

Parallel Symbiotic Organisms Search Algorithm

Absalom E. Ezugwu(✉), Rosanne Els, Jean V. Fonou-Dombeu, Duane Naidoo, and Kimone Pillay

School of Computer Science, University of KwaZulu-Natal, King Edward Road, Pietermaritzburg Campus, Pietermaritzburg 3201, KwaZulu-Natal, South Africa
{ezugwua, rosanne, fonoudombeuj, 214550818, 214553962}@stu.ukzn.ac.za

Abstract. Symbiotic organisms search algorithm is a population-based evolutionary optimization technique that is motivated by the simulation of social behaviour that emanates from the symbiosis relationship amongst organisms in an ecosystem. It is a popular global search swarm intelligence metaheuristic that is widely being used in conjunction with several other algorithms in different fields of study. Fascinatingly, the algorithm has also been shown to have the capability of optimizing several NP-hard problems in both continuous and binary search spaces. More so, because most of the modern day real-world computational problems requires machines with high processing power and improved optimization techniques, it is important to find ways to improve the speedup of the optimization process of this algorithm, as the complexity of the problems increase. Therefore, this paper explores the possibility of improving the optimization speedup and performance of the symbiotic organisms search algorithm through parallelization methods. The proposed parallelization procedure is implemented using OpenMP on a shared memory architecture and evaluated on a set of twenty mathematical test problems. The computational results of the parallel symbiotic organisms search algorithm was compared to its serial counterpart using a measure of run-time complexity.

Keywords: Symbiotic organisms search · Parallel symbiotic organisms search · OpenMP

1 Introduction

The symbiotic organisms search (SOS) algorithm is a simple yet effective population-based metaheuristic technique that has been applied widely to solve problems in both continuous and combinatorial domains [1–9]. As SOS is relatively new in the literature compared to other related state-of-the-art swarm intelligence and population based optimization metaheuristic algorithms such as the Particle Swarm Optimization (PSO), Genetic Algorithm (GA), Ant Colony Optimization (ACO), Firefly Algorithm (FA), Flower Pollination Algorithm (FPA) and Differential Evolution (DE), its development as a standard optimization tool is still in its early stage. Although several aspects of improvement for the SOS algorithm have been proposed [2–4], nonetheless others

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11623, pp. 658–672, 2019.
https://doi.org/10.1007/978-3-030-24308-1_52

require considerable research effort. Performance efficiency in terms of computational speedups with regards to using the SOS algorithm to solve complex and large scale real-world problems has not been investigated and therefore can be considered as a serious bottleneck to the algorithm's development.

The advent of parallel programming has in so many ways been useful in solving compute intensive problems, which require the use of high computational resources. Parallelization has likewise been used to improve the efficiency of other similar population-based algorithms, which is the case for PSO [10], ACO [11], GA [12], FA [13], and DE [14]. This paper proposes a parallelization strategy for the standard SOS algorithm and describes the application of the parallel SOS (PSOS) to solve a set of continuous optimization problems, which the algorithm was originally designed to solve. This paper is the first to parallelize the SOS algorithm, which makes this the main motivation of the current study.

The rest of the paper is organized as follows: Sect. 2 presents detailed discussion of some related works on other population-based metaheuristic algorithms. Section 3 gives a brief overview of the SOS algorithm and subsequently describes the parallelization model used to implement the PSOS version of the serial code. Section 4 describes the experimentation, parallel implementation strategy for the PSOS, results and discussion. Finally, Sect. 5 gives the concluding remarks and future directions.

2 Related Work

In the field of parallelizing population-based metaheuristics, Zhou *et al*. [15] parallelized the ACO algorithm on a multi-core Single Instruction Multiple Data (SIMD) CPU architecture. In this model, each ant was mapped with a CPU core and the construction of a solution, for each ant, was accelerated through vector instructions. However, it was found that this simple vectorization strategy was not suited to the SIMD CPUs. An alternative approach was proposed which involved the algorithm distinguishing between branch codes and data-intensive codes and all codes were then executed on scalar and vector units. A wide range of traveling salesman problem cases, which varied from 198 to 4461 cities, was used to assess the algorithm. The speedup achieved was 57.8 times faster than its serial counterpart. This approach was then verified by comparing it with GPU-based ACO algorithms as it proved to be highly competitive with respect to its performance.

Shonkwiler [16] parallelized the GA by executing identical copies of the algorithm on independent processors. The only communication that took place between the processes is that one process collected the final result from each of the other processes and reported it to the user. This is called identical, independent processing – IIP parallel. The theoretical analysis from this investigation showed that this particular technique achieved a speedup using m processors given by $m.s^{m-1}$ where s, the acceleration factor, is a parameter that depends on the specifications of the GA. Usually $s > 1$. Shonkwiler showed that super-linear speedup is possible with the Genetic Algorithm and algorithms similar to it.

