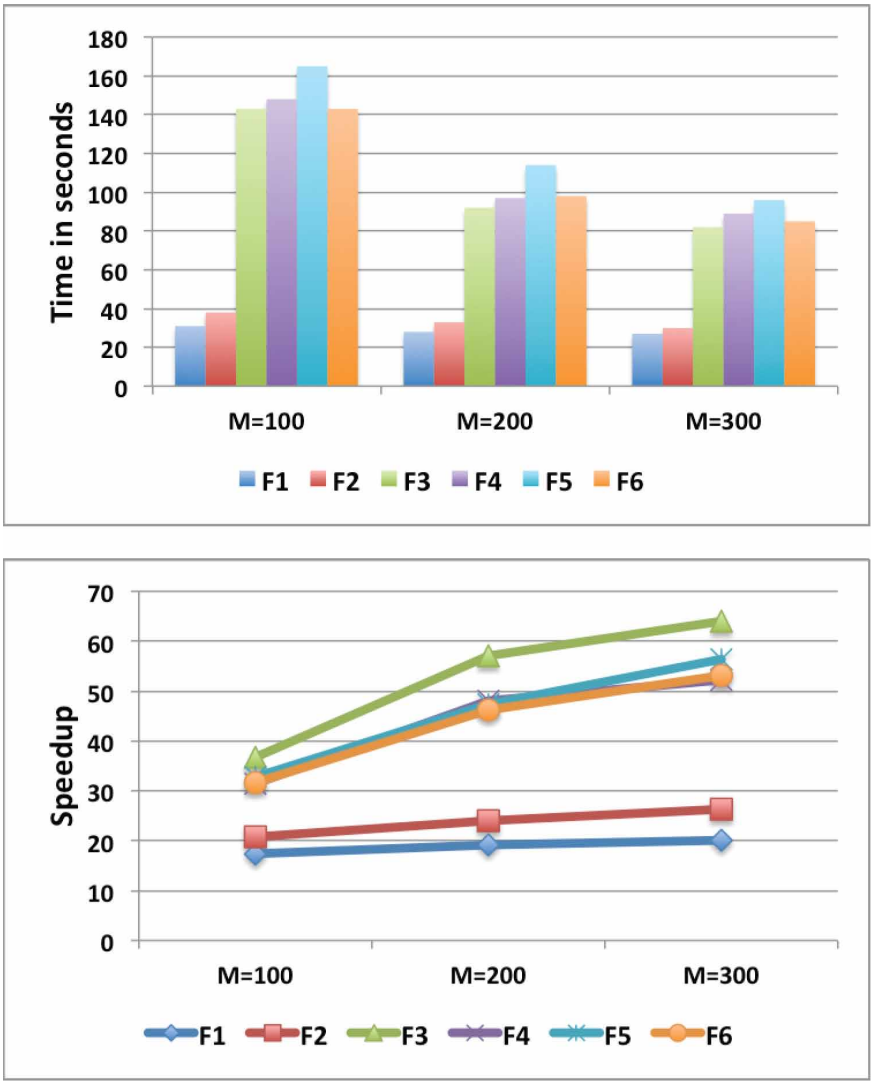


5. CONCLUSION

This paper presents a scalability analysis of the Enhanced Fireworks Algorithm (EFWA), an improved version of the original Fireworks algorithm by introducing a parallel implementation of EFWA. We employed the MapReduce framework to investigate the scalability of our implemented MR-EFWA for different benchmark functions with different dimensions.

From the different experiments conducted we can see that our MR-EFWA implementation scales very well achieving good speedup values. The highest speedup that was achieved was 77 for the Rosenbrock function when 300 independent EFWA optimization runs were executed. When 500 independent runs were used, a speedup of 58 was achieved as on the Rosenbrock

Figure 5. Time and speedup for six functions for different numbers of mappers



function. Furthermore, given the difficulty of the optimization functions used, the speedup values vary. The more difficult the optimization function is, the better speedup values can be achieved. In addition, a comparison with a MapReduce PSO (MR-PSO) algorithm demonstrated that the MapReduce framework is better utilized by the MR-EFWA algorithm. This finding concurs with the finding of the GPU-based implementation versus the PSO-based implementation in (Ding, Zheng, Tan, 2013).

Even though FWA has been parallelized before using a GPU architecture achieving speedup values of 250 on some benchmark functions, however, first of all the basic FWA was used (EFWA has shown an improved execution time compared to FWA), and secondly, a different architecture (GPU) was used. These two facts make a direct comparison impossible. However, the benefit of using the MapReduce concept is that commodity hardware can be used to execute

MapReduce-enabled applications, which makes the execution more affordable. Secondly, fault-tolerance and load-balancing is already implemented within the framework, whereas for the GPU implementation this needs to be explicitly programmed. Thirdly, the implementation using the MapReduce concept is easy and straightforward.

