PARALLELIZATION OF GENERIC PSO JAVA CODE USING MPJEXPRESS

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

By

Manoj Babu Madamanchi

In Partial Fulfillment
For the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

December 2012

Fargo, North Dakota

North Dakota State University

Graduate School

Title

Parallelization of Generic PSO Java Code Using MPJExpress

By

Manoj Babu Madamanchi

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

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Simone Ludwig

Advisor

Dr. Kendall Nygard

Dr. Saeed Salem

Dr. María de los Ángeles Alfonseca-Cubero

Approved by Department Chair:

12/21/2012 — Date

Kenneth Magel — Signature

ABSTRACT

Many scientific, engineering and economic problems involve the optimization of a set of parameters. The Particle Swarm Optimization (PSO) is one of the new techniques that have been empirically shown to perform well. The PSO algorithm is a population-based search algorithm based on simulating the social behavior of birds within a flock. 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 parallel algorithms and parallel architectures have drawn lots of attention. Parallelization of PSO has proved to enhance computational throughput and global search capability

In this paper, we detail the parallelization of an increasingly popular global search method, the PSO algorithm using MPJ Express. Both synchronous and asynchronous parallel implementations are investigated. The parallel PSO algorithm's robustness and efficiency are demonstrated by using four standard benchmark functions Alpine, Rosenbrock, Rastrigin and Schaffer.

ACKNOWLEDGEMENTS

I would take this opportunity to thank my advisor Dr. Simone Ludwig, who has given me immense support, motivation and valuable advice that helped me to grow as a researcher and complete my paper. I am thankful to my committee members Dr. Kendall Nygard, Dr. Saeed Salem, Dr. Maria de los Angeles Alfaonseca-Cubero, for their consistent support. I am thankful to my parents and friends for supporting throughout my life time.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

1. INTRODUCTION

Since the 19th century, evolution in optimization theory has been noticed. First in this field was Linear Programming, invented around 1940 [1]. Thereafter, scientists kept inventing new ways to optimize linear and non-linear problems. It is the arrival of Genetic Algorithms, which set the scientific wheel of improvement in motion. In the 1960s and early ‘70s artificial evolution became a widely known optimization method and till now work is going on to improve the existing once. The Particle Swarm Optimization (PSO) algorithm is a recent addition to this list. It is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy [2]. PSO has been successfully applied to large-scale problems in several engineering disciplines. Being a population based approach, it is readily parallelizable.

1.1. Particle Swarm Optimization

The initial intent of the particle swarm concept was to graphically simulate the unpredictable choreography of a bird flock with the aim of discovering patterns that govern the ability of birds to fly synchronously and to suddenly change direction with a regrouping to make an optimal formation. From this initial objective, the concept evolved into a simple and efficient optimization algorithm. In PSO individuals are referred to as particles. Changes to the position of particles within the search space are based on the social-psychological tendency of individuals to emulate the success of other individuals.

The changes to a particle within the swarm are therefore influenced by the experience, or knowledge, of its neighbors. The search behavior of a particle is thus affected by that of the other particles within the swarm [2].

1.1.1. Basic PSO

A PSO algorithm maintains a swarm of particles, where each particle represents a potential solution. A swarm is similar to a population, while a particle is similar to an individual. In simple terms, the particles are “flown” through a multidimensional search space, where the position of each particle is adjusted according to its own experience and that of its neighbors [2]. Initially, particles are distributed throughout the design space and their positions and velocities are modified based on the knowledge of the best solution found thus far by each particle in the ‘swarm’. Attraction towards the best-found solution occurs stochastically and uses dynamically-adjusted particle velocities. Let x_k^i denote the position of particle i in the search space at time step k; where k denotes discrete time steps. The position of the particle is changed by adding a velocity, v_{k+1}^i, to the current position:

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

It is the velocity vector that drives the optimization process, and reflects both the experiential knowledge of the particle and socially exchanged information from the particle’s neighborhood [2]. The experiential knowledge of a particle is generally

referred to as the cognitive component. The socially exchanged information is referred to as the social component of the velocity equation. Originally, two PSO algorithms have been developed which differ in the size of their neighborhoods. These two algorithms are the global best and local best PSO.

1.1.2. Global Best PSO

For the global best PSO, or gbest PSO, the neighborhood for each particle is the entire swarm. The social network employed by the gbest PSO reflects the star topology. For the star neighborhood topology, the social component of the particle velocity update reflects information obtained from all the particles in the swarm. In this case, the social information is the best position found by the swarm, referred to as p_k^g.

For gbest PSO, the velocity of particle i is calculated as

$$v_{k+1}^i = w_k v_k^i + c_1 r_1 [p_k^i - x_k^i] + c_2 r_2 [p_k^g - x_k^i] \quad (2)$$

Here, subscript k indicates a (unit) pseudo-time increment, p_k^i represents the best ever position of particle i at time k and p_k^g represents the global best position in the swarm at time k. c_1 and c_2 are positive acceleration constants used to scale the contribution of the cognitive and social components respectively and r_1, r_2 are random values in the range [0, 1], sampled from a uniform distribution. These random values introduce a stochastic element to the algorithm. The notation w_k denotes the inertia of

the particle. The personal best position p_k^i associated with particle i is the best position the particle has visited since the first time step. The pseudo code of the gbest PSO is shown below:

1. Create and initialize an n_x-dimensional swarm;
2. repeat
3. for each particle i = 1, . . . , n //set the personal best position
4. if $p_k^i < p_{best}^i$ then
5. $p_{best}^i = p_k^i$
6. end //set the global best position
7. if $p_k^i < p_{best}^g$ then
8. $p_{best}^g = p_k^i$
9. end
10. end
11. for each particle i = 1, . . . , n do
12. update velocity
13. update position
14. end
15. until stopping condition is true

1.1.3. Local Best PSO

The local best PSO, or lbest PSO, uses a ring social network topology where smaller neighborhoods are defined for each particle. The social component reflects information exchanged within the neighborhood of the particle, reflecting local knowledge of the environment [3]. With reference to the velocity equation, the social contribution to particle velocity is proportional to the distance between a particle and the best position found by the neighborhood of particles. The velocity is calculated as:

$$v_{k+1}^i = w_k v_k^i + c_1 r_1 [p_k^i - x_k^i] + c_2 r_2 [p_k^l - x_k^i] \quad (3)$$

where p_k^l is the best position found by the neighborhood of particle i at time k. The pseudo code of the lbest PSO is shown below.