Differential Evolution algorithms possess special attributes which enable it to be used to build asynchronous parallel versions. These parallel versions make more

sufficient use of the available computational power. Ntipteni, *et al*. [17] developed a parallel DE that made use of computer clusters. This parallelization was performed using an asynchronous implementation of the master-slave approach. The procedure was then tested with two airfoil optimization problems and the parallel version was compared to its serial counterpart. It was found that for optimization problems of low dimensions, the parallel DE algorithm behaved well, in terms of its convergence rate and the speedup of the computational procedure. For larger populations, this procedure became unstable.

Chang *et al*. [18] parallelized the PSO algorithm with three communication strategies, each of which can be used according to the data independency. The first strategy was designed for independent or loosely correlated solution parameters (Rosenbrock and Rastrigin functions). The second strategy was designed for more strongly correlated solution parameters (Griewank function), while the last strategy was a hybrid communication strategy which can be used when the properties of the parameters are unknown. The parallel PSO algorithm was effective in speeding up the algorithm with the first communication strategy. With the second strategy, performance could be improved by up to 66%. Lastly, the hybrid strategy was effective for all three functions (Rosenbrock, Rastrigin and Griewank).

In Koh *et al.* [19] a parallel asynchronous PSO (PAPSO) algorithm was introduced with the aims of enhancing its efficiency. The proposed parallel algorithm dynamically adjusts the workload allocated to each processor. Hence, all available processors, in a heterogeneous cluster, are efficiently used. This implementation was compared to a parallel synchronous implementation of PSO (PSPSO) in computing environments of both a heterogeneous and homogenous nature. Small and medium analytical test problems, as well as a biomechanical test problem, were used in order to make this comparison. The results showed that the asynchronous parallel implementation was notably more effective than its synchronous counterpart. In the case of the biomechanical test problem, PAPSO proved to be 3.5 times faster than PSPSO, which was executed on a heterogeneous cluster with 20 processors.

3 Symbiotic Organisms Search Algorithm

The SOS algorithm is inspired by the interactive behaviour between different organisms in nature. Organisms normally live with and rely on each other for survival and feeding. This kind of relationship is called symbiosis [1, 2]. There are two kinds of symbiotic relationships, namely: obligate relationships and facultative relationships [20]. Obligate relationship refers to relationship where two organisms live with each other and depend on each other for survival, while facultative relationships refer to relationships where two organisms live with each other, but do not depend on each other for survival.

The SOS algorithm was designed to handle problems in a continuous space, and the algorithmic process starts with an initial population, also known as the ecosystem. First, different organisms are randomly generated in the ecosystem, where each organism refers to the problem at hand. Each organism is attached to a specific fitness value, which mirrors the amount of change to the defined objective function.

Furthermore, new sets of solutions are generated based on the interactive behaviour between two organisms in the population or ecosystem. There are different kinds of symbiotic relationships found in nature, but the most popular relationships include: mutualism, commensalism, and parasitism. Also, there are three major phases in SOS and these phases are designed to mimic parasitism, mutualism and commensalism. In the parasitic phase, interactions between organisms benefit one party but the other is disadvantaged or harmed. Also, in mutualism phase, interaction benefit both parties, while in commensalism, interactions benefit only one party but does not harm or affect the other party. Each organism randomly interacts with the other organisms in the ecosystem through all the afore-mentioned phases. These phases are briefly explained next.

As aforementioned, mutualism describes the interaction between two different organisms that benefits both organisms. The two organisms $(x_i \, and \, x_j)$ interact with each other with the goal of increasing their chances of survival within the ecosystem [2]. As shown in Eqs. (1) and (2), new solutions for the two organisms $(x_i \, and \, x_j)$ are calculated based on their interactions.

$$X_{inew} = X_i + rand(0,1) * (X_{best} - MV * BF_1) \tag{1}$$

$$X_{jnew} = X_j + rand(0,1) * (X_{best} - MV * BF_2) \tag{2}$$

$$MV = \frac{X_i + X_j}{2} \tag{3}$$

where $rand(0,1)$ refers to random numbers between 0 and 1, $BF_1 \, and \, BF_2$ represent benefit factors for the two interacting organisms. The benefit factor represents the level of benefit for each organism – partial or full. The term MV refers to the mutual vector obtained using Eq. (3), which represents the relationship features between the two interacting organisms.

Commensalism describes the relationship between two organisms, where one of the organisms benefits and the other is not harmed or affected. During interaction, a new solution for the benefitting candidate (X_i) is calculated using Eq. (4).

$$X_{inew} = X_i + rand(-1,1) * (X_{best} - X_J) \tag{4}$$

where $rand(-1,1)$ refers to random numbers between -1 and 1, X_{best} refers to the best organism in the ecosystem.