Future work will investigate the effect of other parameters settings such as larger numbers of sparks (normal and Gaussian) on the execution time. Moreover, in this paper we kept the number of function evaluations constant, however, the effect of different numbers of function evaluations should be investigated.

ACKNOWLEDGMENT

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

REFERENCES

Al-Madi, N., Aljarah, I., & Ludwig, S. A. (2014). Parallel Glowworm Swarm Optimization Clustering Algorithm based on MapReduce: In *Proceedings of the IEEE Symposium Series on Computational Intelligence (SSCI)*. Orlando, USA. doi:10.1109/SIS.2014.7011794

Aljarah, I., & Ludwig, S. A. (2012). Parallel Particle Swarm Optimization Clustering Algorithm based on MapReduce Methodology. In *Proceedings of the Fourth World Congress on the Nature and Biologically Inspired Computing (NaBIC)*. Mexico City, Mexico. doi:10.1109/NaBIC.2012.6402247

Aljarah, I., & Ludwig, S. A. (2013). MapReduce Intrusion Detection System based on a Particle Swarm Optimization Clustering Algorithm. In *Proceedings of the IEEE Congress on Evolutionary Computation*. Cancun, Mexico. doi:10.1109/CEC.2013.6557670

Apache software foundation. (2011). *Hadoop mapreduce*. Retrieved July 10, 2014, from http://hadoop.apache.org/mapreduce

Beni, G., & Wang, J. (1989). Swarm Intelligence in Cellular Robotic Systems. *In Proceedings of the NATO Advanced Workshop on Robots and Biological Systems*. Tuscany, Italy.

Dean, J., & Ghemawat, S. (2004). *Mapreduce: Simplified data processing on large clusters*. Google Inc.

Ding, K., Zheng, S., & Tan, Y. (2013). A gpu-based parallel fireworks algorithm for optimization. In *Proceedings of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference, ser. GECCO*. New York, NY, USA. doi:10.1145/2463372.2463377

Disco mapreduce framework. (2011). Retrieved Oct 10, 2014 from http://discoproject.org

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics*, *26*(1), 29–41. doi:10.1109/3477.484436 PMID:18263004

Eberhart, R., & Kennedy, J. (1995). Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, 4, 1942-1948.

Eiben, A. E., & Smith, J. E. (2003). Introduction to Evolutionary Computing, Springer, Natural Computing Series, 1st edition. doi:10.1007/978-3-662-05094-1

Gandomi, A., Yang, X. S., & Alavi, A. (2013). Cuckoo search algorithm: A metaheuristic approach to solve structural optimization problems. *Engineering with Computers*, *29*(1), 17–35. doi:10.1007/s00366-011-0241-y

Gunarathne, T., Wu, T., Qiu, J., & Fox, G. (2010). Cloud computing paradigms for pleasingly parallel biomedical applications. *In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 460–469. doi:10.1145/1851476.1851544

Hadoop (2011). *Facebook engg, note*. Retrieved July 20, 2014 from http://www.facebook.com/note.php?noteid=16121578919

Huang, D. W., & Lin, J. (2010). Scaling populations of a genetic algorithm for job shop scheduling problems using mapreduce. In *Proceedings of the IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*. Indianapolis, Indiana, USA. doi:10.1109/CloudCom.2010.18

Janecek, A., & Tan, Y. (2011a). Iterative improvement of the multiplicative update NMF algorithm using nature-inspired optimization. In *Proceedings of the IEEE Seventh International Conference on Natural Computation (ICNC)*. Shanghai, China. doi:10.1109/ICNC.2011.6022356

Janecek, A., & Tan, Y. (2011b). Swarm intelligence for non-negative matrix factorization. [IJSIR]. *International Journal of Swarm Intelligence Research*, *2*(4), 12–34. doi:10.4018/jsir.2011100102

Janecek, A., & Tan, Y. (2011c). Using population based algorithms for initializing nonnegative matrix factorization. In *Proceedings of the second international conference on Advances in swarm intelligence*. Springer-Verlag. doi:10.1007/978-3-642-21524-7_37

Jin, C., Vecchiola, C., & Buyya, R. (2008). MRPGA: An extension of mapreduce for parallelizing genetic algorithms. In *Proceedings of the Fourth IEEE International Conference on eScience*. Washington, DC, USA. doi:10.1109/eScience.2008.78