1. Create and initialize an n_x-dimensional swarm;
2. repeat
3. for each particle i = 1, . . . , n do //set the personal best position
4. if $p_k^i < p_{best}^i$ then
5. $p_{best}^i = p_k^i$
6. end //set the neighborhood best position
7. if $p_k^i < p_{best}^l$ then
8. $p_{best}^l = p_k^i$
9. end
10. end
11. for each particle i = 1, . . . , n do
12. update the velocity
13. update the position
14. end
15. until stopping condition is true;

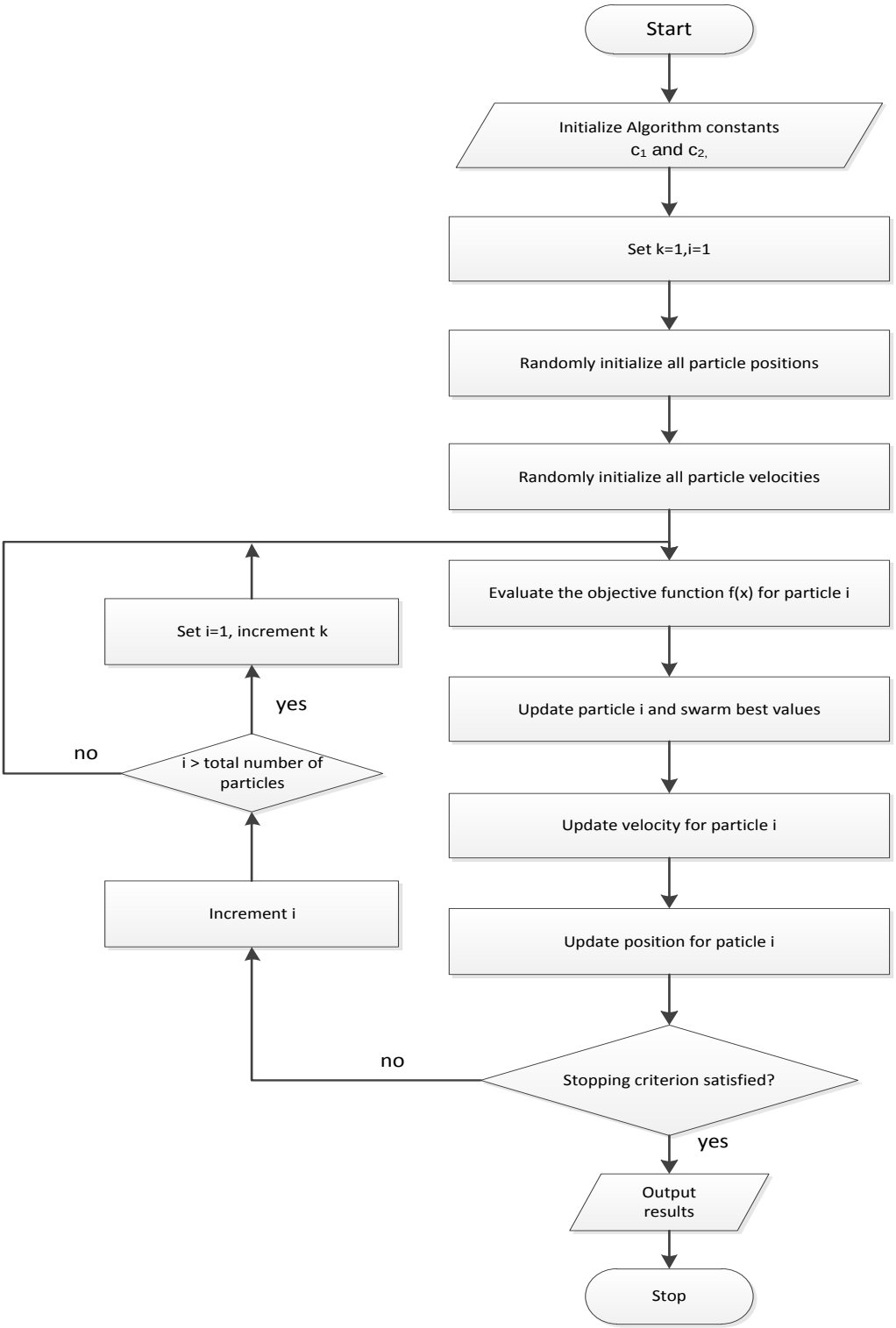

Figure 1.1: Serial implementation of PSO algorithm

2. RELATED WORK

The task of optimization involves determining the measure of optimality, subject to certain constraints. This task of optimization is of great importance to many professions, since everyone is interested in finding the optimal value subject to constraints, e.g. cost and utilization. Scientists require optimization techniques when performing model fitting. Economists and operation researchers have to consider the optimal allocation of resources. Thus, optimization is of atmost importance in every field. The term optimization refers to either minimization or maximization. Maximization of the function f is equivalent to minimizing the function −f, therefore the terms minimization, maximization and optimization are interchangeable. Of all the available optimization methods, PSO is very helpful when the search space is too large to search exhaustively. Moreover, PSO is a stochastic optimization method often used to solve complex engineering problems such as structural and biomechanical optimizations [2]. The performance of serial PSO is affected when complex engineering optimizations are considered; this motivated the development of parallel optimization. Parallel optimization can either be synchronous or asynchronous depending on the particle update factor. Parallel optimization algorithms employing synchronous and asynchronous approaches are a potential solution to the load imbalance problem [2].