Parasitism describes the relationship between two organisms, where one of the organisms' benefits and the other is disadvantaged. In this relationship, the parasite organism (X_i) is given a role, by creating a parasite vector. The parasite vector is created in the ecosystem by duplicating the parasite organism (X_i), and then modifying a randomly selected dimension with the aid of a random number. Further, the host organism (X_j) is randomly selected, which serves as the host to the parasite organism (X_i). The fitness value for the host and parasite organisms are both calculated, and if the fitness value of the host organism is better, then the parasite organism will be eliminated from the ecosystem, otherwise, the host organism will be eliminated. Pseudocode

for the classical SOS algorithm is presented in Algorithm 1 [1].

Algorithm 1: Standard SOS pseudocode

```
Begin SOS
 Randomly Initialize population
 Evaluate the fitness of initial population
 Retain the best solution
  While (termination point is not reached)
        Mutualism phase
        Commensalism phase
        Parasitism phase
        Evaluate fitness of the population
        Update the best solution
 End While
 Return best solution
End SOS
```

3.1 Open Multiprocessor

Open multiprocessor or OpenMP as its often called is a simple application programming interface which is used to write multithreaded applications. It consists of a set of compiler directives and library routines which makes parallelizing applications in C, C ++ and Fortran straightforward to implement [21]. Most OpenMP constructs are applied to a "structured block" which is a block of one or more statements with one point of entry at the beginning of the block and one exit point at the end of the block, with each thread executing a copy of the code in the block. OpenMP is a multithreading, shared address model, in essence the threads communicate by the use of shared variables [22]. It provides a fork-join type of parallelism which consists of a master thread that spawns a team of threads at parallel regions of the code. An illustration of the fork–join model is shown in Fig. 1, in which the parallelizable regions are identified using various coloured blocks. The sequential program execution is presented at the top of the diagram, while the fork–join program execution section is displayed on the bottom part.

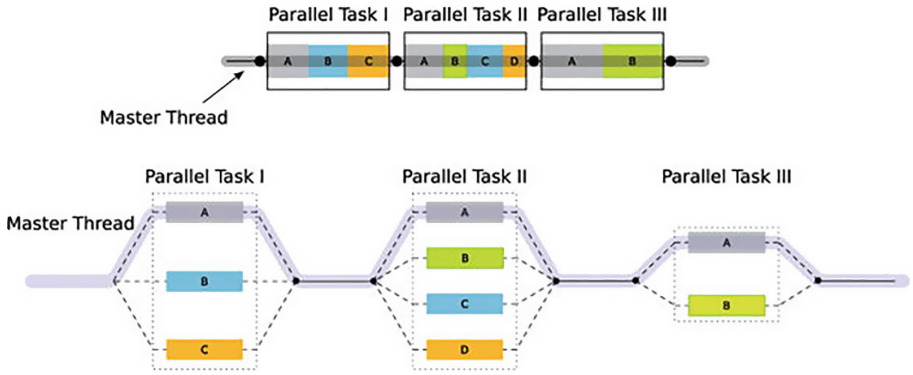


Fig. 1. OpenMP fork-join parallelism model [23]

The basic approach, when using OpenMP, is to find in a program, the specific sections with compute intensive loops and then modify them so that the iterations are independently implemented by adding the OpenMP directives. The following code snippet illustrates an OpenMP loop construct which splits the iterations among the threads.

```
#pragma omp for
for (int i = 0; i < dim; i++) {
        array_rand[i]=randon (-1, 1);
        new_x_i[i]=pop[index_i] [i] + (array_rand[i]×(best[i]-pop[index_j] [i]));
        if(new_x_i[i]<lb)new_x_i[i]=lb;
        if(new_x_i[i]>ub)new_x_i[i]=ub;
}

#pragma omp for
for (int i = 0; i < dim; i++) {
        parasite[i] = pop[index_i] [i];
}
```

The other more complex approach of parallelizing a for-loop is to write a program that shares the loop iterations among the threads instead of using the OpenMP for-loop construct. The OpenMP for-loop construct is a simplified and time-efficient way of parallelizing "for" loops. This is because just by simply adding one line of code above the "for" loop, the iterations are split among the threads that are spawned from the master thread. It is worth noting that when the number of threads is not specified, the default number of threads will be used which is equal to the number of cores on the machine that is running the program.

