Parallel Implementation of the Bat Algorithm on a Single Computer

Aleš ČEP, Damijan NOVAK, Domen VERBER
Laboratory for Real-Time Systems
University of Maribor
Maribor, Slovenia
{ales.cep, damijan.novak, domen.verber}@um.si

ABSTRACT
The Bat algorithm is an optimization algorithm based on the echolocation behavior of bats. In this paper, we propose different parallel implementation of the algorithm on a single computer. The first implementation was accelerated on multicore processors using OpenMP, the second one used the graphical processors and was implemented in CUDA. The experiment was conducted on standard benchmark functions and results show that the CUDA implementation achieved significant speedup in comparison to both single threaded and OpenMP implementation. CUDA implementation also shows that execution time was nearly the same regardless of the number of dimensions.

Keywords: parallel algorithm; Bat algorithm; speedup; OpenMP, CUDA.

1. INTRODUCTION
In this paper, we discuss different parallel implementation of the Bat optimization algorithm on a single computer. Our research is based on the study of contemporary programming tools for parallel programming. At first the Bat algorithm was chosen as a benchmark. However, after the literature review, we found that there is no adequate parallel implementation of this algorithm.

Frequently, in engineering and the other fields, optimization problems arise. In such problems, we need to optimize (minimize or maximize) some objective function by choosing the proper input parameters, with regard to some constraints. Formally, we can define a scalar objective function as:

\[ f(x) : \mathbb{R}^D \rightarrow \mathbb{R}; \quad x_i \in [x_{\text{min}}, x_{\text{max}}] \]  

(1)

The x is a vector variable that represent the input parameters of the observed system. The D represents the dimension of the input domain. There can be more complex boundary conditions, expressing more sophisticated interactions between the input parameters. The objective function can have a global optimum and several local ones. One of the major problems we usually need to consider is how not to be stuck in the local optimum and to find the global one instead.

Several strategies can be used to find an optimum of such function. Firstly, the solution can be found analytically. Of course, no additional programming is required in such case. However, in most practical cases, the objective function is too complex for this. Secondly, the solution can be found with the numerical methods. Several numerical evaluation algorithms exist for finding the optimum of some function. However, this approach is usually suitable only for some classes of functions (e.g. the objective function needs to be expressed as a set of differential equations). The third method is using some sort of searching algorithm that explores the space of input variables to find the global optimum. The algorithm starts with some (arbitrary) initial vector for x and tries to change it iteratively until the optimum is attained. In each step the algorithm must decide in which direction and how far the current x must be changed. It must do this efficiently and it must avoid the local optimums. To increase the chance of finding the global optimum, several starting points can be chosen and evaluated from the start. Each starting point will produce a different searching path and (hopefully) at least one of them would lead to the proper result. In this way, we can also reduce the number of iterations to find the solution. Some of the initial searching points could be nearer to the solution than others and converge to the optimum more quickly.

Parallel computing is a technique that utilizes abundant computing resources to solve the problems more quickly. This can be done in two ways. We can execute different jobs simultaneously (e.g. with the multicore processors) or we can apply the same job on different pieces of data at the same time (e.g. with GPU accelerators). We can define a speedup as a measure of efficiency of the parallelization. This is the ratio between the times required for execution of the same job with a single processing element and with multiple ones. Not all problems are suitable for parallelization. To utilize it efficiently, the problem should be divisible into equivalent parts with no inter-dependencies. Usually, the implementation of parallelization also introduces some additional (but constant) overheads, because the parallelization can be efficient only if there are enough parts to start with.

Optimization algorithms with multiple searching paths are ideal for the parallelization. Each starting path can be evaluated independently from each other. If there are enough computational resources available, all of the search paths can be assessed simultaneously. Furthermore, we can often parallelize the evaluation of a single search path itself by observing the input dimensions for each search space concurrently. The data from the different dimensions must be consolidated only on some spots in the searching algorithm.

The article is organized as follows. Firstly, a brief overview is presented of the work related with the paper. Next, we describe our implementation of the bat optimization algorithm. The tools used in the experiments are described in the fourth Chapter. Next, the details of the experiment are presented and the results are elaborated.