Several authors have performed analysis to prove the performance of distributed PSO. Byung and Alan have performed analysis to determine the parallel performance

[5]. They evaluated four analytical test problems and a biomechanical test problem. They have calculated the parallel performance of test cases on both homogeneous and heterogeneous Linux clusters.

In the asynchronous case they considered one of the processors as master and the remaining as slaves. The master processor is used to initialize all the optimization parameters and to update particle positions. The parallel asynchronous case uses First-In-First-Out (FIFO) to evaluate particles. Communication between master and slave processors is achieved using a point-to-point communication scheme implemented with the Message Passing Interface [5].

The analytical test problems that are evaluated are Schaffer, Corona, Griewank and simple 2-dimensional function. Every test problem has several local maxima and global maxima. On an average they used 15000 function evaluations and 0.001 as acceptable error. For Corona they considered design variables of 4, 8, 16. For Griewank they had considered 32 and 64 design variables [5]. For the Biomechanical test case they want to determine patient-specific parameter values that permit a three-dimensional kinematic ankle joint model to reproduce experimental movement data as closely as possible.

The evaluation metrics such as performance, robustness and parallel efficiency of PAPSO (Parallel Asynchronous PSO) is compared to that of PSPSO (Parallel Synchronous PSO). They considered a homogeneous Linux cluster of 20 identical

machines where each machine possessed a 1.33 GHz Athlon processor and 256MB of memory. The second was a group of 20 heterogeneous machines chosen from several Linux clusters. Convergence speed for both algorithms was statistically the same on all problems except Schaffer, where PSPSO performed slightly better. PAPSO parallel efficiency was worse than that of PSPSO for small numbers of processors but generally better for large numbers of processors [2]. For the biomechanical test problem, the total execution time decreased and the speedup increased for both algorithms as the number of processors increased. Communication overhead for the PAPSO algorithm was smaller than for the PSPSO algorithm. Finally, they concluded saying the PAPSO algorithm exhibits good parallel performance for large numbers of processors as well as good optimization robustness and performance.

3. APPROACHES

3.1. Message Passing Java

The Message Passing Interface (MPI) was introduced in June 1994 as a standard message passing API for parallel scientific computing [6]. Message Passing Java (MPJ) is being developed as a middleware between the user program and communications protocols [7]. MPJ Express is a message-passing library that can be used by application developers to execute their parallel Java applications on compute clusters or network of computers. Compute clusters is a popular parallel platform, which is extensively used by the High Performance Computing (HPC) community for large-scale computational work. MPJ Express is essentially a middleware that supports communication between individual processors of clusters. The programming model followed by MPJ Express is Single Program Multiple Data (SPMD). Although MPJ Express is designed for distributed memory machines like network of computers or clusters, it is possible to efficiently execute parallel user applications on desktops or laptops that contain shared memory or multicore processors. The MPJ Express software can be configured in two ways. The first configuration, known as the Multicore Configuration, is used to execute MPJ Express user programs on laptops and desktops. The second configuration, known as the Cluster Configuration, is used to execute MPJ Express user programs on clusters or network of computers.

MPJ provides initialization and finalization methods for the network connections. JavaComm and GridComm hold Java sockets and GridTcp sockets, respectively. These two classes do not provide any major functionality other than maintaining the input and output streams of their respective sockets. Communicator is the class that provides the primary communications capabilities of MPJ. Communicator contains the major functions like Send(), Recv(), Barrier(). MPJMessage is a wrapper around each message received by the Recv() functions. It holds the message's status and the actual message itself. The various Datatype subclasses provide serialization and deserialization of their respective types for Communicator.

3.1.1. Configurations

The MPJ Express software can be configured to work on clusters (network of computers) or on laptops/desktops (multicore processors).

3.1.2. Multicore Configuration

The multicore configuration is meant for users who plan to write and execute parallel Java applications using MPJ Express on their desktops or laptops—typically such hardware contains shared memory and multicore processors. In this configuration, users can write their message passing parallel application using MPJ Express and it will be ported automatically on multicore processors. Users can first develop applications on their laptops and desktops using multicore configuration, and then take the same code

to distributed memory platforms including clusters. It might be noted that user applications stay the same when executing the code in multicore or cluster configuration. Under the hoods, the MPJ Express library starts a single thread to represent MPI process. The multicore communication device uses efficient inter-thread mechanism.

