PARALLEL PARTICLE SWARM OPTIMIZATION

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Priyanka Manne

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

April 2016

Fargo, North Dakota

North Dakota State University

Graduate School

Title

PARALLEL PARTICLE SWARM OPTIMIZATION

ALGORITHM: PARTICLE SWARM OPTIMIZATION

By

Priyanka Manne

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Simone Ludwig

Dr. Saeed Salem

Dr. María de los Ángeles Alfonseca-Cubero

Approved by Department Chair:

04/15/2015	Dr. Brian M. Slator
Date	Signature

ABSTRACT

PSO is a population based evolutionary algorithm and is motivated from the simulation of social behavior, which differs from the natural selection scheme of genetic algorithms. It is an optimization technique based on swarm intelligence, which simulates the bio-inspired behavior. PSO is a popular global search method and the algorithm is being widely used in conjunction with several other algorithms in different fields of study. Modern day computational problems demand highly capable processing machines and improved optimization techniques. Since it is being widely used, it is important to search for ways to speed up the process of PSO, as the complexity of the problems increase. The paper describes a way to improve it via parallelization. The parallel PSO algorithm's robustness and efficiency is demonstrated. This paper evaluates the parallelized version of the PSO algorithm with the use of Parallel Computing Toolbox available in Matlab.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Simone Ludwig for giving me this opportunity to work under her guidance, for the immense support and motivation. I am glad to find a mentor in her.

I would also like to thank my committee members for taking part in my research and for their support.

Finally I would like to thank my parents and my husband Dr. Bhavani Kundeti, for making this dream possible for me and being supportive throughout my Masters. To my friends and well wishers who had belief in me that I could do well in my career- I would always be thankful to them.

DEDICATION

Dedicated to,

My Parents,

My Sister and My Husband

Dr. Bhavani Kundeti.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Numerical optimization has been widely used in engineering to solve a variety of NP-complete problems in areas such as structural optimization, neural network training, control system analysis, design, and layout and scheduling problems to name a few. In these and other engineering disciplines, two major obstacles limiting the solution efficiency are frequently encountered. First, large-scale problems are often computationally demanding, requiring significant resources in time and hardware to solve. Second, engineering optimization problems are often plagued by multiple local optima and numerical noise, requiring the use of global search methods such as population-based algorithms to deliver reliable results. Fortunately, recent advances in microprocessor technology and network technology have led to increased availability of low cost computational power through clusters of low to medium performance computers. To take advantage of this, communication layers such as Message Passing Interface (MPI) and Parallel Virtual Machines (PVM) have been used to develop parallel optimization algorithms, the most popular being gradient-based, genetic (GA), and simulated annealing (SA) algorithms [1, 2, 3]. In biomechanical optimizations of human movement, for example, parallelization has allowed previously intractable problems to be solved in a matter of hours [1].

The Particle Swarm Optimization (PSO) algorithm is a recent addition to the list of global search methods [4]. This derivative free method is particularly suited to continuous variable problems and has received increasing attention in the optimization community. It has been successfully applied to large-scale problems [5, 6, 7] in several engineering disciplines and, being a population based approach, is readily parallelizable. It also has fewer algorithm parameters than either GA or SA algorithms. Furthermore, generic settings for these parameters work well on most problems.

Particle Swarm Optimization (PSO) is a well developed swarm intelligence method that optimizes a nonlinear or linear objective function iteratively by trying to improve a candidate solution with regards to a given measure of quality. Motivated by a simplified social model, the algorithm was first introduced by Kennedy and Eberhart in [10], where some primitive analysis of the convergence of PSO is also provided. It is an optimization technique based on swarm intelligence which simulates the bio-inspired behavior. Since the PSO algorithm requires only elementary mathematical operations and is computationally efficient in terms of both memory requirements and speed, it solves many optimization problems quite efficiently, particularly some nonlinear, non convex optimization problems. Consequently, the application of PSO has been widely seen from interdisciplinary subjects ranging from computer science, engineering, biology, to mathematics, economy [8], [9] etc. Several applications are reviewed in [10], which includes evolving neural networks, and reactive power and voltage control. Also, many researches are being done to optimize the PSO further. But, present day large-scale engineering optimization problems impose large computational demands, resulting in long solution times even on modern high-end processors. To obtain enhanced computational throughput and global search capability, the parallelization of the Particle Swarm Optimization (PSO) algorithm called Parallel Particle Swarm Optimization (PPSO) will be detailed.

The mechanism of the PSO algorithm can be briefly explained as follows. The algorithm searches the solution space of an objective function by updating the individual solution vectors called particles. In the beginning, each particle is randomly assigned to a position in the solution space and a velocity. Each particle has a memory of its previous best value and the corresponding previous best position. In addition, every particle in the swarm can know the global best value among all particles and the corresponding global best position. In each iteration, the velocity of each particle is updated so that the particle is guided by the previous best position of the particle and the global best position stochastically.

To further improve the efficiency of the PSO algorithm, the selection of the parameters becomes crucial. References [5], [6] study the relationship between convergence rate and parameter selection while [7] focuses on the impact of inertia weight and maximum velocity of PSO in which an adaptive inertia weight is equipped to guarantee the convergence rate near a minimum. On the other hand, some variations of PSO are proposed to improve the various aspects of the algorithm, not limited to efficiency.

It is difficult to find an algorithm, which is both efficient and applicable for all types of problem. As shown in previous work by Shi and Eberhart [10], the fuzzy adaptive particle swarm optimization algorithm is effective for solutions, which are independent or are loosely correlated such as the generalized Rastrigrin or Rosenbrock functions. However, it is not very effective when solutions are highly correlated such as for the Griewank function.

As the PSO algorithm is used more extensively, more research efforts are devoted to its refinement. To improve the efficiency of the PSO algorithm, we first implement it in a parallel computing way by introducing MATLAB’s built-in function parfor.

This new implementation will be more suitable to distributed and parallel computation for solving large-scale physical network optimization problems by means of high performance computing facilities. Parallel processing is concerned with producing the same results using multiple processors with the goal of reducing the running time.

Our research has indicated that the performance of the PPSO is also highly dependent on the level of correlation between the parameters and the nature of the communication strategy. With 8 nodes in the computational cluster, the PPSO algorithm has demonstrated almost linear increase in throughput. The number of nodes in the cluster depends on the complexity of the problem the PPSO is aimed to solve. This investigation proves that with large scale engineering problems PPSO will provide a new option for large global optimization problems.

2. RELATED WORK

2.1. PSO Algorithm

The particle swarm is a population-based stochastic algorithm for optimization, which is based on social–psychological principles. Unlike evolutionary algorithms, the particle swarm does not use selection; typically, all population members survive from the beginning of a trial until the end. Their interactions result in iterative improvement of the quality of the problem solutions over time.

A numerical vector of *D* dimensions, usually randomly initialized in a search space, is conceptualized as a point in a high-dimensional Cartesian coordinate system. Because it moves around the space testing new parameter values, the point is well described as a particle. Because a number of them (usually $10 < N < 100$) perform this behavior simultaneously, and because they tend to cluster together in optimal regions of the search space, they are referred to as a *particle swarm*.

The following is a brief introduction to the operation of the particle swarm algorithm. Consider a flock or swarm of p particles, with each particle's position representing a possible solution point in the design problem space D. For each particle i, Kennedy and Eberhart proposed that the position x^i be updated in the following manner:

$$x_{k+1}^{2} = x_k^i + v_{k+1}^i \tag{1}$$

With a pseudo-velocity v_{k+1}^i calculated as follows:

$$v_{k+1}^i = v_k^i + c_1 r_i\left(p_k^i - x_k^i\right) + c_2 r_2\left(p_k^g - x_k^i\right) \tag{2}$$

Here, subscript *k* indicates a (unit) pseudo-time increment, p_k^i represents the best position of particle *i* at time *k* (the cognitive contribution to the search vector,v_{k+1}^i), and p_k^g represents the global best position in the swarm at time k (social contribution), r_1 and r_2 represent uniform random numbers between 0 and 1. To allow the product c_1r_1 or c_2r_2 to have a mean of 1, Kennedy and Eberhart proposed that the cognitive and social scaling parameters c_1 and c_2 be selected such that c_1 = c_2 = 2. The result of using these proposed values is that the particles overshoot the target half the time, thereby maintaining separation within the group and allowing for a greater area to be searched.

The serial PSO algorithm as it would typically be implemented on a single CPU computer is described below, where p is the total number of particles in the swarm. The best ever fitness value of a particle at design coordinates p_k^i is denoted by f_{best}^i and the best ever fitness value of the overall swarm at coordinates p_k^g by f_{best}^g. At the initialization time step k=0, the particle velocities v_0^i are initialized to random values within the limits $0 \leq v_0 \leq v_0^{max}$. The vector v_0^{max} is calculated as a fraction of the distance between the upper and lower bound.