2. RELATED WORK

There are several programming frameworks that can be used effectively for parallel programming and high performance
computing. Here, we are focused on the solutions that can be run on a single machine as a contrast to the solutions with clusters of computers. The most mature framework is CUDA (Compute Unified Device Architecture) [1]. It utilizes the computing resources of GPU accelerators. The application is executed with the ensembles of 32 threads, each executing the same code in parallel. CUDA has a rich programming community, diverse set of programming libraries, etc. [2] However, for efficient computing, CUDA requires a deeper knowledge of computing hardware and low-level programming. It is also a proprietary solution, bound to the specific vendor of GPUs. Similar to CUDA is the OpenCL (Open Computing Language) [3]. It used as a similar executing model. However, the OpenCL is an open standard, maintained with an international consortium [4]. It is not bound to a specific vendor.

Both CUDA and OpenCL requires the programmer to rewrite the original code into a form that can be executed on parallel computing resources. Another approach is to keep the existing code and let the compiler do all the necessary transformation. This approach is used with the OpenMP (Open Multi-Processing) programming framework [5]. OpenMP utilizes multi core processing resources and multithreading of the main CPU. The original code is augmented with the special comments (pragmas) that are used by the compiler to parallelize the execution.

The subject of optimization algorithms is vast [6]. Some problems are tough to optimize. To cope with that, different metaheuristic algorithms were developed in the past. A group of such methods was inspired by nature and especially by the behavior of animals. One of the first such algorithms was the Ant Colony optimization algorithm, where different problems can be solved by imitating the foraging behavior of the ants [7]. Nowadays, we have a “zoo” of such optimization techniques. One of the animals which can be imitated is a bat. Bats use an echolocation to find and hunt their prey, which can be observed as a method for finding the optimum in the search space of the optimization problem. The bat metaheuristic algorithm was first introduced in [8]. However, the original article did not describe all the details of the algorithm and allows for different interpretations. Nowadays, the bat optimization algorithm is used in solving different kinds of problems. In most cases, it is combined with other optimization techniques.

3. BAT ALGORITHM

The Bat algorithm was proposed by Xin-She Yang in 2010 [8]. Bats use echolocation to navigate through space and search for food. They emit a loud sound pulse and listen for the echo that bounces back. The time that is needed for the sound pulse to travel from bat to obstacle or prey and back is used to determine distance. Normally, they emit a 10 to 20 sound burst. This number can increase up to 200 pulses when they are closing in on their prey. The properties of sound bursts change in correlation with prey size and hunting strategies. The wavelength can be in the range from 2mm to 14mm and frequencies from 25 kHz to 150 kHz.

For simplicity, Yang [8] defined rules for moving bats: Bats fly randomly with velocity \(v_i\) at position \(x_i\) with fixed wavelength, varying frequency \(Q_i\) and loudness \(A_i\) to search for prey. They can adjust the frequency of their emitted pulses and also the rate of pulse emission \(r \in [0, 1]\) depending on their proximity to the prey.

The pseudo code of our implementation of the algorithm is presented in Algorithm 1.

### Algorithm 1: The Bat Algorithm

1. **Objective function** \(f(x_i), x = (x_1, ..., x_D)\)
2. Initialize the bat population \(x_i\) and \(v_i\) for \(i = 1 \ldots N\)
3. Define pulse frequency \(Q_i \in [Q_{\text{min}}, Q_{\text{max}}]\)
4. Initialize pulse rates \(r_i\) and the loudness \(A_i\)
5. **while** (\(t < \text{Max number of iterations}\))
6. Generate new solutions by adjusting frequencies
7. and updating velocities and locations \(
   \begin{align*}
   v_i(t+1) &= v_i(t) + A_i(t) \times Q_i(t) \\
   x_i(t+1) &= x_i(t) + v_i(t+1)
   \end{align*}
   \)
8. **if** (\(r \text{and} f(x_i) < f(x_j)\))
9. **end if**
10. **end if**
11. **if** (\(\text{rand}(0,1) > r_i\))
12. **Accept new solution**
13. Update \(r_i\) and \(A_i\)
14. **end if**
15. **end while**
16. **Find current best** \(x^*_i\)
17. Increase \(t\) by 1
18. **end while**
19. Post process results and visualization

The first step (lines 2–4) of this algorithm is initialization, where the bat population and frequency vector are filled with random values.