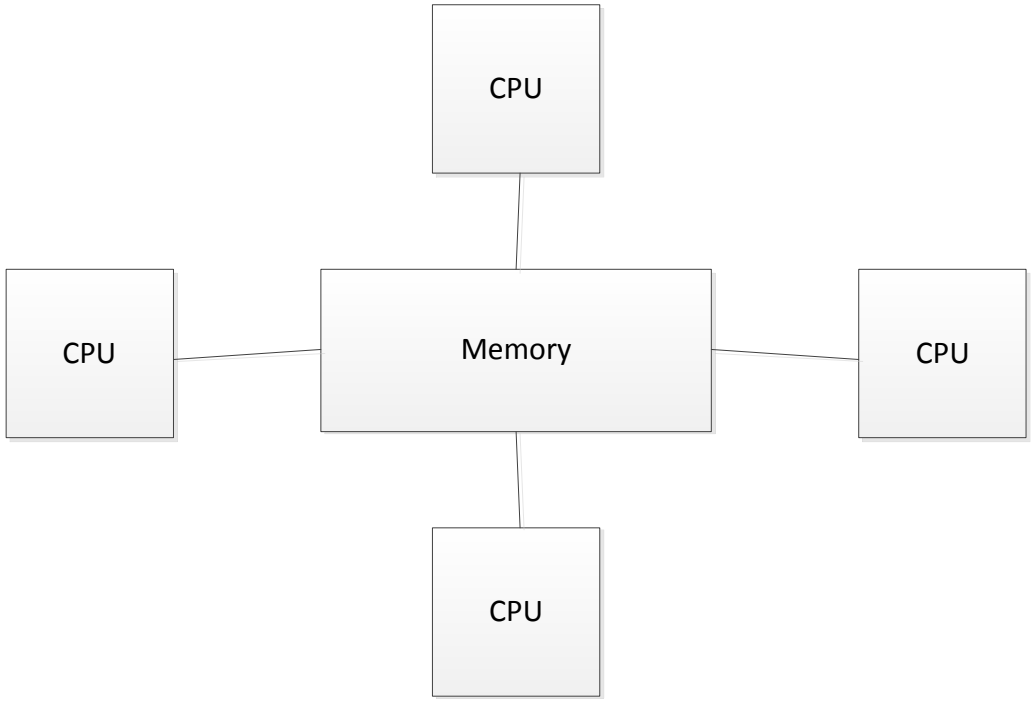

Figure 3.1: MPJExpress multicore configuration

3.1.3. Cluster Configuration

The cluster configuration is meant for users who plan to execute their parallel Java applications on distributed memory platforms including clusters or network of

computers. Application developers can opt to use either of the two communication devices in the cluster configuration: the communication devices including Java New I/O (NIO) device or Myrinet device.

The Java NIO device driver (also known as niodev) can be used to execute MPJ Express programs on clusters or network of computers. The niodev device driver uses Ethernet-based interconnect for message passing. On the other hand, many clusters today are equipped with high-performance low-latency networks like Myrinet. MPJ Express also provides a communication device for message passing using Myrinet interconnect—this device is known as mxdev and is implemented using the Myrinet eXpress (MX) library by Myricom [3]. These communication drivers can be selected using command line switches. As an example, consider a cluster or network of computers shown in Figure 3.3 that shows eight compute nodes connected to each other via private interconnect. The MPJ Express cluster configuration will start one MPJ Express process per node, which communicates to each other using message passing.

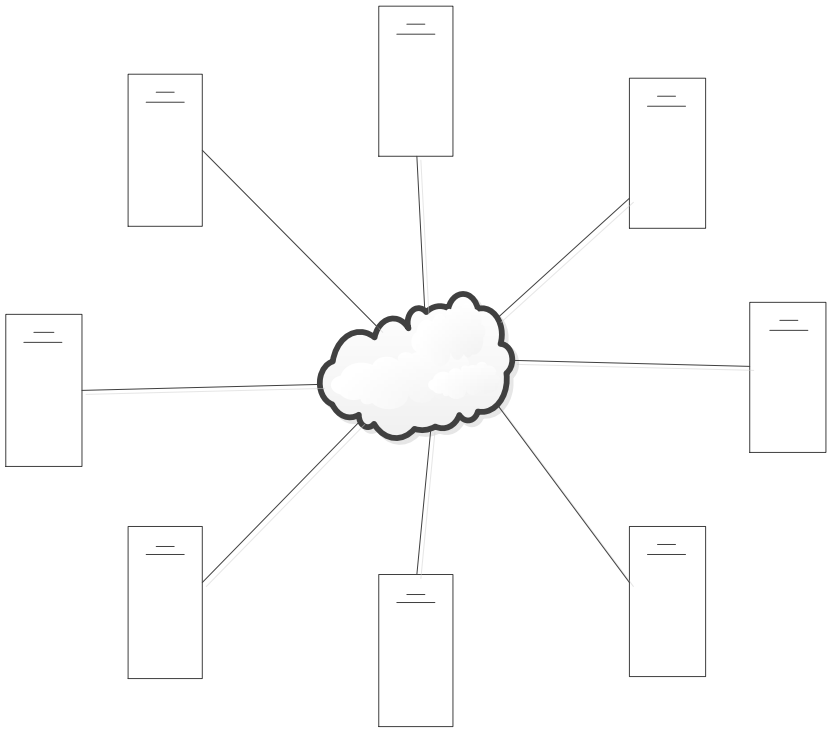

Figure 3.2: MPJExpress cluster configuration

3.2. MPJ Methods

3.2.1. MPJ.Init (String[ ] args) for Java

The Init() function establishes all to all connections using Java sockets. First, MPJ receives initialization commands such as rank, slave, master node, number of processes, etc. Such arguments are mainly used to identify each process [3]. On the master process, Init() creates a ServerSocket and accepts connections from slaves until the number of connections equals the number of processors. Following each

connection, the master process reads the connecting process's rank and identifies each connection with that rank, storing the rank-connection information in JavaComm. Then, the master broadcasts the ranks and their corresponding hostname to all slave processes. At this point, the master process's Init() is complete.

For the slave processes, they first connect to the master process, send their rank, and then receive a table with other slaves' ranks and their hostnames. Once such information is exchanged with the master, the slaves will connect with each other. First, all of the slaves except for the highest-ranking slave will create Server Sockets. Afterwards, the lowest ranking slave will accept connections from higher-ranking slaves. The lowest ranking slave will then receive a rank from the connection it received and update its rank connection table (much like the master). When the lowest ranking slave has received all connections, its Init() is complete. The second lowest ranking slave then accepts connections from the higher ranking slaves, and the process repeats until the second highest ranking slave has accepted a connection from the highest ranking slave, indicating all slaves are connected to all other slaves. At this point, the Init() process is complete for Java sockets.