The algorithm shown below is explained this way, considering a swarm of particles in the space, each particle is initialized with a position value for the purpose of our experiment (these are considered as local best values), given an initial velocity at that particular position for each particle. At the initial positions (local best values), the position of a particle close to the goal is considered as global best value. Usually these positions are initialized to uniformly cover the search space. It is important to note that the efficiency of the PSO is influenced by the initial diversity of the swarm, i.e., how much of the search space is covered, and how well the particles are distributed over the search space. If regions of the search space are not covered by the initial swarm, the PSO will have difficulty finding the optimum if it is located within the uncovered region. The PSO will discover such an optimum only if the momentum of the particle carries the particle into the uncovered area, provided that the particle ends up on either a new personal best for itself, or a position that becomes the new global best. Thus we define the range vectors that cover the search space. So, when the velocity (momentum) of the particle changes, it changes its position to a new position his process is repeated until a global best value in the swarm that is close to the goal is achieved.

2.1.1. Step-by-Step Algorithm for Serial PSO:

1. Initialize

    (a) Set constants k_{max}, c_1, c_2

    (b) Randomly initialize particle positions $x_0^i \in$ D in IR^n for i=1,...,p

    (c) Randomly initialize particle velocities $0 \leq v_0^i \leq v_0^{max}$ for i=1,...,p

    (d) Set k=1

2. Optimize

    (a) Evaluate function value f_k^i using design space coordinates x_k^i

    (b) If $f_k^i \leq f_{best}^i$ then $f_{best}^i = f_k^i$, $p^i = x_k^i$

    (c) If $f_k^i \leq f_{best}^g$ then $f_{best}^g = f_k^i$, $p^g = x_k^i$

    (d) If stopping condition is satisfied then go to Step 3

    (e) Update particle velocity vector v_{k+1}^i using Eq. (2)

    (f) Update particle position vector x_{k+1}^i using Eq. (1)

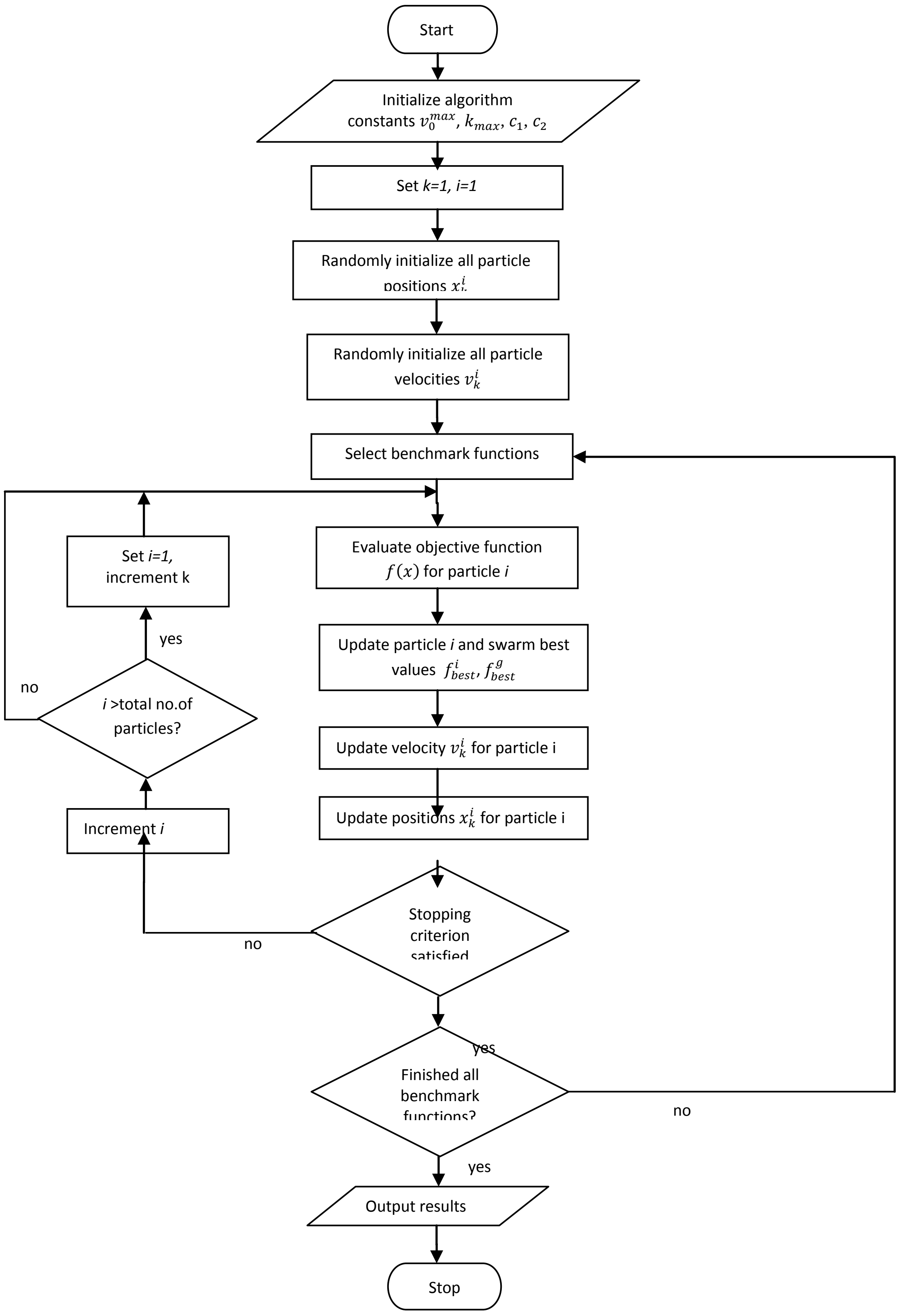
    (g) Increment I; if i>p then increment k, and set i=1

    (h) Go to Step 2(a)

3. Report results

4. Terminate

The above algorithm description is illustrated as a flow diagram in Figure 1.



Start
Initialize algorithm constants v_0^{max}, k_{max}, c_1, c_2
Set *k=1, i=1*
Randomly initialize all particle positions x_k^i
Randomly initialize all particle velocities v_k^i
Select benchmark functions
Evaluate objective function $f(x)$ for particle *i*
Update particle *i* and swarm best values f_{best}^i, f_{best}^g
Update velocity v_k^i for particle i
Update positions x_k^i for particle i
Stopping criterion satisfied
no
Increment *i*
i >total no.of particles?
yes
no
Set *i=1*, increment k
yes
Finished all benchmark functions?
no
yes
Output results
Stop

Figure 1: Flowchart of serial PSO

2.2.1. Parallel Particle Swarm Optimization Algorithm

The following issues had to be taken into consideration to create a parallel PSO algorithm.

2.2.1. Concurrent Operation and Scalability

The algorithm should operate in such a fashion that it can be easily decomposed for parallel operation on a multi-processor machine. Furthermore, it is highly desirable that it be scalable. This implies that the nature of the algorithm should not place a limit on the amount of computational nodes that can be utilized.

An example of an algorithm with limited scalability is a parallel implementation of a gradient-based algorithm. This algorithm is decomposed by distributing the workload of the derivative calculations for a single point in design space among multiple processors. The upper limit on concurrent operations using this approach is therefore set by the number of variables in the problem. On the other hand, population-based methods such as the GA and PSO algorithms are better suited to parallel computing. Here the population of individuals representing designs can be increased or decreased according to the availability and speed of processors. Any additional agents in the population will allow for a higher fidelity search in the design space, lowering susceptibility to entrapment in local minima. However, this comes at the expense of additional fitness evaluations.

2.2.2. Coherence

Parallelization should have no adverse affect on algorithm operation. Calculations sensitive to program order should appear to have occurred in exactly the same order as in the original formulation, leading to the exact same final answer as obtained by a serial implementation. In the serial PSO algorithm the fitness evaluations form the bulk of the computational effort for the optimization and can be performed independently. For our parallel implementation, we therefore chose to use the parfor, in the algorithm in such a way that the executions are independent as much as possible.

2.2.3. Network Communication

In a parallel computational environment, the main performance bottleneck is the communication latency between processors. This is especially true for large clusters of computers where the use of high performance network interfaces is limited due to their high cost. To keep communication between different computational nodes at a minimum, proper placement of the parfor ensures that the iterations are independent and not many resources are shared between the nodes. In MATLAB, all the communications between the nodes is taken care of by the client machine, the section that goes under the parfor loop is analyzed before the execution and the information is distributed between the workers.

The algorithm described below follow the same sequence of steps described in the serial PSO, except that the loops (iterations) are distributed between number of nodes defined before start of execution rather than on a single PC in serial execution. In addition, a number of fitness functions are defined to optimize the position values. The resources between the threads i.e., the local best, global best values and other parameters in the code are managed by the Matlab tool box. Considering all the issues discussed above an optimal number of nodes are chosen to execute the following PPSO such that the overhead for using the nodes are minimized. For example one of the overhead is that invoking a node and starting it up by telling it what it has to do can cause latency. But it is negligible considering the time taken to run the serial PSO. As discussed the number of nodes should be invoked depending on the complexity that includes the number of particles, the dimensions, number of iterations etc.