Krishnan, S., Baru, C., & Crosby, C. (2010). Evaluation of mapreduce for gridding lidar data. In *Proceedings of the CLOUDCOM '10*. Washington, DC, USA. doi:10.1109/CloudCom.2010.34

Ludwig, S. A. (2014). MapReduce-based Optimization of Overlay Networks using Particle Swarm Optimization. In *Proceedings of Genetic and Evolutionary Computation Conference (ACM GECCO)*. Vancouver, BC, Canada. doi:10.1145/2576768.2598269

McNabb, A., Monson, C., & Seppi, K. (2007). Parallel pso using mapreduce. In *Proceedings of the IEEE Congress on Evolutionary Computation*. Singapore.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. E., & Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, *26*(1), 80–113. doi:10.1111/j.1467-8659.2007.01012.x

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., & Dongarra, J. (1995). *MPI: The Complete Reference*. MA, USA: MIT Press Cambridge.

Tan, Y., & Xiao, Z. (2007). Clonal particle swarm optimization and its applications. *IEEE Congress on Evolutionary Computation*. Singapore. doi:10.1109/CEC.2007.4424758

Tan, Y., & Zhu, Y. (2010). Lecture Notes in Computer Science: Vol. 6145. *Fireworks algorithm for optimization. Advances in Swarm Intelligence* (pp. 355–364). Springer Berlin Heidelberg. doi:10.1007/978-3-642-13495-1_44

Verma, A., Llora, X., Goldberg, D., & Campbell, R. (2009). Scaling genetic algorithms using mapreduce. Paper presented at *Ninth International Conference on Intelligent Systems Design and Applications*. Pisa, Italy. doi:10.1109/ISDA.2009.181

Veronese, L. de P., & Krohling, R. A. (2009). Swarm's flight: Accelerating the particles using c-cuda. *IEEE Congress on Evolutionary Computation, CEC '09*. Trondheim, Norway. doi:10.1109/CEC.2009.4983358

Wong, M. L., Wong, T. T., & Fok, K. L. (2005). Parallel evolutionary algorithms on graphics processing unit. In Proceedings of the *IEEE Congress on Evolutionary Computation.* Edinburgh, Scotland.

Wu, B., Wu, G., & Yang, M. (2012). A mapreduce based ant colony optimization approach to combinatorial optimization problems. In Proceedings of the *Eighth International Conference on Natural Computation* (ICNC). Chongqing, Sichuan, China. doi:10.1109/ICNC.2012.6234645

Yahoo inc. (2011). *Hadoop at Yahoo!* Retrieved Oct 12, 2014 from http://developer.yahoo.com/hadoop

Yang, X. S. (2005). Engineering optimizations via nature-inspired virtual bee algorithms. *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*. *Lecture Notes in Computer Science*, *3562*, 317–323.

Yang, X. S. (2009). Firefly algorithms for multimodal optimization. *Stochastic Algorithms: Foundations and Applications*. *Lecture Notes in Computer Science*, *5792*, 169–178.

Yang, X. S., & Gandomi, A. (2012). Bat algorithm: A novel approach for global engineering optimization. *Engineering Computation*, *29*(5), 464–483. doi:10.1108/02644401211235834

Zheng, S., Janecek, A., & Tan, Y. (2013). Enhanced Fireworks algorithm. *IEEE Congress on Evolutionary Computation*. Cancun, Mexico.

Zheng, Y., Xu, X., Ling, H., & Chen, S.-Y. (2012). A hybrid fireworks optimization method with differential evolution operators. *Neurocomputing*, *148*, 75–82. doi:10.1016/j.neucom.2012.08.075

Zheng, Y. J., Song, Q., & Chen, S. Y. (2013). Multiobjective fireworks optimization for variable-rate fertilization in oil crop production. *Applied Soft Computing*, *13*(11), 4253–4263. doi:10.1016/j.asoc.2013.07.004

ENDNOTES

[1] https://www.tacc.utexas.edu/systems/rustler

Simone A. Ludwig is an associate professor of Computer Science at North Dakota State University, USA. She received her PhD degree and MSc degree with distinction from Brunel University, UK, in 2004 and 2000, respectively. Before starting her academic career she worked several years in the software industry. Her research interests include swarm intelligence, evolutionary computation, and fuzzy reasoning.

Deepak Dawar received the BTech degree in Electronics and Communication Engineering from Kurukshetra University, India, in 2007, and an MS degree from North Dakota State University in 2013. He is currently a first year Phd student and a teaching assistant at North Dakota State University. His current research interests include evolutionary computing, swarm intelligence, and pattern recognition.