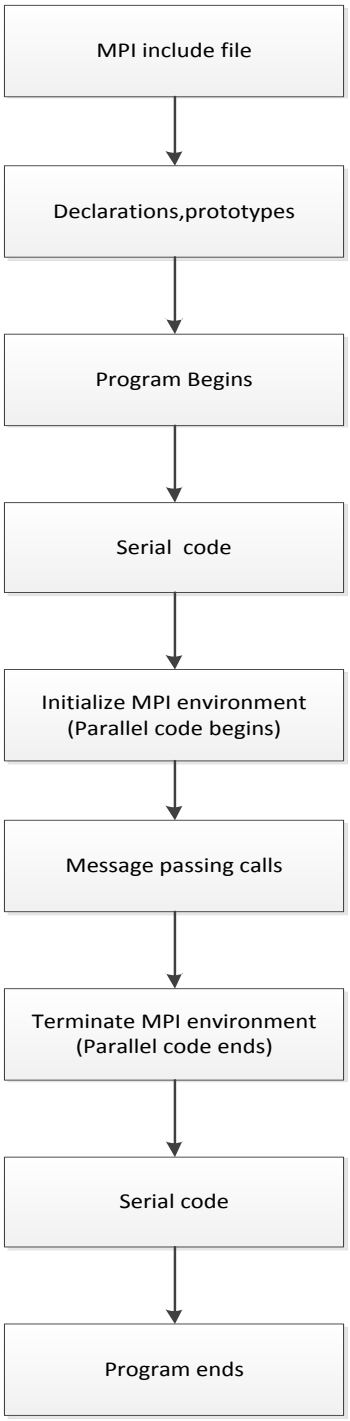

Figure 3.3: MPJ flow chart

3.2.2. Communicator.Send()

Communicator.Send(Object[ ] buf, int offset, int count, Datatype type, int dest, int tag)

Buf	send buffer array
Offset	initial offset in send buffer
Count	number of items to send
Datatype	datatype of each item in send buffer
Dest	rank of destination
Tag	message tag

The Ssend() function takes in various parameters describing the datatype, the send count, the send buffer, the offset, the destination rank, and the message tag. The send buffer must be an array. The datatype is actually a Datatype object from MPJ. Before sending a message, the Ssend() function first creates a header for the message, including the message's type, size in bytes, count, and the tag. Ssend() then serializes the message using the Datatype specified in the function parameters, and writes the header along with the serialized message to the output stream corresponding to the destination rank.

3.2.3. Communicator.Recv()

Communicator.Recv(Object[ ] buf, int offset, int count, Datatype type, int src, int tag)

Buf	receive buffer array
Offset	initial offset in receive buffer
Count	number of items in receive buffer
Datatype	datatype of each item in receive buffer
Source	rank of source
Tag	message tag

When a user calls the Recv() function, Recv will perform a blocking read operation on the input stream corresponding to the source rank. First, Recv() reads 16 bytes of the message header and then reads the rest of the body with respect to the size defined in the header. Then, Recv() will deserialize the message using the correct Datatype. The completed message is stored in an MPJMessage, which is then checked against the parameters of Recv(). If the tag or datatype does not match, the message is stored in a message queue, and Recv() will read again for a new message. If MPJ.ANY_SOURCE is specified, Recv() will poll each socket and read from the first socket with available data, and then check the tag. If MPJ.ANY_TAG is specified, then Recv() will return the message if the data type matches the parameter. Recv() will crash if the count parameter is smaller than the actual message's count.

3.2.4. Barrier()

Barrier() is simple in that rank 0 will receive a message from all other ranks, and broadcast another message once it has received from all other ranks. Thus, each process is blocked until all processes have called Barrier().

3.2.5. Isend()

Isend() spawns a thread to send a message, like send(). It is generally used for asynchronous communication, i.e non-blocking communication.

public Request Isend(Object buf, int offset, int count, Datatype type, int dest, int tag)

Buf	send buffer array
Offset	initial offset in send buffer
Count	number of items to send
Datatype	datatype of each item in send buffer
Dest	rank of destination
Tag	message tag

3.2.6. Irecv()

Irecv() spawns a thread to receive a message, like recv(). It is generally used for asynchronous communication.

public Request IRecv(Object buf, int offset, int count, Datatype type, int src, int tag)

Buf	receive buffer array
Offset	initial offset in receive buffer
Count	number if items in receive buffer
Datatype	datatype of each item in receive buffer

Source	rank of source
Tag	message tag

3.3. Synchronous and Asynchronous

Presently, day large-scale engineering optimization problems impose large computational demands, resulting in high-end processors. To obtain enhanced computational throughput and global search capability parallel algorithms and parallel architectures have drawn lots of attentions. PSO, like most stochastic optimization algorithms, can be easily implemented in a parallel distributed computing environment. The simplest and most common distributed PSO algorithms utilize a master-slave architecture, where a single controlling (master) processor runs only the optimization algorithm, and utilizes on external (slave) processors to compute potential solutions.

Parallelization should have no adverse effect on algorithm operation. Calculations sensitive to program order should appear to have occurred in exactly the same order as in the original formulation. In the serial PSO algorithm, the fitness evaluations form the bulk of the computational effort for the optimization and can be performed independently. For the parallel implementation, we therefore chose to decompose the algorithm to perform the fitness evaluations concurrently on a parallel machine. The pseudo-code for a multiple iterations of sequential synchronous PSO algorithm is shown below:

1. Initialize optimization
2. Initialize algorithm constants
3. Randomly initialize all particle positions and velocities
4. Perform optimization
5. For k=1 number of iteration
6. For i = 1, number of particles
7. Evaluate fitness function f(x)
8. End
9. Check convergence
10. Update p^{g}_{best}, p^{i}_{best}, and particle positions and velocities x^{i}_{k}, v^{i}_{k}
11. End
12. Report Results

In a parallel computational environment, the main performance bottleneck is the communication latency between processors. This is especially true for clusters of computers where the use of high performance network interfaces are limited due to their high cost.