2.2.4. Step-by-Step Algorithm for parallel PSO

1. Initialize

    (a) Set values for ps (no. of particles), D (Dimension), me (no. of iterations)

2. Optimize

2.1. Benchmark function

    (a) Select one of the 7 Benchmark functions

2.2. Parallelize

    (a) Randomly initialize particle positions $x_0^i \in$ D in IR^n for i=1,...,p

    (b) Randomly initialize particle velocities $0 \leq v_0^i \leq v_0^{max}$ for i=1,...,p

    (c) Evaluate function value f_k^i using design space coordinates x_k^i

    (d) If $f_k^i \leq f_{best}^i$ then $f_{best}^i = f_k^i$, $p^i = x_k^i$

    (e) If $f_k^i \leq f_{best}^g$ then $f_{best}^g = f_k^i$, $p^g = x_k^i$

    (f) If stopping condition is satisfied then go to Step 2.3

    (g) Update particle velocity vector v_{k+1}^i using Eq. (2)

    (h) Update particle position vector x_{k+1}^i using Eq. (1)

    (i) Increment I; if i>p then increment k, and set i=1

    (j) If stopping condition is satisfied for the fitness function then go to Step 3

    (k) Go to Step 2.1(a)

  2.3 Update Results

3. Update Global best

4. Terminate

The above algorithm description is illustrated as a flow diagram in Figure 2.

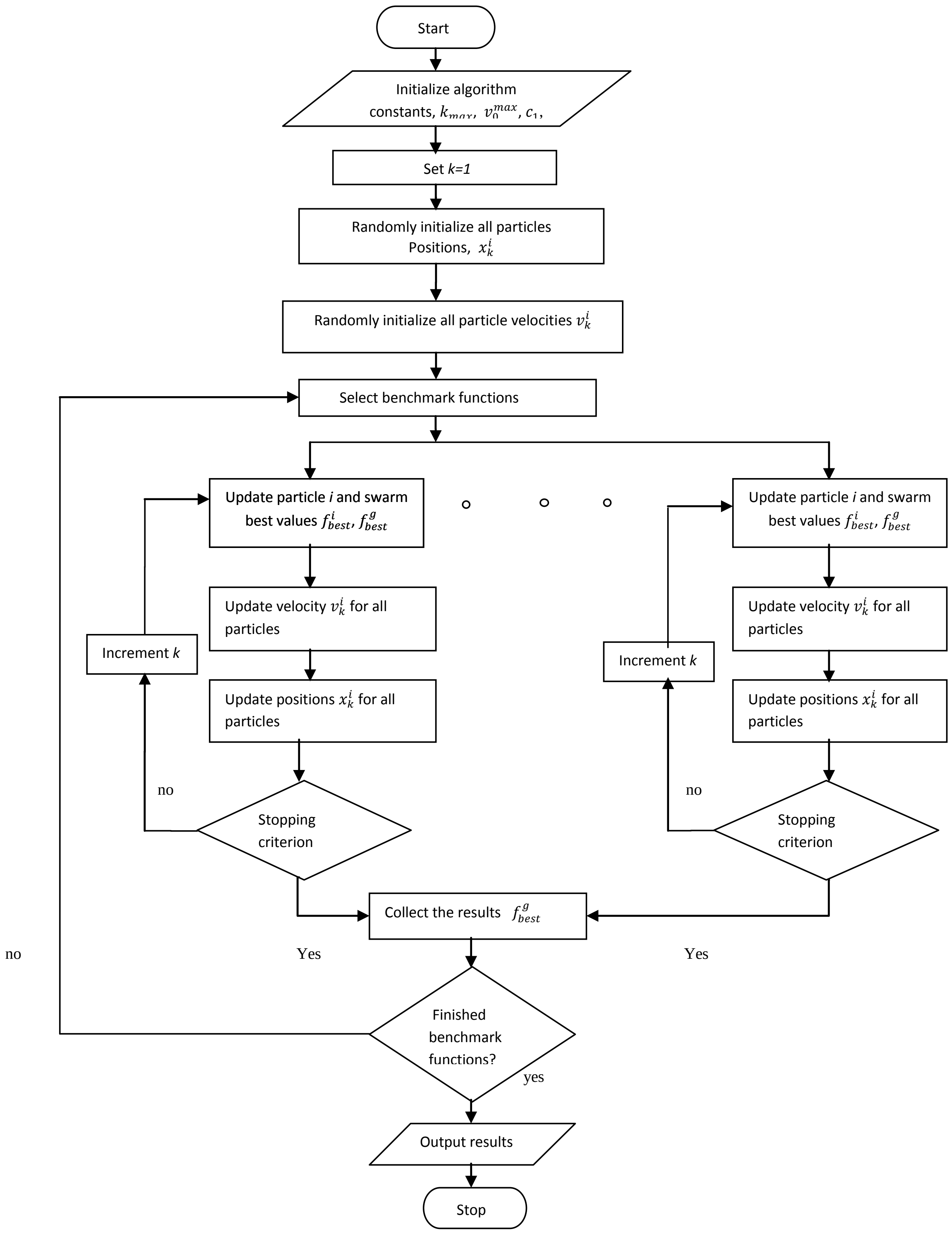

Start
Initialize algorithm constants, k_{max}, v_0^{max}, c_1,
Set $k=1$
Randomly initialize all particles Positions, x_k^i
Randomly initialize all particle velocities v_k^i
Select benchmark functions
Update particle i and swarm best values f_{best}^i, f_{best}^g
Update velocity v_k^i for all particles
Update positions x_k^i for all particles
Increment k
no
Stopping criterion
Update particle i and swarm best values f_{best}^i, f_{best}^g
Update velocity v_k^i for all particles
Update positions x_k^i for all particles
Increment k
no
Stopping criterion
Collect the results f_{best}^g
no
Yes
Yes
Finished benchmark functions?
yes
Output results
Stop

Figure 2: Flowchart for Parallel PSO

The optimization infrastructure is organized into a coordinating node and several computational nodes. PSO algorithm functions and task orchestration are performed by the coordinating node, which assigns the design coordinates to be evaluated, in parallel, to the computational nodes. With this approach, no communication is required between computational nodes as individual design fitness evaluations are independent of each other. The only necessary communication is between the coordinating node and the computational nodes and encompasses the following:

1. Several distinct design variable configuration vectors assigned by coordinating node to slave nodes for fitness evaluation.
2. Fitness values reported from slave nodes to coordinating node.
3. Synchronization signals to maintain program coherence.
4. Termination signals from coordinating node to slave nodes on completion of analysis in order for the program to stop cleanly.

2.2.5. Synchronization

From the parallel implementation algorithm, it is clear that some means of synchronization is required to ensure that all of the particle fitness evaluations have been completed, the velocity and position calculations are executed (Steps 2c and 2d) before the results are reported. This is taken care of by the Matlab tool box itself, synchronizing the computational nodes by temporarily stopping the coordinating node from proceeding with the next run until all of the computational nodes have responded with a global best value.

3. PROBLEM DESCRIPTION

This chapter deals with the problem definition and detailed description of the Parallel Particle swarm optimization algorithm. Parallel processing is used to achieve the same results using multiple processors with the goal of reducing the running time. In serial PSO, with the increase in search space, i.e., dimension, number of particles, number of iterations and number of benchmark functions the processing time is going to increase exponentially. To reduce the processing time, this paper uses Matlab's Parallel Computing Toolbox. Matlab provides a powerful interface and platform to run scaling experiments with huge amounts of sample data. With so many built-in features for a wide range of users from researchers to students who needed an easy to use platform for all of their mathematical computations, Matlab provides it all. We chose Matlab for its ability to provide readily available and easy to use Parallel Processing feature. To use this feature, we need to have multiple cores or workers (in general PCs) ready and connected through a network in a cluster to a PC as coordinating node and have Matlab installed on them; this is called a Matlab pool. Our OS manages these connections to the workers depending on the requirements specified by us. When we run the program on Matlab, a default 'local profile' will be created with the configurations and information of the available number of workers. But this 'local profile' can be changed within our program. The Particle Swarm Optimization algorithm is modified by adding parallelization code that will invoke several workers in the Matlab pool.

Parallelization in Matlab can be done in different ways, one of which we used in our project to achieve parallelization is by using "parfor". Parfor-loop technique is used in our PSO logic to loop through the code instead of regular for-loop. When parfor loop is used instead of for loop the serial execution is converted to parallel execution. Here, each loop under the parfor loop section is distributed between the workers and Matlab takes care of the shared data between each loop. We have to make sure shared resources are kept to the minimum as the communication between the threads/workers gets complex with the number of loops that are being shared. The resource sharing and communication between the threads are taken care of by the Matlab itself. The necessary data on which parfor operates is sent from the client to workers, where most of the computation happens, and the results are sent back to the client and pieced together. We need to consider various possibilities of where to parallelize the code, weigh their pros and cons, compare the results and choose the best way to parallelize the PSO. Proper placement of parfor also decides the quality of the parallelization. By following the guidelines by Matlab as to how and where to use the parfor in our code is critical for smooth transition from the serial code to parallel code.