3.2 Test Functions

Standard mathematical test functions are used to evaluate certain attributes of an optimization algorithms. These attributes include convergence rate, precision, robustness and general performance. Since the SOS algorithm was proposed to solve single-objective optimization problems, single-objective optimization test functions were used to also investigate the performance of the proposed PSOS. In this paper, a collection of twenty different test functions is presented and used as benchmark functions to evaluate the PSOS implementation. Each benchmark function is identified by its own unique properties that are described in terms of continuity, differentiability, convexity, modality and separability properties. A comprehensive list of the selected test functions is shown in Table 1.

Table 1. Twenty mathematical test functions (D: dimension, M: multimodal, N: non-separable, U: unimodal, S: separable)

S/n	Test function	Search space range	D	Type	Min
1	Ackley	[32, 32]	30	MN	0
2	Beale	[−4.5, 4.5]	2	MN	0
3	Boachevsky1	[−100, 100]	2	US	0
4	Bohachevsky2	[−100, 100]	2	UN	0
5	Bohachevsky3	[−100, 100]	2	MN	0
6	Booth	[−10, 10]	2	US	0
7	Colville	[−10, 10]	4	MN	0
8	Dixon-Price	[10, 10]	30	UN	0
9	Griewank	[−600, 600]	30	UN	0
10	Michalewicz10	[0, π]	30	MS	−9.6602
11	Michalewicz2	[0, π]	2	MS	−1.8013
12	Michalewicz5	[0, π]	5	MS	−4.6877
13	Quartic	[−1.28, 1.28]	30	US	0
14	Rastrigin	[−5.12, 5.12]	30	MS	0
15	Rosenbrock	[−30, 30]	30	UN	0
16	Schaffer	[−100, 100]	2	UN	0
17	Sphere	[−100, 100]	30	US	0
18	Step	[−5.12, 5.12]	30	US	0
19	SumSquares	[100, 100]	30	US	0
20	Zakharov	[−5, 10]	30	UN	0

4 Experimentation

The C programming language was used for the implementation of the serial SOS algorithm on a Linux platform. Furthermore, OpenMP was used to implement the parallel programs on a multicore architecture. The experiments were conducted on a personal computer desktop with the following specifications: Processor- Intel Core i7 CPU 870 @ 2.93 GHz x 8, RAM- 4 GB, Operating system – Linux Fedora 25, Graphics card- NVIDIA GeForce GTX 470. The SOS serial code was obtained from open-source material available online in [24]. Twenty different test functions were selected, and the SOS algorithm was then executed with each of the test functions in order to obtain the serial processing time of the algorithm. The parameter setting used for the algorithm implementation were set as follows: number of iterations: 10, population size: 50, number of problem variables: 50, number of replication runs: 20, the number of dimensions varied for each of the test functions, and similarly for the upper and lower bounds. A Gprof profiler, which is a performance analysis tool for Unix applications was used to observe the implemented PSOS algorithm execution time [25].

4.1 Parallel Algorithm Methodology

Thereafter, OpenMP was applied to the serial implementation of the SOS algorithm to parallelize it. This was done differently for each test function. The reason for this was that a maximum speedup was only obtained for each test function when an OpenMP was applied to different search phases of the SOS algorithm. That is, OpenMP had to be applied to each of the SOS's algorithm phases namely, mutualism, commensalism, and parasitism respectively to achieve a speedup when using a particular test function. For example, for a particular test function, when an OpenMP call function was implemented in the mutualism phase, the computational time usually slowed down, whereas when OpenMP call function was implemented in the commensalism phase alone, a speedup in the computation time was observed. In Table 2 below, it can be seen where the OpenMP parallel model was executed for each of the different test functions in order to obtain the fastest execution time in parallel.

Table 2. OpenMP application in SOS for each test function

Fn	Test function	Objective function	Mutualism	Commensalism	Parasitism
F1	Ackley		√	√	√
F2	Beale			√	√
F3	Bohachevsky1			√	
F4	Bohachevsky2			√	√
F3	Bohachevsky3			√	√
F6	Booth			√	√
F7	Colville			√	
F8	Dixon-Price			√	
F9	Griewank		√	√	√
F10	Michalewicz2			√	
F11	Michalewicz5	√		√	√
F12	Michalewicz10		√	√	√
F13	Quartic	√	√	√	√
F14	Rastrigin	√	√	√	√
F15	Rosenbrock		√	√	√
F16	Schaffer	√		√	√
F17	Sphere		√	√	√
F18	Step			√	
F19	SumSquares			√	√
F20	Zakharov			√	√

It was found that in most cases, parallelizing the mutualism phase of the SOS algorithm actually incurred additional computational cost, while parallelizing the commensalism phase mostly reduced the execution time. For the parallel algorithm implementation, different sets of threads were employed namely, two, four and eight threads. Finally, the computational results were recorded, analysed and compared with the serial times. It is noteworthy to mention here that only the computational time complexity was used as a measure of merit in determining the performance superiority of the proposed PSOS algorithm. This is because there were no observed significant differences in the solution qualities obtained by both the serial and parallel codes. The evaluation process is further characterized by the identified differences or gaps in computational time between the serial and the parallel implementation. Furthermore, each algorithm speedup was computed using the formula represented in Eq. (5).

$$speedup = \frac{T_{serial}}{T_{parallel}} \tag{5}$$

Where T_{serial} denotes the computational time to solve a test function with the fastest serial code on a specific parallel computer, while $T_{parallel}$ is the computational time to solve the same problem with the parallel code using P processors (or thread) on the same computer.