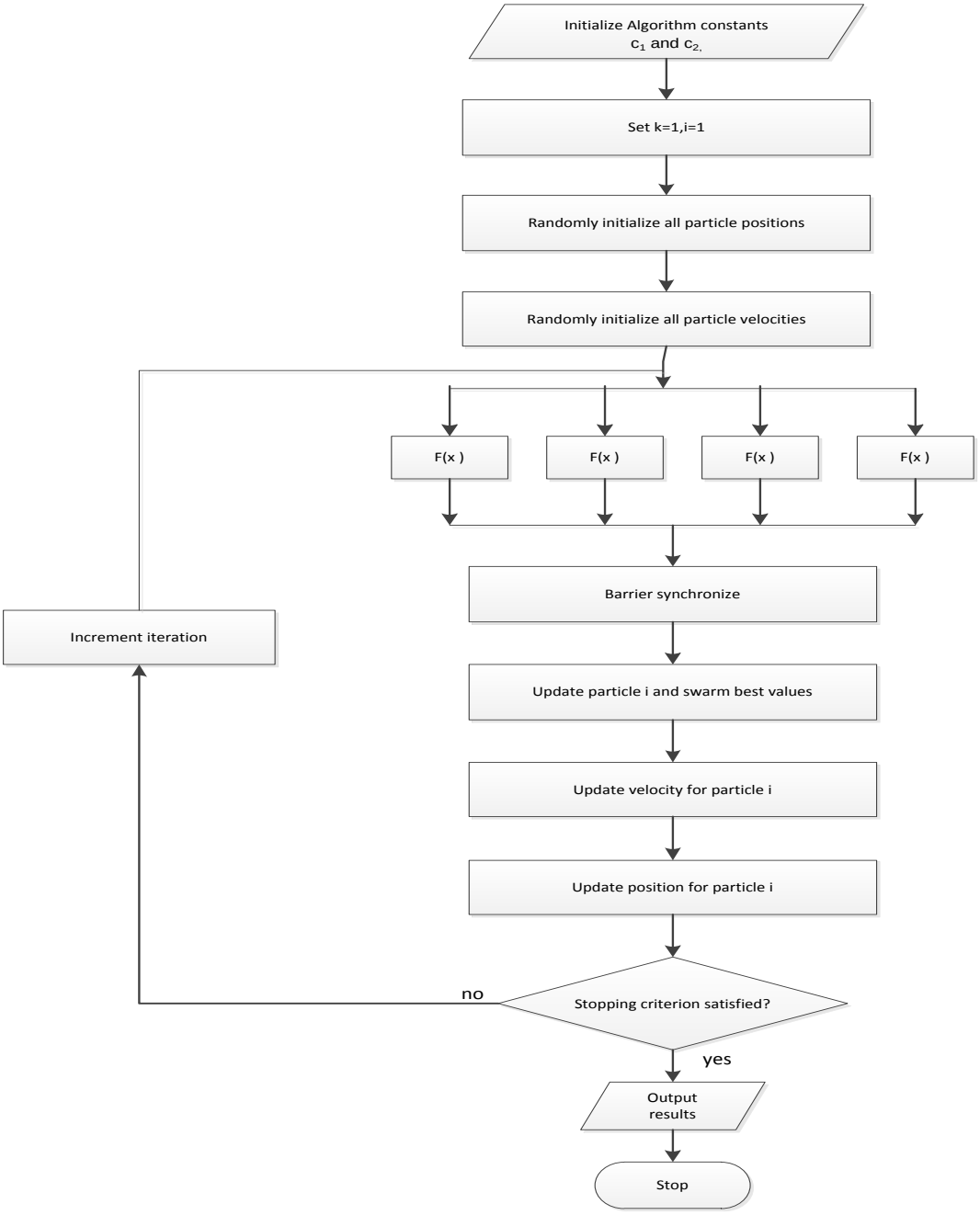

Figure 3.4: Parallel implementation of PSO algorithm

For the parallel implementation algorithm, synchronization is required to ensure that all of the particle fitness evaluations have been completed and results reported before the velocity and position calculations can be executed [1]. This is done by using a global synchronization or barrier function in the MPI communication library, which temporarily stops the coordinating node from proceeding with the next swarm iteration until all of the computational nodes have responded with a fitness value. This, however implies that the time required for a single parallel swarm fitness evaluation will be dictated by the slowest fitness evaluation in the swarm [1]. The majority of parallel particle swarm implementations are based on the synchronous model, where the optimization algorithm waits at the end of every iteration until all particle solutions have been returned before updating particle velocities and positions (known as a Parallel Synchronous Particle Swarm Optimization algorithm, or PSPSO). This approach can work quite well provided a number of conditions are met:

1) The optimization algorithm has uninterrupted access to a homogeneous computer cluster.

2) The fitness function can be evaluated in roughly constant time, regardless of the input parameters.

3) The number of particles in the swarm can be evenly divided by the number of available nodes.

However, it is often difficult (or even impossible) to obtain these ideal operating conditions, and when this happens the parallel efficiency of the algorithm drops. This drop in parallel efficiency can be mitigated somewhat through the use of asynchronous updates. In a Parallel Asynchronous Particle Swarm Optimization (PAPSO) algorithm, the algorithm does not wait for all solutions to be returned before updating the velocity and position of solved particles and re-evaluating them. These algorithms can display varying degrees of synchronous behavior, ranging from 1% synchronous (or pure asynchronous) where particles are updated and submitted for re-evaluation as soon as they are solved, to 100%, at which point the algorithm becomes fully synchronous once more, waiting for all particles to be returned before updates are carried out [5]. Parallel optimization algorithms employing an asynchronous approach are a potential solution to the load imbalance problem. The pseudo-code for multiple iterations of the sequential synchronous PSO algorithm is shown below:

1. Initialize optimization
2. Initialize algorithm constants
3. Randomly initialize all particle positions and velocities
4. Perform optimization
5. For k = 1, number of iterations
6. For i = 1, number of particles
7. Evaluate fitness function f(x)
8. End
9. Check convergence
10. Update p^{g}_{best}, p^{i}_{best}, and particle positions and velocities x^{i}_{k}, v^{i}_{k}
11. End
12. Report Results