When we compare the time taken to run the PSO code without parallelization and PSO code with parallelization we can see a significant difference. Although the use of parallel code has some overhead, the time taken with parallelization is still appreciable as compared with non-parallelized code. When we try to parallelize any algorithm in general we cannot see a difference if we use 1 or 2 nodes, selecting the number of nodes to parallelize is crucial, it is dependent on the complexity of our problem we are trying to solve. Here in our problem, we are considering varying number of particles from (50 to 300), varying number of dimensions (2 to 200), and varying number of iterations (up to 200) also with varying the number of computational nodes (up to 8). Since this experiment is mainly focused on testing the PSO algorithm for “parallelism” and how we go about determining the number of nodes we need in the computational cluster depending upon the complexity of the problem, we are not considering additional optimization techniques to go with the PSO algorithm.

The Parallel Computing Toolbox has guidelines on how to incorporate parallel processing into an existing block of code with serial execution. It also explains what types of serial code are eligible for parallelization and which are not. It addresses any issues while converting the serial code to parallel code and make suggestions on how to convert it. Moreover, the client machine where the Matlab resides takes care of the delegation of tasks to the workers in the cluster and coordination between the workers.

To parallelize the existing serial PSO, we have used “parfor” as described in the previous section. Everything that goes inside of a parfor loop is scrutinized for any complexities or interdependencies. If the resources or variables are tightly-coupled or highly cohesive the Parallel Computing Toolbox rejects the code. We have to make sure our code is complying with all the guidelines and rules to go through parallelization. We have to modify the code as suggested by the toolbox.

Next step is finding the cluster with the desirable number of workers on it. If the workers in the clusters are too few, we may not be able to see much difference in the processing time while the workers are too many, it may not be economical and moreover, after some point the processing time remains constant since the minimum time to create the “pool” of workers, assigning tasks and coordination cannot be skipped, this is called partitioning overhead with using parfor, however, the processing time is still considerably less than serial PSO. We chose the cluster in such a way that, each iteration inside of parfor can be distributed to each worker in the cluster, though the sequence in which they are executed is never known and never the same. We made sure that the number of runs in the PSO code matches with the maximum number of nodes in the cluster to study the distribution of the iterations between the nodes, their results and time taken to complete the execution. The Matlab tool box shows us the performance graph of each node that is invoked in the computational cluster during execution. We can study when a particular node is utilized to its maximum performance and when it is utilized less. We can actually study the behavior of the nodes in the cluster and how the execution flows through the nodes in the cluster. We believe this is a good way to analyze the behavior of the system itself. It gives us an inside look at how processing takes place and how they communicate with each other at the machine level.

A Matlab “pool” is a group of machines we will need to execute the parallel code on. We need to specify Matlab on how many workers or nodes we need in a pool to work for us, within the program. For that, we need to start the Matlab pool by using “open” command from within the program. Accordingly, Matlab divides the tasks and assign them to the workers. Once the job is done, we have to “close” the Matlab pool or it runs default until Matlab is closed.

4. IMPLEMENTATION DETAILS

The application developed as a part of this project is a MATLAB application consisting of different functions. The PSO code sits on a PC with Windows 7 OS and MATLAB installed on it during these tests. For the program to run on multiple machines we need to be connected to a cluster of at least 8 computers at all time. The computer on which the Matlab sits is considered a Client machine. The client machine should be connected to the cluster of computers through a network. We briefly discuss those functions later in this chapter. Inputs required for this application are the number of iterations, dimension and number of particles. All these inputs are given as parameters at the beginning of the algorithms.

4.1. Test.m

This is the main function, which calls all other functions from within. All the inputs needed for the PSO code are initialized at the beginning. The Matlab pool is created here by specifying the number of workers needed and the pool is opened. Parallelization starts from the next line after “parfor”.

Start up MATLAB in the regular way. This copy of MATLAB that you start with is called the "client" copy; the copies of MATLAB that will be created to assist in the computation are known as "workers". The process of running your program in parallel now requires three steps:

1. Request a number of workers;
2. Issue the normal command to run the program. The client program will call on the workers as needed;
3. Release the workers.

Open or close pool of MATLAB sessions for parallel computation, enables the parallel language features in the MATLAB language (e.g., parfor) by starting a parallel job that connects this MATLAB client with a number of workers.

On the other hand if we want to have much more control over the data that is being shared between workers, then we can always use the spmd command to achieve parallelization. MATLAB also works in a simplified kind of MPI model for parallel processing. There is a special "client" process, each worker process has its own memory and separate ID. Though it is a single program, it is divided into client and worker sections by special spmd statements. Workers can "see" the client's data; the client can access and change worker data. The workers can also send messages to other workers. SPMD programming makes use of distributed arrays, which is logically one array, and a large set of MATLAB commands can treat it that way. However, portions of the array are scattered across multiple processors. This means such an array can be really large. The local part of a distributed array can be operated on by that processor very quickly. A distributed array can be operated on by explicit commands to the SPMD workers that "own" pieces of the array, or implicitly by commands at the global or client level.

4.2. PSO_fun.m

This is the function where we initialize the velocities and positions of the particles. We then update them by using Equation (1) and compare the personal best positions of all the particles in a run and get the global best position. Likewise, the second run and so on.

4.3. Test_func.m

One of the shortcomings of population based search techniques is that there are not many concrete proofs available to establish their authority for solving a wide range of problems. Therefore the researchers often depend on empirical studies to scrutinize the behavior of an algorithm. The numerical problems may be divided into two classes; benchmark problems and real life problems. For the present study we have taken seven benchmark functions to analyze the behavior of our PSO algorithm. These benchmark functions are selected from a group of benchmark functions that are categorized as below, based on their properties:

1) unimodal problems,
2) unrotated multimodal problems,
3) rotated multimodal problems and
4) composition problems.

The properties of these functions are presented below.

Group1: Unimodal and Simple Multi-modal Problems:

1) Sphere function

$$f_1(x) = \sum_{i=1}^{D} x_i^2 \tag{3}$$

2) Rosenbrock's function

$$f_2(x) = \sum_{i=1}^{D-1} (100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2) \tag{4}$$

The first problem is the sphere function is a continuous, strictly convex and unimodal function and usually do not pose much difficulty for an optimization algorithm and is easy to solve. The second problem is the Rosenbrock function. It can be treated as a multimodal problem. It has a narrow valley from the perceived local optima to the global optimum.

Group 2: Unrotated Multimodal Problems:

In this group, there are six multimodal test functions. We are only using 5 of them in this project. Ackley's function has one narrow global optimum basin and many minor local optima. It is probably the easiest problem among the six as its local optima are neither deep nor wide.

Griewank's function has a $\prod_{i=1}^{D} \cos(\frac{x_i}{\sqrt{i}})$ component causing linkages among dimensions thereby making it difficult to reach the global optimum. An interesting phenomenon of Griewank's function is that it is more difficult for lower dimensions than higher dimensions 11. Rastrigin's function is a complex multimodal problem with a large number of local optima. When attempting to solve Rastrigin's function, algorithms may easily fall into a local optimum. Hence, an algorithm capable of maintaining a larger diversity is likely to yield better results. Non-continuous Rastrigin's function is constructed based on the Rastrigin's function and it has the same number of local optima as the continuous Rastrigin's function. The complexity of Schwefel's function is due to its deep local optima being far from the global optimum. It will be hard to find the global optimum, if many particles fall into one of the deep local optima. Schewefel function and Ackley function are both again multimodal functions having several optima. Such functions provide a suitable platform for testing the credibility of an optimization algorithm.

3) Ackley's function

$$f_3(x) = -20\, exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^2}\right) - exp\left(\frac{1}{D}\sum_{i=1}^{D} cos(2\pi x_i)\right) + 20 + e \tag{5}$$

4) Griewanks's function

$$f_4(x) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} cos(\frac{x_i}{\sqrt{i}}) + 1 \tag{6}$$

5) Rastrigin's function

$$f_5(x) = \sum_{i=1}^{D} (x_i^2 - 10\, cos(2\pi x_i) + 10) \tag{7}$$

6) Non-continuous Rastrigin's function

$$f_6(x) = \sum_{i=1}^{D} (y_i^2 - 10\, cos(2\pi y_i) + 10),$$

$$y_i = \begin{cases} x_i & |x_i| < 1/2 \\ round(2x_i)/2 & |x_i| >= 1/2 \end{cases} \text{ for } i = 1, 2, .., D \tag{8}$$

7) Schwefel's function

$$f_7(x) = 418.9829 \times D - \sum_{i=1}^{D} x_i \, sin(|x_i|^{1/2}) \tag{9}$$

Table 1 shows the global optimum, search ranges and initialization ranges of the benchmark problems.