4.2 Results and Discussion

The results obtained from the serial executions of the SOS algorithm, with the various test functions, are shown below in Table 3. The table also presents the ranges (upper and lower bounds), dimensions used, formulations, the average solutions and the standard deviations, for each test function. The averages were calculated over 20 runs. Considering the results presented in Tables 3 and 4, for the Ackley test function, when SOS was parallelized with all thread variations (two, four, and eight threads), there was a noticeable reduction in execution time mostly for two and four threads. However, the average solution and standard deviation for each of the implementations, remained consistent. A similar observation where speedup was recorded in favour of the parallel implementation as compared to their serial counterparts, includes those test functions with high dimension between ten and thirty, while the serial implementation seemed to have outperformed the parallel approach in those test functions with smaller dimensions. It can also be seen that the execution time increases as the number of threads increases.

Table 3. Serial execution times (in seconds) for the serial SOS algorithm (Avg.: average solution for 20 runs, s.dev.: standard deviation)

Fn	Function	Range	D	Avg.	s.dev.	CPU time
F1	Ackley	[−32, 32]	30	0	0	0.033878
F2	Beale	[−4.5, 4.5]	2	0	0	0.005961
F3	Bohachevsky1	[−100, 100]	2	0	0	0.005580
F4	Bohachevsky2	[−100, 100]	2	0	0	0.005515
F5	Bohachevsky3	[−100, 100]	2	0	0	0.005299
F6	Booth	[−10, 10]	2	0	0	0.004399
F7	Colville	[−10, 10]	4	−131.619	28.2474	0.006420
F8	Dixon-Price	[−10, 10]	30	0.2959	0.5911	0.027063
F9	Griewank	[−600, 600]	30	0	0	0.042255
F10	Michalewicz2	[0, π]	2	−1.8013	0	0.008939
F11	Michalewicz5	[0, π]	5	−4.6877	0	0.018634
F12	Michalewicz10	[0, π]	10	−9.3681	0.1576	0.036714
F13	Quartic	[−1.28, 1.28]	30	0.0021	0.0008	0.067570
F14	Rastrigin	[−5.12, 5.12]	30	0	0	0.037385
F15	Rosenbrock	[−30, 30]	30	24.8821	0.3386	0.031379
F16	Schaffer	[−100, 100]	2	0	0	0.008314
F17	Sphere	[−100, 100]	30	0	0	0.021405
F18	Step	[−5.12, 5.12]	30	0	0	0.023198
F19	SumSquares	[−10, 10]	30	0	0	0.021615
F20	Zakharov	[−5, 10]	10	0	0	0.013108
	Average					0.021232

In Table 4 below, the parallel results for two and four threads, are presented along with the corresponding average solutions and standard deviations. Again, the averages were calculated over 20 runs. With regard to the parallel versions of SOS with the Bohachevsky1 test function, using two threads achieved a computational time of 0.000179 s which was faster than the serial time, using four threads achieved an execution time that was 0.000033 s faster, whilst using eight threads slowed down the execution time. The least execution time was achieved with just two threads and as the thread number increased, so did the execution time. When the Bohachevsky2 test function was used, using two threads achieved a time that was 0.000082 s faster than the serial, using four threads achieved a time that was 0.000024 s faster and using eight threads slowed down the time. Again, it can be seen that the execution time increases proportionately with an increase in the number of threads. For the Bohachevsky3 test function, two threads achieved a slight decrease in execution time of 0.00020 s, four threads achieved a better execution time of 0.0000129 s, while eight threads witnessed an increase in execution time. With this particular test function, it can be seen that initially, increasing the thread number increased the speed of the execution time, but when the thread number got too large (eight threads), the execution time deteriorated. For all the three Bohachevsky functions, the average solutions and standard deviations (when compared to that of the serial SOS) remained unchanged.