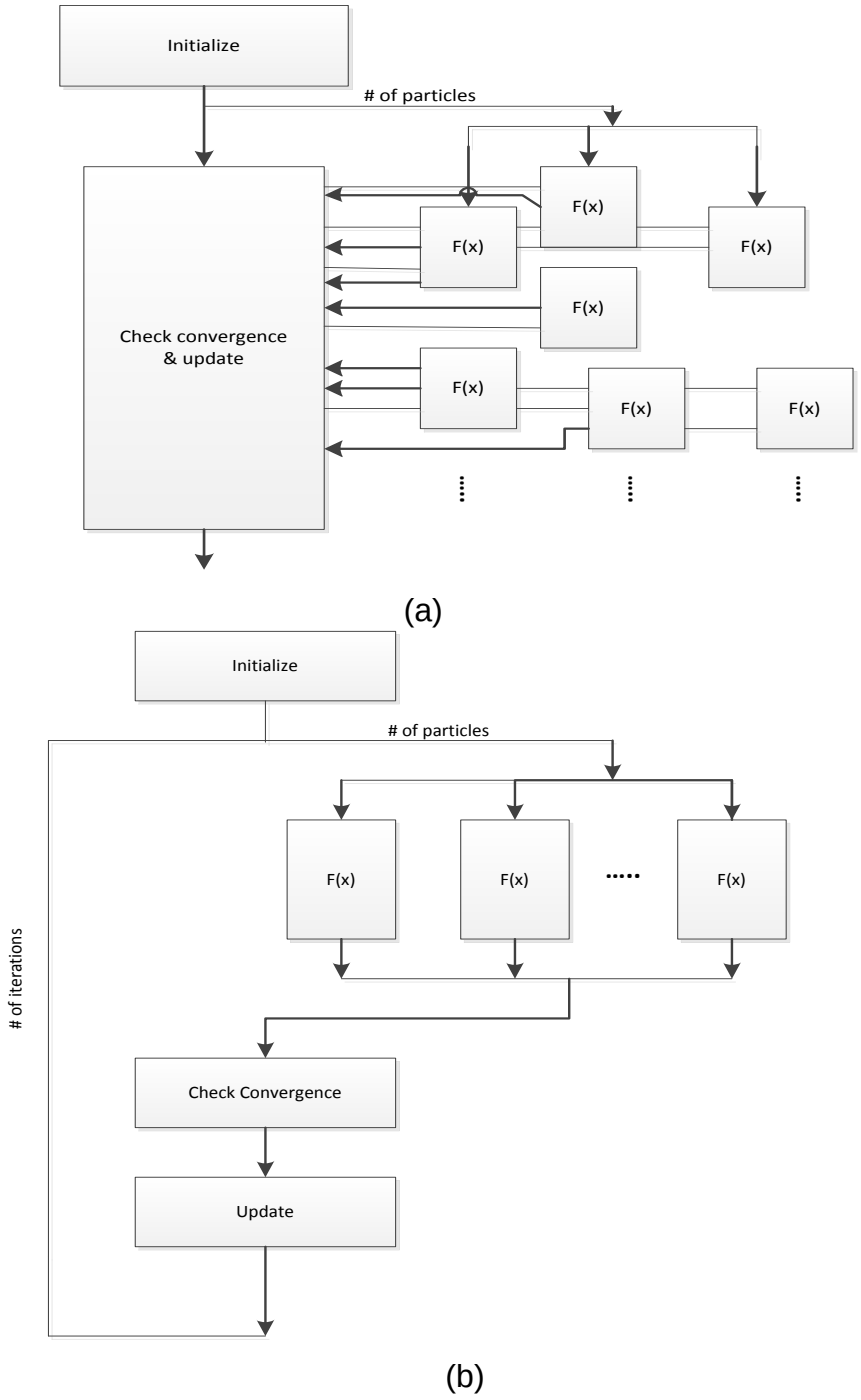

Figure 3.5: Block diagrams: (a) parallel asynchronous; (b) parallel synchronous PSO

4. NEIGHBORHOOD STRUCTURES

The feature that drives PSO is social interaction. Particles within the swarm learn from each other and, on the basis of the knowledge obtained, move to become more similar to their “better” neighbors. The social structure for PSO is determined by the formation of overlapping neighborhoods, where particles within a neighborhood influence one another [7]. This is in analogy with observations of animal behavior, where an organism is most likely to be influenced by others in its neighborhood, and where organisms that are more successful will have a greater influence on members of the neighborhood than the less successful ones. Within the PSO, particles in the same neighborhood communicate with one another by exchanging information about the success of each particle in that neighborhood. All particles then move towards some quantification of what is believed to be a better position. The performance of the PSO depends strongly on the structure of the social network. The flow of information through a social network depends on the degree of connectivity among nodes (members) of the network and the average shortest distance from one node to another. With a highly connected social network, most of the individuals can communicate with one another, with the consequence that information about the perceived best member quickly filters through the social network. In terms of optimization, this means faster convergence to a solution than for less connected networks. However, for highly connected networks, the faster convergence comes at the price of susceptibility to local minima, mainly due to the fact that the extent of coverage in the search space is less than for less connected

social networks. For sparsely connected networks with a large amount of clustering in neighborhoods, it can also happen that the search space is not covered sufficiently to obtain the best possible solutions. Each cluster contains individuals in a tight neighborhood covering only a part of the search space. Within these network structures there usually exist a few clusters, with a low connectivity between clusters. Consequently information on only a limited part of the search space is shared with a slow flow of information between clusters. Selection of neighborhood can be done in several ways. Some of them include particle indices, spatial similarity and Euclidian distance. Different social network structures have been developed for PSO and empirically studied.

4.1. Star Social Structure

The star social structure, where all particles are interconnected, is illustrated in Figure 4.1. Each particle can therefore communicate with every other particle. In this case each particle is attracted towards the best solution found by the entire swarm [2]. Each particle therefore imitates the overall best solution. The first implementation of the PSO used a star network structure, with the resulting algorithm generally being referred to as the gbest PSO. The gbest PSO has been shown to converge faster than other network structures, but with a susceptibility to be trapped in local minima.