Table 1: Global optimum, search ranges and initialization ranges of the benchmark problems

f	$\mathbf{x}^*$	$f(\mathbf{x}^*)$	Search Range	Initialization Range
f_1	[0,0,…,0]	0	$[-100, 100]^D$	$[-100, 50]^D$
f_2	[1,1,…,1]	0	$[-2.048, 2.048]^D$	$[-2.048, 2.048]^D$
f_3	[0,0,…,0]	0	$[-32.768, 32.768]^D$	$[-32.768, 16]^D$
f_4	[0,0,…,0]	0	$[-600, 600]^D$	$[600, 200]^D$
f_5	[0,0,…,0]	0	$[-0.5, 0.5]^D$	$[-0.5, 0.2]^D$
f_6	[0,0,…,0]	0	$[-5.12, 5.12]^D$	$[-5.12, 2]^D$
f_7	[0,0,…,0]	0	$[-5.12, 5.12]^D$	$[-5.12, 2]^D$

5. EXPERIMENTS AND RESULTS

The experiments that were conducted as part of this project proves the theory discussed above. The first step is to examine the time taken to run the original PSO code on a single core without any parallelization and then compare the result of the experiment with the result of another experiment that was conducted using the parallel PSO code on a single core as well. In both the experiments all benchmark functions discussed above are evaluated. The comparison of the two experiments are different, in the sense that the original PSO on a single core is faster than parallel PSO on the same single core. In this scenario the results of original PSO code and parallel PSO yielded 358 seconds and 736 seconds, respectively with dimension (D)=2, and number of particles (PS)=100. This difference is because of the overhead of the parallel code invoking a Matlab pool, which is a costlier process in terms of time. However, for higher-dimensional problems and by executing the code with a larger number of cores (depending on the size of the problem) in a cluster would actually minimize the effect. The following experiments are all done on a cluster of 8 cores, which is considered as the Matlab pool.

We considered for these experiments a scenario where the sample space containing 0-350 particles, the dimensions between 0 and 10 and with varying number of iterations to run the PSO code.

Scaling Experiment 1: Figure 3 shows the time taken to run the Parallelized PSO code with varying number of workers (i.e. number of threads=1,2,3,4,5,6,7,8) keeping all other variables constant, maximum number of iterations as 200, number of particles as 100, and dimension as 2. This experiment is done to learn the optimal number of workers that are needed for our experiment.

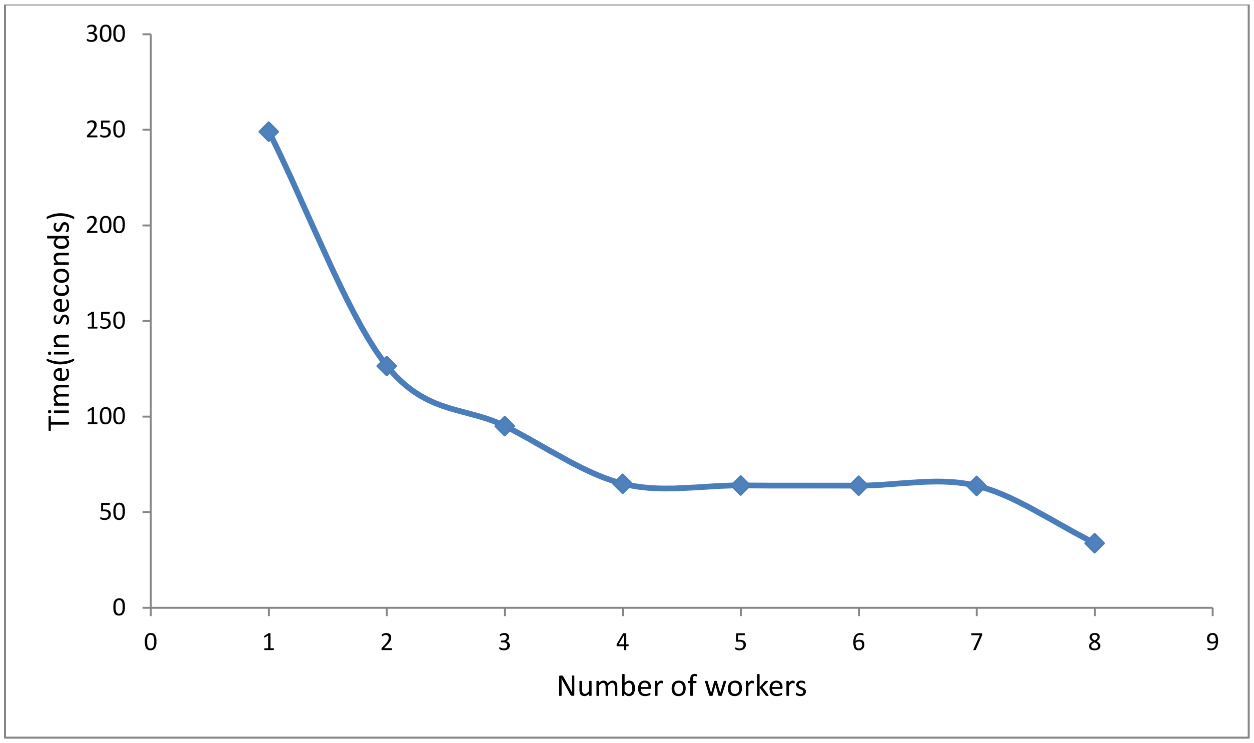


Figure 3: Running time versus number of workers

The figure shows the time taken for the parallel PSO code with varying number of workers with all other parameters kept constant. In the above figure, as we can see by increasing the number of cores the time taken to complete the work is considerably decreasing.

Result: The result of this experiment is that by increasing the number of cores or processing power the time taken to run the parallel PSO code is considerably reduced.

Scaling Experiment 2: Figure 4 shows the scenario in which the sample space contains 50 particles but with varying dimensions. This is a study to find out the time taken to run the PSO code when the dimensions of the sample space change. The time taken to run the parallelized PSO code is recorded with varying dimensions (D=10,20,30,40,50,60,70,80,90,100) while using 8 workers, number of particles as 50, and maximum number of iterations as 200.

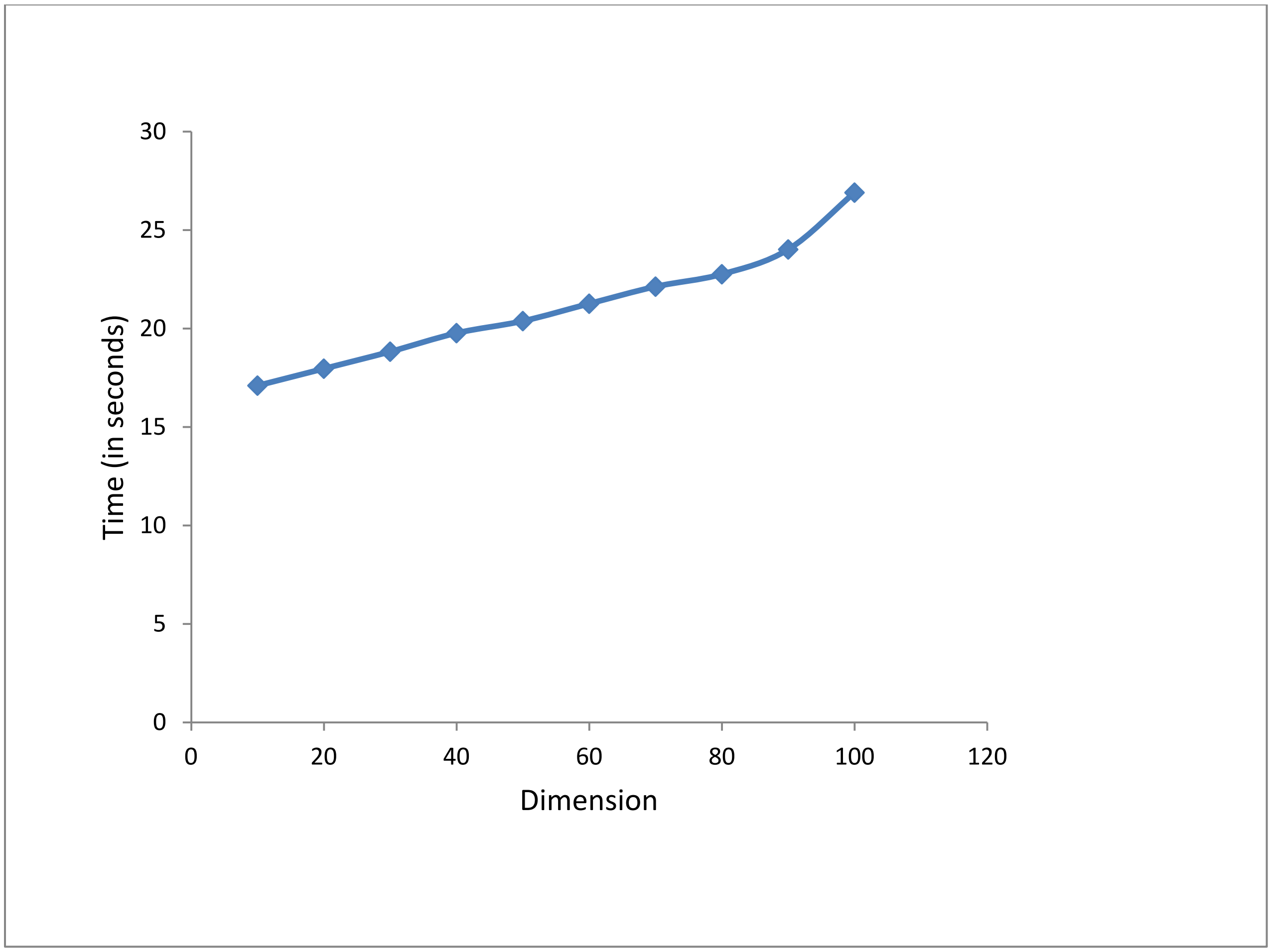


Figure 4: Running time versus number of dimensions with number of particles=50

Result: The above experiment proves that as the increasing dimensions increase the running time of the PSO algorithm.