Table 4. Parallel SOS algorithm results with two and four threads for each test function

Function	CPU time	Avg.	s.dev.	CPU time	Avg.	s.dev.	CPU time	Avg.	s.dev.
F1	0.031994	0	0	0.032092	0	0	0.033058	0	0
F2	0.00507	0	0	0.005141	0	0	0.006035	0	0
F3	0.005401	0	0	0.005547	0	0	0.00559	0	0
F4	0.005433	0	0	0.005491	0	0	0.00552	0	0
F5	0.005097	0	0	0.00517	0	0	0.005464	0	0
F6	0.004472	0	0	0.004323	0	0	0.004667	0	0
F7	0.006179	−130.372	28.2159	0.006186	−125.155	38.8878	0.006459	−131.633	28.2505
F8	0.025908	1.0991	2.7537	0.026	0.9075	2.673	0.027028	2.3903	6.0451
F9	0.041157	0	0	0.041252	0	0	0.042116	0	0
F10	0.009102	−1.8013	0	0.009109	−1.8013	0	0.009242	−1.8013	0
F11	0.018772	−4.6877	0	0.019441	−4.6877	0	0.018884	−4.6877	0
F12	0.035273	−9.3091	0.1867	0.036675	−9.3505	0.1916	0.037106	−9.4740	0.1514
F13	0.054024	0.0018	0.0006	0.05705	0.0023	0.0010	0.066896	0.0022	0.0011
F14	0.03042	0	0	0.031309	0	0	0.040076	0	0
F15	0.030851	24.8821	0.2451	0.0311	24.9423	0.3891	0.030557	24.9482	0.2116
F16	0.007656	0	0	0.007802	0	0	0.008011	0	0
F17	0.020077	0	0	0.020756	0	0	0.022487	0	0
F18	0.021006	0	0	0.022383	0	0	0.023545	0	0
F19	0.01159	0	0	0.021464	0	0	0.022469	0	0
F20	0.011017	0	0	0.011463	0	0	0.01301	0	0
Average	0.019025			0.019988			0.021411		

The results shown in Tables 2 and 3 also revealed that the serial SOS algorithm solved the Booth function more efficiently with less computational time when compared to the parallel version of the algorithm. Furthermore, the parallelized versions of the SOS resulted in slower execution times than that of the serial SOS when used to solve Michalewicz2 and Michalewicz5 test functions respectively. The results also show that as the number of threads increased, so did the execution time. Using the Colville function to analyse both algorithms, shows that with a parallel SOS, two threads lead to an execution time that was 0.000241 s faster than its serial counterpart. Using four threads also lead to an increase in the speed of execution, which was 0.000234 s faster than the serial time, but eight threads lead to a slower time than that of the serial implementation. However, the average solutions and standard deviations increased as SOS was parallelized using more threads.

When a parallelized SOS was run with the Dixon-Price function, using two threads produced a better execution speed which was 0.00155 s faster than the serial time, while using four threads resulted in a speed that was 0.001063 s and eight threads produced a better execution speed of 0.000035 s above the serial time. However, two and four threads obtained average solutions and standard deviations that were much higher than that of the serial time. When running the Griewank function with a parallel SOS, two threads obtained a better execution speed, four and eight threads obtained

execution speeds that were 0.001003 and 0.00039 s better than the serial program. The average solutions and standard deviations remained unchanged. The average computation cost difference (denoted as Gap) between SOS serial code and its parallel code versions is shown in Fig. 2 below.

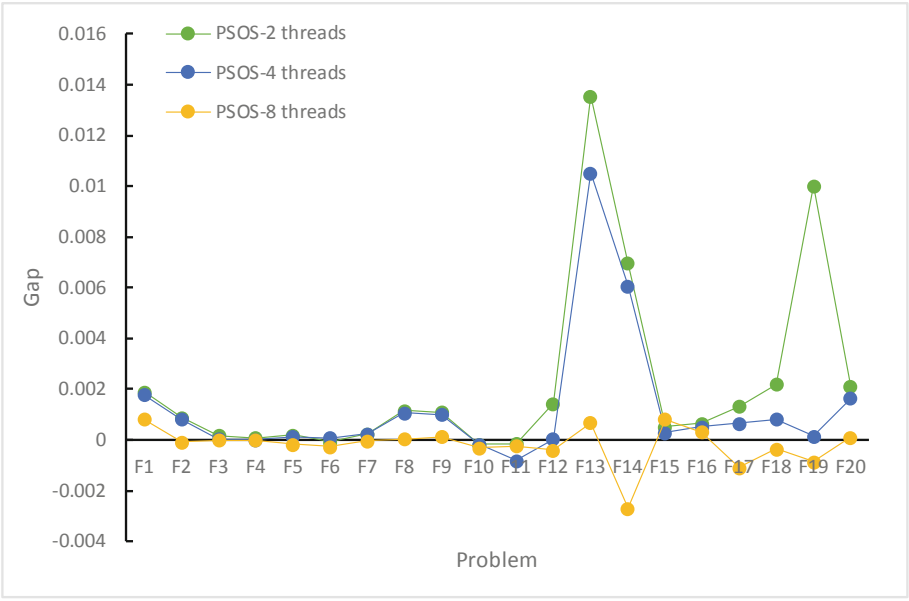


Fig. 2. Difference in execution time between SOS serial code and PSOS with two, four, and eight threads