The next step (lines 6–7) generates a new population/solution by moving bats and adopting frequencies according to following equations:

\[
Q_i = Q_{\text{min}} + (Q_{\text{max}} - Q_{\text{min}}) \times \text{rand}(0,1) \tag{2}
\]

\[
v_i(t+1) = (x_i - x_f) \times Q_i \tag{3}
\]

\[
x_i(t+1) = x_i(t) + v_i(t+1) \tag{4}
\]

where \(\text{rand}(0,1)\) returns a random number of a uniform distribution between [0, 1]. In Eq. (3) \(x_i\) represents the current best solution.

The third step (line 9) is a local search that moves the current solution closer to the best solution according to equation:

\[
x_i(t+1) = x_i(t) + \epsilon \times A_i \tag{5}
\]

where \(x^*_i\) is the current best solution, \(\epsilon \in [-1, 1]\) is a random number and \(A_i\) is the average loudness of all bats.

The fourth step (line 11) is flying randomly where the current solution is changed according to the following equation:

\[
x_i(t+1) = x_i(t+1) + \theta \tag{6}
\]

where \(\theta \in [-0.5, 0.5]\).

The last step (lines 12–15) is accepting the current solution if it is better than the current best. When a bat finds its prey
loudness $A_i$, decreases and pulse rate $r_i$ increases according to equations:

$$A_i^{(t+1)} = \alpha * A_i^t,$$

$$r_i^{(t+1)} = r_i^0 * [1 - \exp(-\gamma * t)],$$

where $\alpha$ and $\gamma$ are constants that have been set to $\alpha=0.99$ and $\gamma=0.9$. Eventually, the value of $A_i^t$ will convert to 0 and $r_i^t$ will convert to the value of $r_i^0$.

4. PARALLEL ENVIRONMENTS

We implemented the Bat algorithm on two different parallel architectures:
- OpenMP
- CUDA

OpenMP
OpenMP [5] is an application program interface for shared-memory parallel programming. It defines compiler directives and callable runtime library routines. It supports Fortran and C/C++ and can be used on most architectures and operating systems. OpenMP is implemented with threads and is based on the fork-join model of parallel execution: Master thread spawns slave thread(s). The task is then divided among all threads. They run concurrently and have access to shared memory. OpenMP supports synchronizations mechanisms like critical region (only one thread at a time can enter), atomic operations (mutual exclusion, but only for updating memory location), barriers (each thread waits until all threads arrive) and locks. Most constructs in OpenMP are compiler directives defined in C/C++ as #pragma.

CUDA
CUDA [1] is a parallel computing architecture and programming model developed by NVIDIA. It allows software developers to access the computing power of NVIDIA graphics processing units. It is designed to work with programming languages such as C, C++ and Fortran. CUDA has low a learning curve for programmers familiar with those languages. CUDA extends C program language with a function library's srand() and rand() functions for generating random numbers.

OpenMP solution
In OpenMP implementation we used similar code as in single threaded solution. OpenMP parallelism was used for bat initialization and bat operations inside one iteration (lines 6-16 in Algorithm 1). Average loudness (used in (5)) was calculated with an OpenMP reduction clause.