Scaling Experiment 3: Figure 5 shows the time taken to run the parallelized PSO code with varying dimensions (D=10,20,30,40,50,60,70,80,90,100) while using 8 workers, number of particles as 100, and maximum number of iterations as 200.

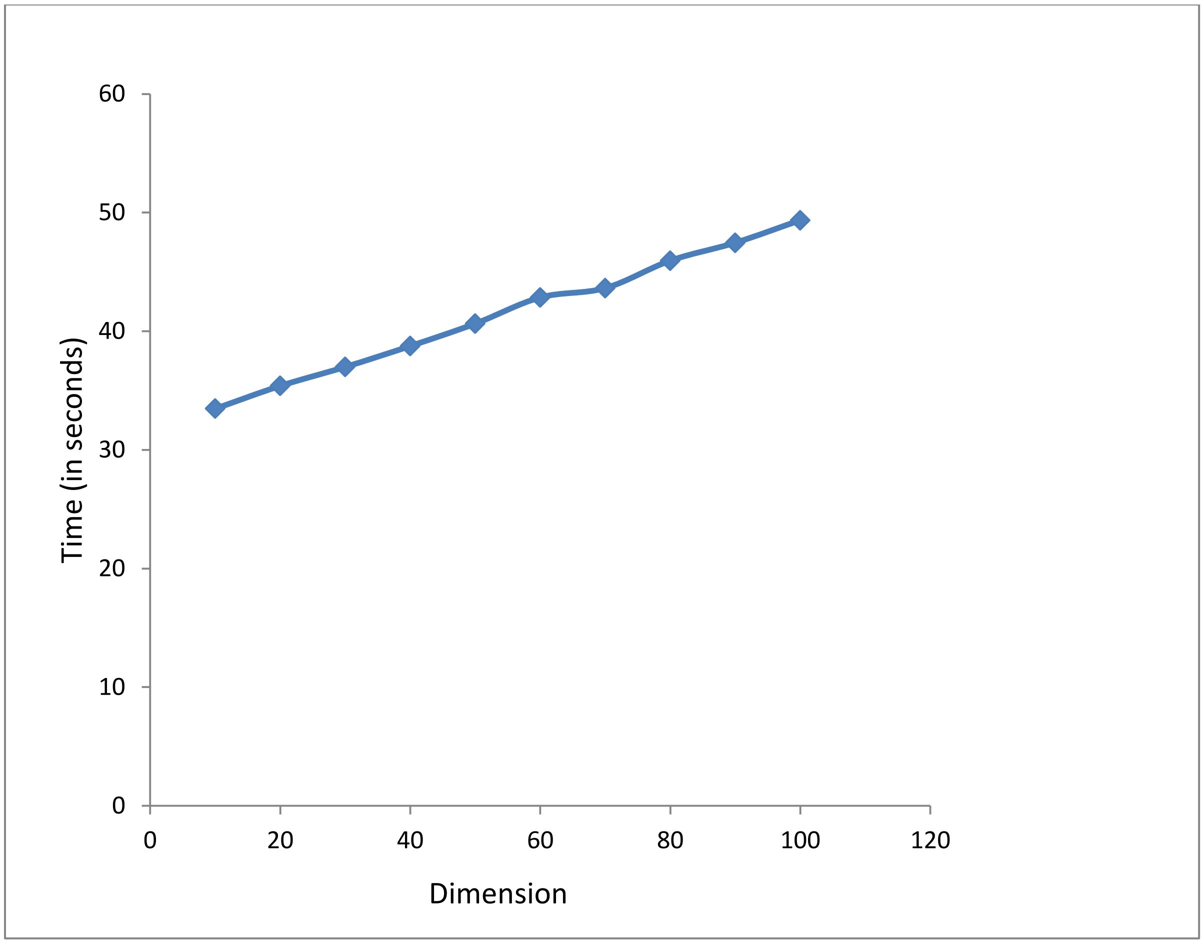


Figure 5: Running time versus number of dimensions with number of particles=100

Result: In this experiment the numbers of particles are increased to 100 and the varying dimensions are observed similar to the previous experiment. This figure shows that the time taken has increased compared to the previous experiments but the resultant times were between 20 seconds.

Scaling Experiment 4: Figure 6 shows the time taken to run the parallelized PSO code with varying dimensions (D=10,20,30,40,50,60,70,80,90,100) while using 8 workers, number of particles as 150, and maximum number of iterations as 200.

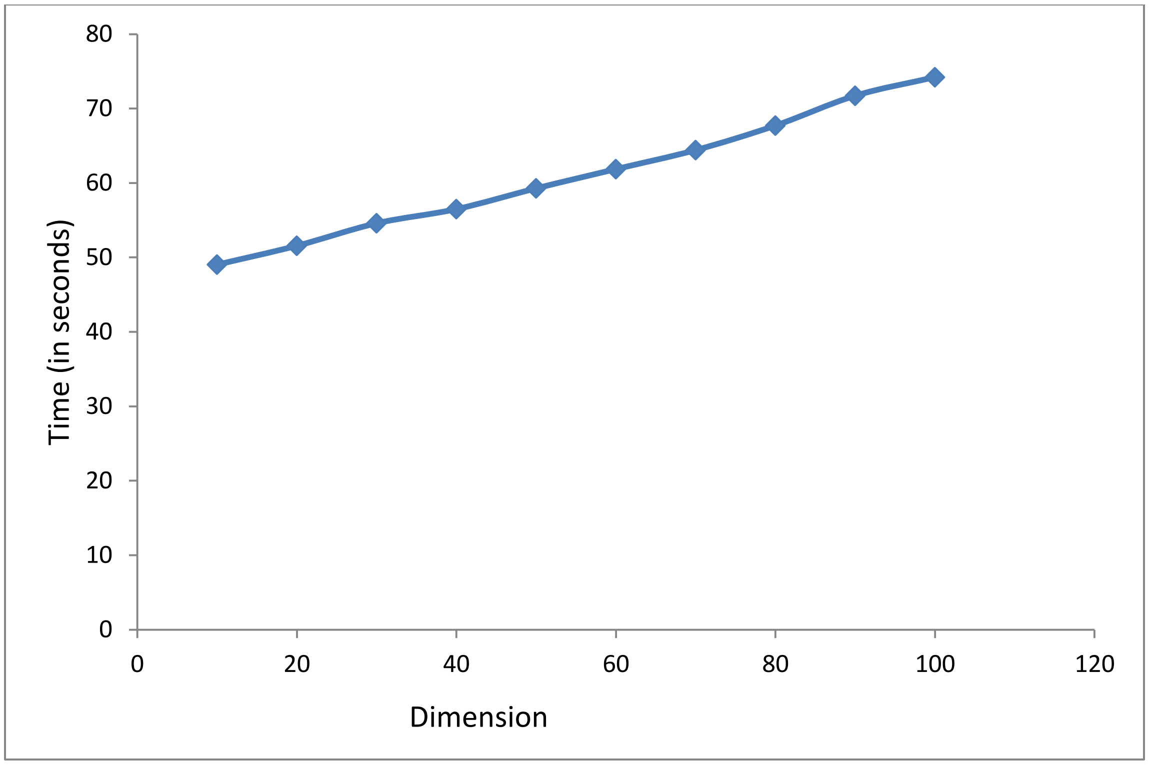


Figure 6: Running time versus number of dimensions with number of particles=150

Here the experiment is conducted using the number of particles as 150 and the same number of varying dimensions as used during the previous experiments.

Result: In this experiment the trend persists to that of the previous two experiments that the resultant time increases but remains between 50 and 70 seconds (within 20 seconds).

Scaling Experiment 5: Figure 7 shows the time taken to run the parallelized PSO code with varying dimensions (D=10,20,30,40,50,60,70,80,90,100) while using 8 workers, number of particles as 200, and maximum number of iterations as 200.