In Fig. 3, the speedup obtained by the parallel SOS or PSOS using two, four, and eight threads is presented. Although by many measures the speedup appears to be rather poor, the illustrated results in Fig. 3 shows that in most cases speedup >1 is achieved for all the test functions, except for test function F7, F11, and F12 using two threads and F11 and F12 using four threads. However, in the case of eight threads, speed >1 were only achieved for test functions F1, F9, F10, F14, F16, F17, and F20. In general, the speedup increase as the test function problem dimension increases concurrently. Therefore, the proposed parallelization strategy could be used to decrease the amount of computational cost required to accomplish real-world problems with complex large graph sizes.

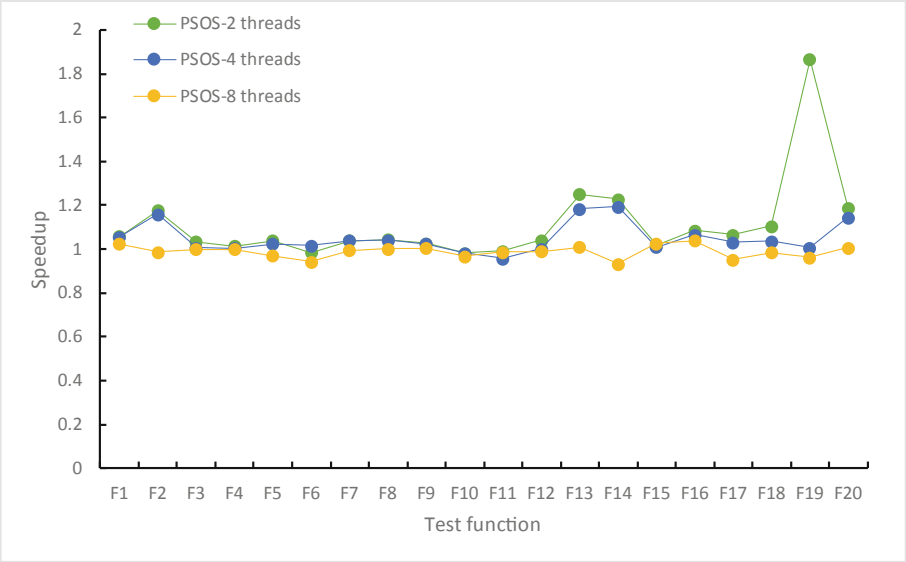


Fig. 3. Speedup on each test function using two, four and eight threads.

To summarize the entire analyses, it can be observed that the parallelization process yielded productive effort, considering the fact that reasonable execution speeds were attained in using the parallel SOS to solve test functions with high dimension. This somewhat appears to attain one of the main goals of parallel computing model. In several instances, it was also observed that using two threads for any test function produced a better speedup for the parallel SOS algorithm than its serial counterpart, and that the execution times increased as the number of threads increased. This characteristic can be attributed to the fact that the communication taking place between the threads actually creates additional computational cost overhead to the program making it less efficient. However, if using eight threads was somehow more efficient than when using four threads, this can be explained by the fact that randomness is present in the program (SOS being a population based metaheuristic) or by the fact that a particular execution time may have been an outlier when compared to the other times.

5 Conclusion and Future Work

This paper experimented the first parallel implementation techniques for the symbiotic organisms search algorithm on a multicore computing system using the well-known Open multiprocessor parallel programming model or OpenMP. The computational results obtained from the experiments showed that the SOS algorithm does possess some potential for parallelism. For example, an overall average of 0.021232 s was recorded in the case of using the serial SOS to solve the twenty test functions, while overall averages of 0.019025, 0.019988, and 0.021411 s were recorded for the PSOS respectively. However, using OpenMP may not be the best parallel programming

technique to do so because it is mostly used for loop parallelization. This work can be extended by investigating the use of either a different parallel communication strategy such as Message Passing Interface (MPI) or on a Graphic Processor Unit (GPU) to parallelize the SOS algorithm. Using test functions with higher dimensions can also be investigated to determine whether better execution times will arise due to the fact that parallelization thrives on more compute intensive operations.