CUDA solution
Since the Bat algorithm is of an evolutionary nature, each algorithms iteration is based on the previous one, so we could not parallelize the most outer loop (line 5 in Algorithm 1). Because of that, we used CUDA kernels to parallelize most of the algorithm inside one iteration. Kernels for bat initialization and movement were called with such a number of CUDA blocks and threads, that all the loops were eliminated. The exception is a kernel for the reduction calculation (Algorithm 2) which was implemented with loop. The purpose of that kernel was to find the best bat inside the current solution.

```
__device__ void FindBestSolutionAndCopyResult () {
    int id = threadIdx.x;
    int first = id << 1;
    int second = first + 1;

    //prepare indexes for reduction
    arrayOfIndexes[first] = first;
    arrayOfIndexes[second] = second;

    //copy elements for average calculaction
    averageAArray[first] = A[first];
    averageAArray[second] = A[second];
    int t = NO_BATS;
    int k = 1;

    // k - how many times loop needs to execute
    while (t >>= 1) k <<= 1;

    //reduction is made by changing indexes in array
    while (first < NO_BATS && k > 0) {
        if (second >= NO_BATS) //when even number of bats
            arrayOfIndexes[first] = arrayOfIndexes[first];
        else //best solution gets stored
            if (fitness[arrayOfIndexes[first]] < fitness[arrayOfIndexes[second]])
                arrayOfIndexes[first] = arrayOfIndexes[first];
            else
                arrayOfIndexes[first] = arrayOfIndexes[second];
        first <<= 1;
        second <<= 1;
        k >>= 1;
    }

    if (id == 0) //best solution is stored at index 0
        bestIndeks = arrayOfIndexes[first];
    fbest = fitness[bestIndeks];
    averLoud = averageAArray[0] / (double)NO_BATS;
}
```

Algorithm 2: CUDA implementation of reduction using bit operations for index calculation.

Usually, the most time consuming part of a CUDA program is data movement from CPU to GPU and back. We decided to
store all data in GPU memory, and thus shortening execution time. We used the XORWOW random number generation algorithm with uniform distribution that is part of the curand library.

```
double best;
//divisible by 32
int threadNo = (int)ceil(NO_BATS/2.0);
for (int it = 1; it <= 100; it++) {
clock_t cstart = clock ();
unsigned int innerLoop = 1;
Init <<< NO_BATS, D >>>(it, BenchFunction);
FindBestSolutionAndCopyResult <<<1, threadNo >>> ();
while (innerLoop++ <= MAXNUMBERofITERATIONS) {
CreateAndMoveBats <<< NO_BATS, D >>>
FindBestSolutionAndCopyResult <<<1, threadNo >>> ();
}
clock_t cend = clock ();
cudaMemcpyFromSymbol(&best, fbest, sizeof(double));
}
clock_t cilj = cend - cstart;
```

Algorithm 3: Basic structure of implementation on CUDA.

### 6. EXPERIMENTS AND RESULTS

**Benchmark functions**
Benchmark functions are standard optimization functions taken from [10].

**Easom function:** This function is very specific because it has small area around the global minimum in correlation to the search space. It’s a problem with two variables and it is defined as:

\[
    f_1(x) = -\cos(x_1)\cos(x_2)\exp(-\cos(x_1) - \cos(x_2))
\]

where \((x_1, x_2) \in [-100.00, 100.00]\). Function has the global minimum at 0.

**Eggerate function:** This function is a global optimization problem with several local minima. In our test case it is presented as a two variables problem, defined as:

\[
    f_2(x) = x^2 + y^2 + 25 \* (\sin^2 x + \sin^2 y)
\]

where \((x, y) \in [-2\pi, 2\pi]\). This function has global minimum 0 in point \((0, 0)\).

**Sphere function:** This function is continuous and convex. A general definition is:

\[
    f_3(x) = \sum_{i=1}^{D} x_i^2
\]

where \(x \in [-15.00, 15.00]\) and the value of global minimum is 0.

**Rosenbrock’s function:** This function is defined as:

\[
    f_4(x) = \sum_{i=1}^{D-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2
\]

where \(x \in [-15.00, 15.00]\). Rosenbrock’s function has a global minimum 0 at location \(x = (1, 1, \ldots, 1)\).

**Ackley’s function:** Ackley’s function has numerous local minima that can pose problems for many algorithms. It is defined as:

\[
    f_5(x) = -20 \* \exp\left(-0.2 \sqrt{\frac{1}{D} \sum_{i=1}^{D} x_i^2}\right) - \exp\left(\frac{1}{D} \sum_{i=1}^{D} \cos(2\pi x_i)\right) + 20 + \exp(1),
\]

where \(x \in [-32.00, 32.00]\). The global minimum is 0.

**Griewank’s function:** Griewank’s function has many widespread local minima distributed regularly. It’s defined as:

\[
    f_6(x) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1
\]

where \(x \in [-600.00, 600.00]\). Global minimum is 0.