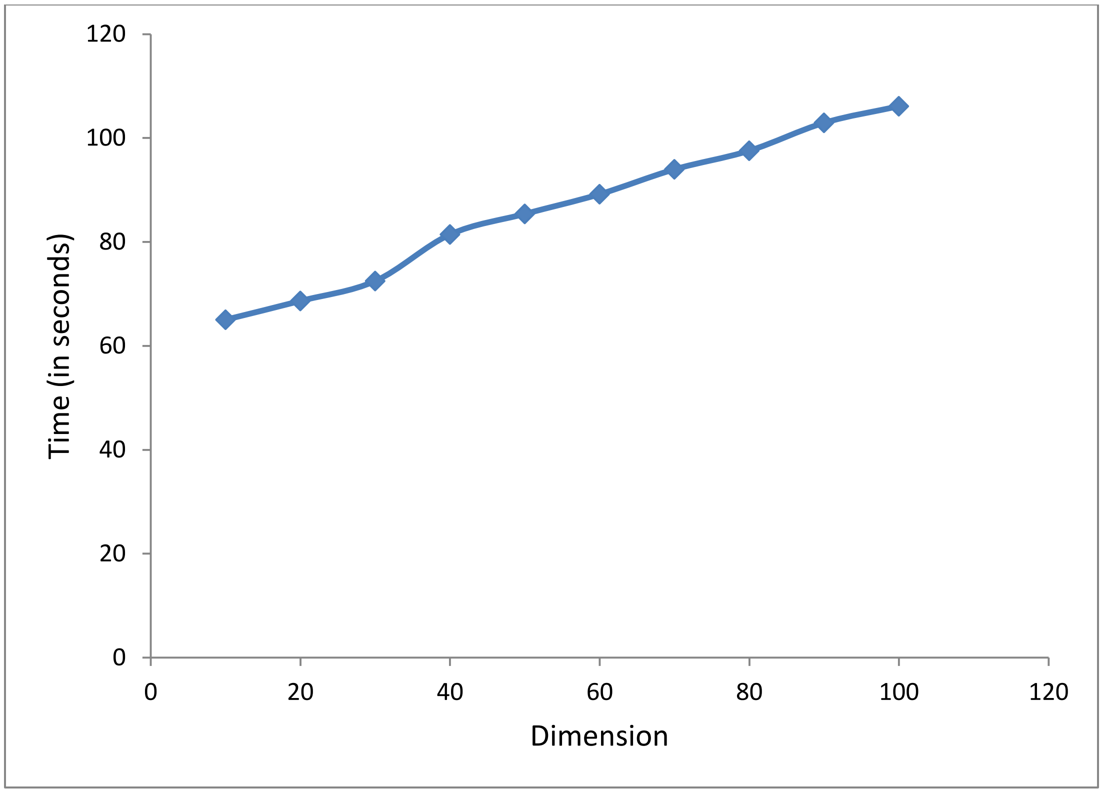


Figure 7: Running time versus varying number of dimensions with number of particles=200

The figure shows the time taken for the parallel PSO code using 8 workers with number of particles=200 with varying Dimensions and maximum number of iterations = 200. There is a gradual increase in the time taken to run the parallel PSO code as observed from Figure 7.

Result: As a result of this experiment there is a steady increase in the time taken similar to the previous experiments but with the number of particles as 200. With dimensions on x-axis and time taken on the y-axis we observe the resultant graph has a gradual slope.

Scaling Experiment 6: Figure 8 shows the time taken to run the parallelized PSO code with varying dimensions (D=10,20,30,40,50,60,70,80,90,100) while using 8 workers, number of particles as 250, and maximum number of iterations as 200.

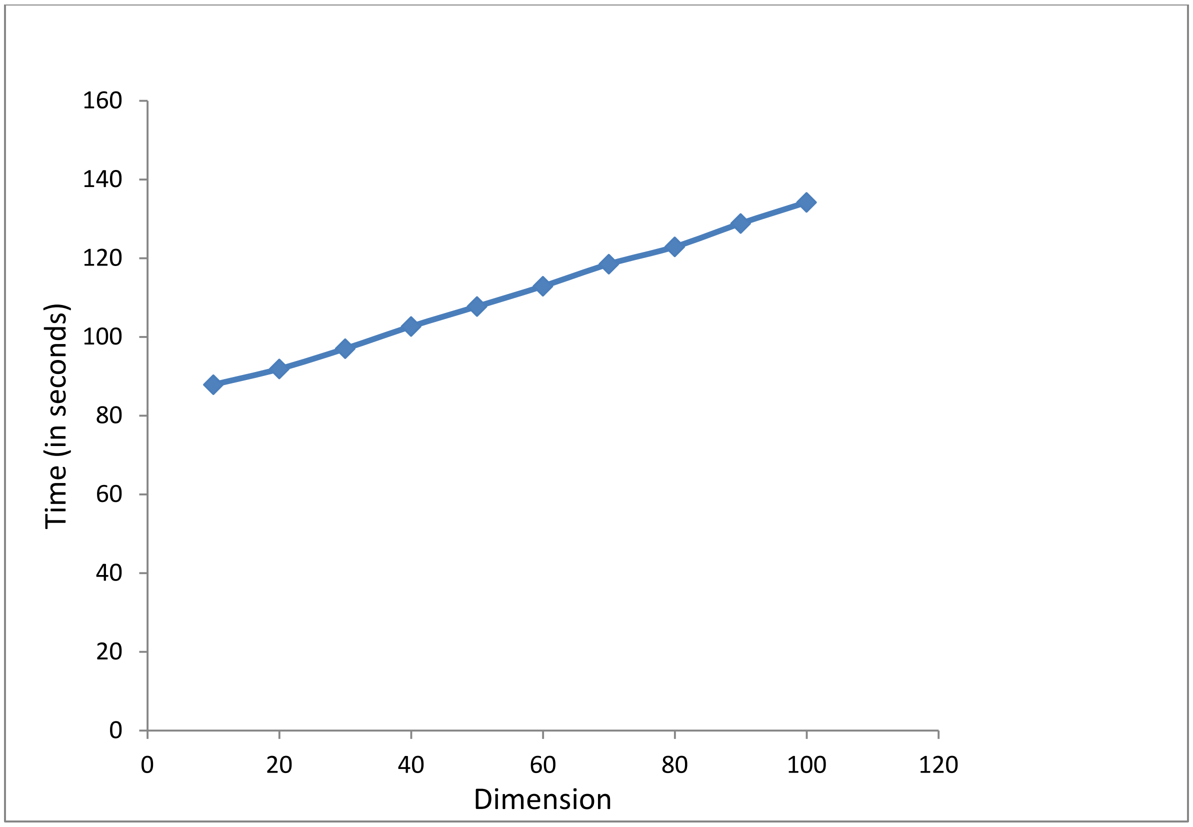


Figure 8: Running time versus number of dimensions with number of particles=250

The figure shows the time taken for the parallel PSO code using 8 workers with varying number of particles=250 with varying dimensions and maximum number of iterations =200.

Result: With this experiment we are trying to demonstrate the behavior of the parallel PSO code when it is run on multiple cores as opposed to a single core. The above graph shows how it responds when thrown on multiple cores in our case 8. It does not show any abnormal behavior and the trend persists as given by the previous experiments.

Scaling Experiment 7: Figure 9 shows the time taken to run the parallelized PSO code with varying dimensions (D=10,20,30,40,50,60,70,80,90,100) while using 8 workers, number of particles as 300, and maximum number of iterations as 200.

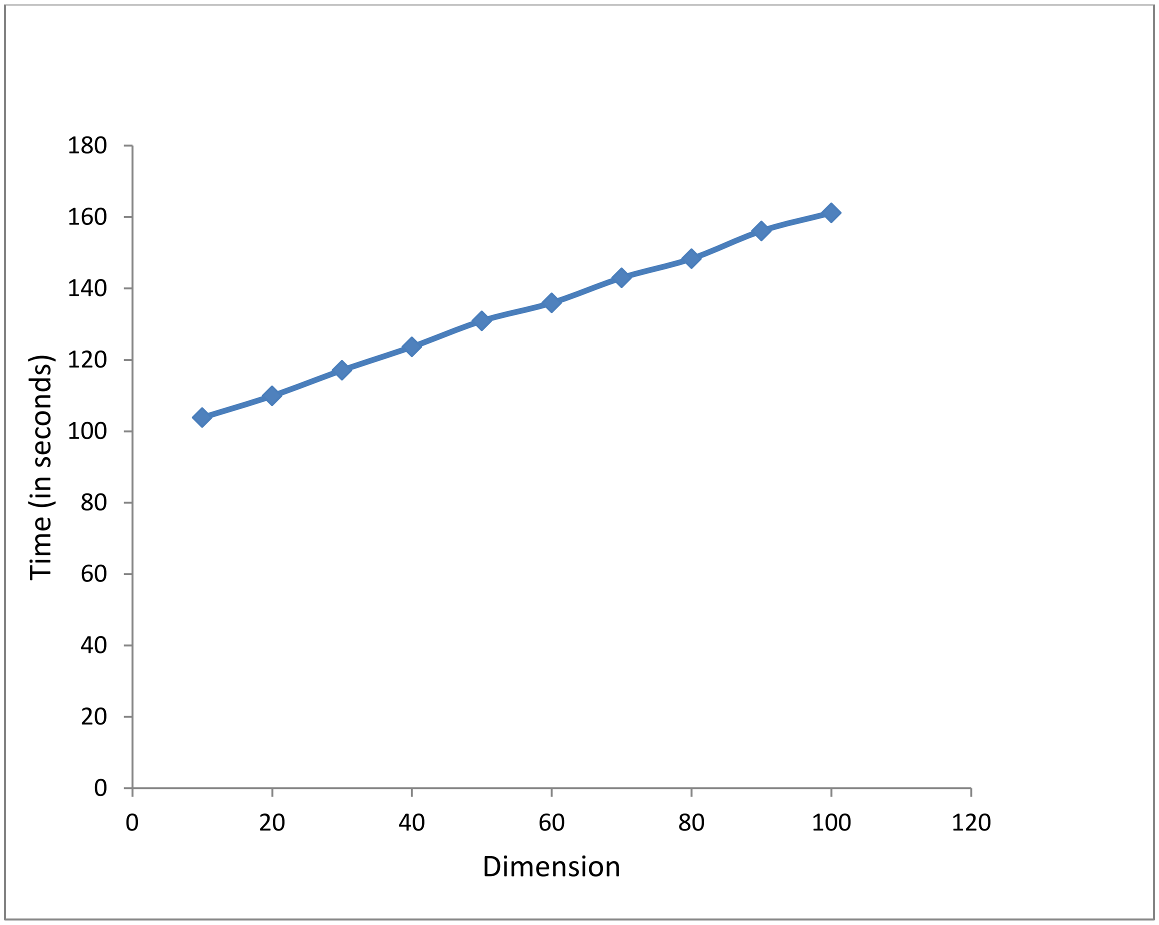


Figure 9: Running time versus number of dimensions with number of particles=300

The figure shows the time taken for the parallel PSO code using 8 workers with number of particles=300 and with varying dimensions, and maximum number of iterations =200.