References

1. Cheng, M.Y., Prayogo, D.: Symbiotic organisms search: a new metaheuristic optimization algorithm. Comput. Struct. **139**, 98–112 (2014)
2. Ezugwu, A.E., Prayogo, D.: Symbiotic Organisms Search Algorithm: theory, recent advances and applications. Expert Syst. Appl. **119**(2019), 184–209 (2018)
3. Ezugwu, A.E.S., Adewumi, A.O., Frîncu, M.E.: Simulated annealing based symbiotic organisms search optimization algorithm for traveling salesman problem. Expert Syst. Appl. **77**, 189–210 (2017)
4. Ezugwu, A.E.S., Adewumi, A.O.: Discrete symbiotic organisms search algorithm for travelling salesman problem. Expert Syst. Appl. **87**, 70–78 (2017)
5. Ezugwu, A.E., Adeleke, O.J., Viriri, S.: Symbiotic organisms search algorithm for the unrelated parallel machines scheduling with sequence-dependent setup times. PLoS ONE **13** (7), e0200030 (2018)
6. Ezugwu, A.E.: Enhanced symbiotic organisms search algorithm for unrelated parallel machines manufacturing scheduling with setup times. Knowl.-Based Syst. **172**, 15–32 (2019)
7. Govender, P., Ezugwu, A.E.: A symbiotic organisms search algorithm for optimal allocation of blood products. IEEE Access **7**, 2567–2588 (2019)
8. Govender, P., Ezugwu, A.E.: A symbiotic organisms search algorithm for blood assignment problem. In: Blesa Aguilera, M.J., Blum, C., Gambini Santos, H., Pinacho-Davidson, P., Godoy del Campo, J. (eds.) HM 2019. LNCS, vol. 11299, pp. 200–208. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05983-5_16
9. Ezugwu, A.E., Adeleke, O.J., Akinyelu, A.A., et al.: A conceptual comparison of severalmetaheuristic algorithms on continuous optimisation problems. Neural Comput. Appl. (2019). https://doi.org/10.1007/s00521-019-04132-w
10. Lalwani, S., Sharma, H., Satapathy, S.C., et al.: A survey on parallelparticle swarm optimization algorithms. Arab. J. Sci. Eng. **44**, 2899 (2019). https://doi.org/10.1007/s13369-018-03713-6
11. Randall, M., Lewis, A.: A parallel implementation of ant colony optimization. J. Parallel Distrib. Comput. **62**(9), 1421–1432 (2002)
12. Mühlenbein, H., Schomisch, M., Born, J.: The parallel genetic algorithm as function optimizer. Parallel Comput. **17**(6–7), 619–632 (1991)
13. Husselmann, A.V., Hawick, K.A.: Parallel parametric optimisation with firefly algorithms on graphical processing units. In: Proceedings of the International Conference on Genetic and Evolutionary Methods (GEM12), Number CSTN-141. CSREA, Las Vegas, USA, 16–19 July 2012 pp. 77–83, July 2012
14. Tasoulis, D.K., Pavlidis, N.G., Plagianakos, V.P., Vrahatis, M.N.: Parallel differential evolution. In: Congress on Evolutionary Computation 2004, CEC2004, vol. 2, pp. 2023–2029. IEEE, June 2004

15. Zhou, Y., He, F., Hou, N., Qiu, Y.: Parallel ant colony optimization on multi-core SIMD CPUs. Future Gener. Comput. Syst. **79**(2018), 473–487 (2017)
16. Shonkwiler, R.: Parallel genetic algorithms. In: ICGA, pp. 199–205, June 1993
17. Ntipteni, M.S., Valakos, I.M., Nikolos, I.K.: An asynchronous parallel differential evolution algorithm. In: Proceedings of the ERCOFTAC Conference on Design Optimisation: Methods and Application (2006)
18. Chang, J.F., Roddick, J.F., Pan, J.S., Chu, S.C.: A parallel particle swarm optimization algorithm with communication strategies. J. Inf. Sci. Eng. **21**(2018), 809–818 (2005)
19. Koh, B., George, A., Haftka, R., Fregly, B.: Parallel asynchronous particle swarm optimization. Int. J. Numer. Meth. Eng. **67**(4), 578–595 (2006)
20. Nama, S., Saha, A., Ghosh, S.: Improved symbiotic organisms search algorithm for solving unconstrained function optimization. Decis. Sci. Lett. **5**(3), 361–380 (2016)
21. Silberschatz, A., Gagne, G., Galvin, P.B.: Operating System Concepts. Wiley, Hoboken (2018)
22. Chapman, B., Jost, G., Van Der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming, vol. 10. MIT Press, Cambridge (2008)
23. OpenMP: Admin Magazine. http://www.admin-magazine.com/HPC/Articles/Programming-with-OpenMP. Accessed 23 Nov 2018
24. SOS source code. http://140.118.5.112:85/SOS/MOSOS.html. Accessed 23 Nov 2018
25. https://www.howtoforge.com/tutorial/how-to-install-and-use-profiling-tool-gprof/. Accessed 28 Nov 2018