**Michalewicz’s function:** It is a multimodal test function with \(D!\) local minima. Parameter \(m\) defines steepness of the valleys and ridges.

\[
    f_7(x) = -\sum_{i=1}^{D} \sin(x_i)(\sin\left(\frac{ix_i^2}{\pi}\right)^{2m
\]

In our case \(m\) was set to 10 and \(x \in [0, \pi]\).

**Experiment**
The experiment was executed on a 64-bit Windows 10 PC with 16GB of RAM and Intel Core i7-4790 CPU @ 3.60GHz that has 8 logic processors. The graphic card in the computer was an NVIDIA GeForce GT 730, which has 2.1 CUDA capability and 2048MB of global memory. We implemented our solutions in C/C++ programming language and CUDA extensions. All programs were compiled with optimization that maximizes speed.

The Bat algorithm has several parameters. During the experiment, initial loudness was set at \(A^0_i = [1, 2]\) and start frequency to \(Q^0_i = [0, 1]\). The number of bats was set to 40. Constant \(\alpha\) was set to 0.99 and \(\gamma\) to 0.9. Functions \(f_1\) and \(f_2\) were tested with one dimension (\(D = 2\)), functions \(f_3\) to \(f_5\) were tested with three different dimensions (\(D = 10, 20, 30\)). The number of iterations for all test functions (\(f_1\) to \(f_5\)) and function dimensions (\(D = 2\) or \(D = 10, 20, 30\)) was set to 10,000.

Each experiment was executed 100 times with different random seeds. In the experiment we measured - results of standard benchmarks functions and elapsed time to complete 10,000 iterations of the algorithm. Results are presented in Table 1. Each Table row contains information about the Result and the elapsed Time. Result is the average benchmark function result of 100 experiment runs; Time is the average elapsed execution time of 100 runs. The measurement’s Standard Deviation is located inside the brackets. Figures 1-7 are the graphical representations of average elapsed time for all three
implementations. Each Figure represents one benchmark function with all dimensions.

Figure 1: Comparison of the average execution time for function $f_1$.

Figure 2: Comparison of the average execution time for function $f_2$.

Figure 3: Comparison of the average execution time for function $f_3$.

Figure 4: Comparison of the average execution time for function $f_4$.

Figure 5: Comparison of the average execution times for function $f_5$.

Figure 6: Comparison of the average execution times for function $f_6$.

Figure 7: Comparison of the average execution times for function $f_7$.

Table 1: Results of the average benchmark function result and the average elapsed execution time of 100 runs.