Result: The result of the experiment described above is a steady increase in the time taken to run the algorithm with the increase in dimensions when the numbers of particles are kept constant between the runs. But, the increase in the time taken is slight between the runs for each dimension value which shows that the increase in the number of Dimensions has a slight increase in the time taken to run.

Scaling Experiment 8: Figure 10 shows the time taken to run the parallelized PSO with varying numbers of particles while using 8 workers (i.e., number of threads), and the maximum number of iterations as 200.

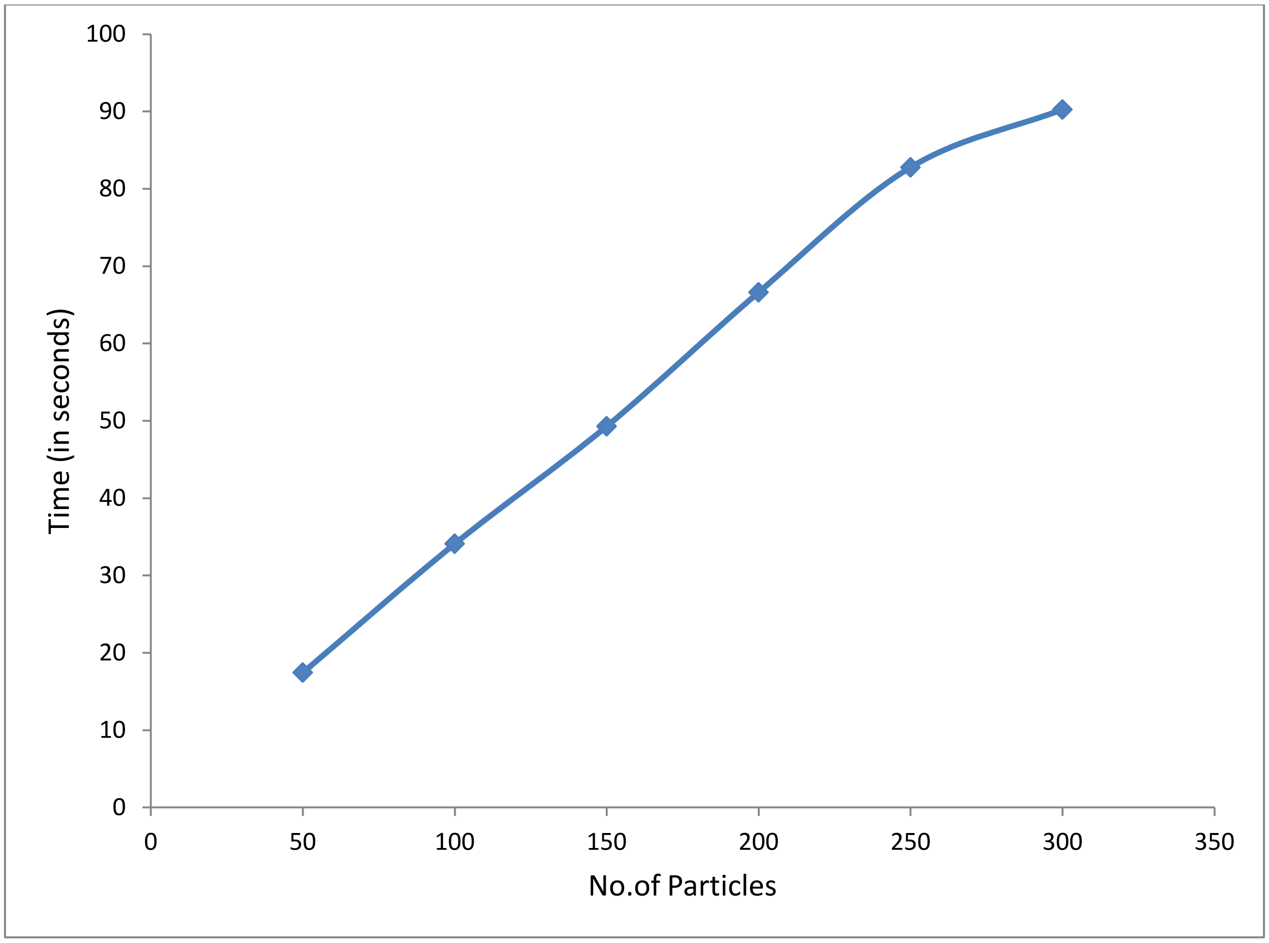


Figure 10: Running time in seconds versus number of particles

This graph shows the time taken for the parallel PSO code using 8 workers with varying number of particles (50,100,150,200,250,300) keeping the dimension d=2 and maximum number of iterations = 200. In this experiment, the dimension of the sample space is kept constant while we observe the behavior of the parallel PSO code when the number of particles in the sample space ranged between 50 and 300.

Result: We observe from the figure above that with the increase in number of particles the time taken to compute the parallel PSO code has a linear increase.

6. CONCLUSION AND FUTURE WORK

With these experiments we have proved that with a very powerful parallel computing toolbox like Matlab, that a search method such as Particle Swarm Optimization can be solved in significantly less amount of time with the help of multi-core processing power computer or a cluster of machines. Using Matlab for the computation of the optimization result of the PSO algorithm has turned out to be efficient. Its parallel computing tool box is easy to use and algorithms can be parallelized with no much of a computing effort.

With parallel PSO, we have seen significant decrease in processing time when compared to serial PSO. Even though, there is communication overhead involved in parallel PSO, it is very negligible as compared to serial PSO and the parallelization of the code to the different cores is taken care of by the Parallel Computing Toolbox in Matlab.

Future work includes applying the parallelization to other optimization techniques. With this I conclude that using Matlab for complex problems on multi core processor will improve the time taken to complete it, while making our job easier.

7. REFERENCES

1. F. C. Anderson, J. Ziegler, M.G. Pandy, and R.T. Whalen. Application of high-performance computing to numerical simulation of human movement. Journal of Biomechanical Engineering 1995; 117:300–308.
2. A. J. van Soest and L. J. R. Casius. The merits of a parallel genetic algorithm in solving hard optimization problems. Journal of Biomechanical Engineering 2003; 125:141–146.
3. B. Monien, F. Ramme, and H. Salmen. A parallel simulated annealing algorithm for generating 3D layouts of undirected graphs. In Franz J. Brandenburg, editor, Proceedings of the 3rd International Symposium of Graph Drawing. Springer-Verlag: Berlin, 1995; 396–408.
4. R. C. Eberhart and Y. Shi. Particle swarm optimization: Developments, applications, and resources. In Proceedings of the 2001 Congress on Evolutionary Computation 2001; 81–86.
5. G. Venter and J. Sobieszczanski-Sobieski. Multidisciplinary optimization of a transport aircraft wing using particle swarm optimization. In 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization 2002, Atlanta, GA.
6. P. C. Fourie and A. A. Groenwold. The particle swarm algorithm in topology optimization. In Proceedings of the Fourth World Congress of Structural and Multidisciplinary Optimization 2001; Dalian, China.
7. R. C. Eberhart and Y. Shi. Particle swarm optimization: developments, applications and resources. In Proceedings of the IEEE Congress on Evolutionary Computation 2001; 27–30.

8. J. B. Park, K. S. Lee, J. R. Shin, and K. Y. Lee, “A particle swarm optimization for economic dispatch with non smooth cost functions,” IEEE Trans. Power Syst., vol. 20, no. 1, pp. 34–42, 2005.

9. J. Robinson and Y. Rahmat-Sammi, “Particle swarm optimization in electromagnetics,” IEEE Trans. Anten. Propag. , vol. 52, no. 2, pp. 397–407, 2004.

10. R. Eberhart and Y. Shi, “Particle swarm optimization: Developments, applications and resources,” in Proc. 2001 IEEE Congr. Evolut. Comput., Seoul, South Korea, 2001, pp. 81–86.

11. J. J. Liang, A. K. Qin, P. N. Suganthan and S. Baskar, "Comprehensive Learning Particle Swarm Optimizer for Global Optimization of Multimodal Functions", IEEE T. on Evolutionary Computation, Vol. 10, No. 3, pp. 281-295, June 2006.

12. S. Z. Zhao and P. N. Suganthan, Q.K Pan, M. Fatih Tasgetiren, “Dynamic Multi-Swarm Particle Swarm Optimizer with Harmony Search”, Expert Systems with Applications. 2011 Apr 30; 38(4):3735-42.