<table>
<thead>
<tr>
<th>Function</th>
<th>D</th>
<th>CPU Result:</th>
<th>CPU Time:</th>
<th>CUDA Result:</th>
<th>CUDA Time:</th>
<th>OMP Result:</th>
<th>OMP Time:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>2</td>
<td>-8,90E-01 (3,13E-01)</td>
<td>5,85E-02 (1,80E-03)</td>
<td>-8,38E-01 (3,66E-01)</td>
<td>2,61E-02 (1,80E-03)</td>
<td>-2,70E-01 (4,44E-01)</td>
<td>3,58E-02 (3,74E-03)</td>
</tr>
<tr>
<td>$f_2$</td>
<td>2</td>
<td>0,00E+00 (0,00E+00)</td>
<td>5,14E-02 (6,75E-04)</td>
<td>2,36E-04 (4,65E-04)</td>
<td>2,70E-02 (2,41E-03)</td>
<td>3,32E+00 (4,72E+00)</td>
<td>3,52E-02 (3,64E-03)</td>
</tr>
<tr>
<td>$f_3$</td>
<td>10</td>
<td>1,33E-06 (1,09E-06)</td>
<td>1,11E-01 (7,14E-04)</td>
<td>3,89E+00 (5,06E+00)</td>
<td>2,62E-02 (2,61E-03)</td>
<td>5,55E-06 (5,97E-06)</td>
<td>4,68E-02 (4,13E-03)</td>
</tr>
<tr>
<td>$f_4$</td>
<td>20</td>
<td>1,32E-04 (6,32E-05)</td>
<td>1,99E-01 (1,84E-03)</td>
<td>4,26E+01 (2,57E+01)</td>
<td>2,47E-02 (2,35E-03)</td>
<td>5,14E-04 (3,05E-04)</td>
<td>6,36E-02 (4,55E-03)</td>
</tr>
<tr>
<td>$f_5$</td>
<td>30</td>
<td>2,19E-03 (6,66E-04)</td>
<td>2,84E-01 (1,41E-03)</td>
<td>1,07E+01 (4,56E+01)</td>
<td>2,47E-02 (1,20E-03)</td>
<td>6,82E-03 (2,28E-03)</td>
<td>8,03E-02 (6,40E-03)</td>
</tr>
<tr>
<td>$f_6$</td>
<td>10</td>
<td>8,97E+00 (2,17E+01)</td>
<td>1,14E-01 (1,28E-03)</td>
<td>1,92E+03 (4,89E+03)</td>
<td>2,58E-02 (2,52E-03)</td>
<td>1,74E+01 (4,26E+01)</td>
<td>4,76E-02 (4,04E-03)</td>
</tr>
<tr>
<td>$f_7$</td>
<td>20</td>
<td>2,30E+01 (3,54E+01)</td>
<td>2,02E-01 (1,14E-03)</td>
<td>4,18E+04 (4,66E+04)</td>
<td>2,51E-02 (2,13E-03)</td>
<td>2,91E+01 (2,96E+01)</td>
<td>6,40E-02 (4,10E-03)</td>
</tr>
<tr>
<td>$f_8$</td>
<td>30</td>
<td>4,44E+01 (5,53E+01)</td>
<td>2,89E-01 (2,25E-03)</td>
<td>1,14E+05 (7,90E+04)</td>
<td>2,46E-02 (1,63E-03)</td>
<td>5,95E+01 (5,29E+01)</td>
<td>8,30E-02 (9,92E-03)</td>
</tr>
</tbody>
</table>
7. DISCUSSION

Figures 1-7 and Table 1 summarize the results of our experiment. Average execution times are presented in Figures 1-7 (for benchmark functions $f_1$ - $f_7$ respectively) for each dimension ($D = 2$ or $10$, $20$ and $30$ - depends on benchmark function) and for all three implementations. Table 1 represents results by two categories: average and standard deviation of values of benchmark functions ($f_1$ - $f_7$) and average and standard deviations of elapsed execution times for 100 runs.

When comparing average values of benchmark functions, it is difficult to say that one implementation is better than the other or that they all return the same result, since they all implement the same algorithm. The biggest difference between the implementations is random number generator.

Another thing that can be observed from the results is that average execution times increase linearly with the dimension ($D=10$, $20$, $30$) growth in a single thread CPU solution and OpenMP solution (with a different factor). CUDA solution is not influenced that much by the dimension growth, which can be seen from Figures 1-7.

8. CONCLUSION

In this paper, we have presented two parallel implementations of the Bat algorithm. The first implementation was accelerated on multicore processors using OpenMP directives, the second one used the graphical processors and was implemented in CUDA.

We ran an experiment in which both implementations were tested against single threaded implementation without any acceleration. In the experiment, we measured - results of standard benchmarks functions and elapsed time to complete 10,000 iterations of the algorithm. Results for benchmarks functions with dimension size 2 ($f_1$ and $f_2$) show that CUDA implementation achieved speedup by a factor of 1.91 and 2.24 in comparison to CPU single threaded implementation and 1.31 and 1.37 in comparison to OpenMP implementation. In functions with dimension sizes 10, 20 or 30 speedup was from 4.26 to 32.85 in comparison to CPU implementation, and from 1.79 to 7.72 in comparison to OpenMP implementation. Results show that the elapsed time of single threaded and OpenMP implementations increases almost linearly with dimension size of the benchmark function, while elapsed times for CUDA implementation were independent of dimension sizes. Average results of benchmark functions for CUDA and single threaded implementation are comparable in most cases.

In future work we will focus on directives-based parallel programming models like OpenACC or OpenMP, since both support program offloading to accelerators like graphical processors. Preliminary results show that OpenACC is a good alternative to CUDA.

9. REFERENCES