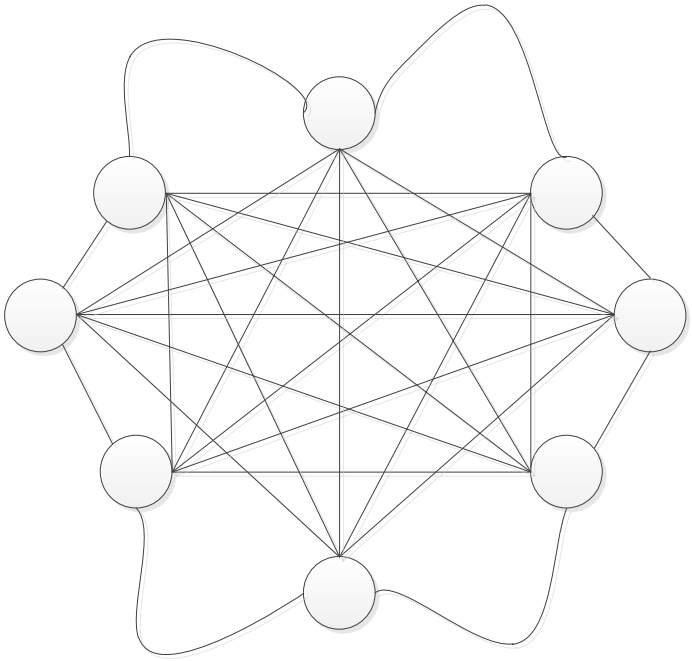

Figure 4.1: Star social structure

4.2. Ring Social Structure

In the ring social structure, each particle communicates with its n immediate neighbors as illustrated in Figure 4.2. Each particle attempts to imitate its best neighbor by moving closer to the best solution found within the neighborhood. Neighborhoods may also overlap, which facilitate the exchange of information between neighborhoods and, in the end, convergence to a single solution. Since information flows at a slower rate through the social network, convergence is slower, but larger parts of the search space are covered compared to the star structure. This behavior allows the ring structure to provide better performance in terms of the quality of solutions found for

multi-modal problems than the star structure. The resulting PSO algorithm is generally referred to as the lbest PSO.

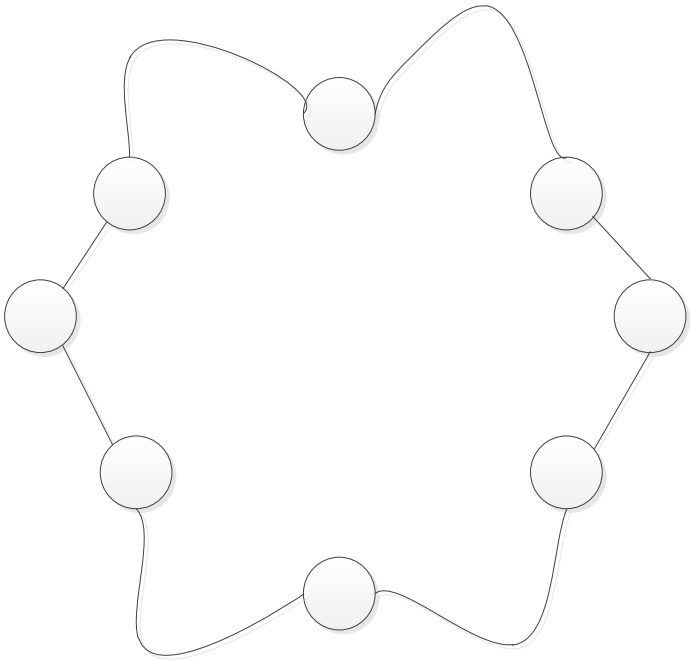

Figure 4.2: Ring social structure

4.3. Wheel Social Structure

In the wheel social structure, the individuals in a neighborhood are isolated from one another as shown in Figure 4.3. One particle serves as the focal point, and all information is communicated through the focal particle. The focal particle compares the performances of all particles in the neighborhood, and adjusts its position towards the best neighbor. If the new position of the focal particle results in better performance, then

the improvement is communicated to all the members of the neighborhood. The wheel social network slows down the propagation of good solutions through the swarm.

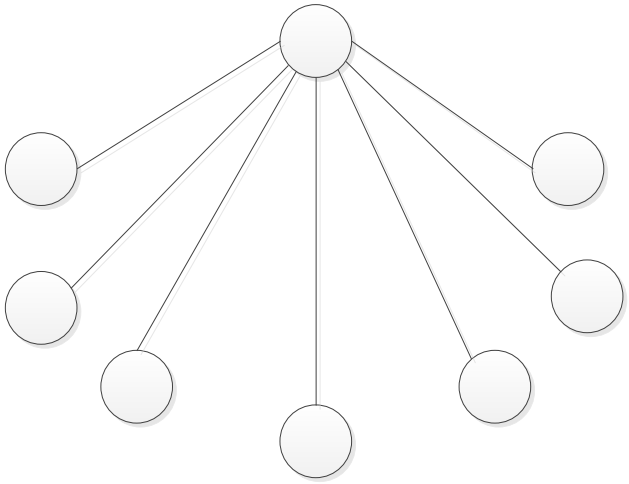

Figure 4.3: Wheel social structure

5. BENCHMARK PROBLEMS

In the field of optimization, it is common to compare algorithms using a large test set, where the tests involve function optimization. In this paper, all the different PSO algorithms are tested based on the four benchmark functions described below. We evaluated the performance, both in terms of the optimum solution and the rate of convergence to the optimum solution.

5.1. Alpine (F1)

A nonseparable function f(x) is called m-nonseparable function if at most m of its parameters x_i are not independent. A nonseparable function f(x) is called fully-nonseparable function if any two of its parameters x_i are not independent [12].

$$F(x) = \sum_{i=1}^{D}[(x_i \sin x_i + 0.1x_i) \quad (4)$$

where D ≥2 is the dimension and $x = (x_1, x_2, \ldots, x_d)$ is a D-dimensional row vector (i.e., a 1×D matrix). Test area is usually restricted to $-10 \leq x_i \leq 10$, i=1,…,n. Its global minimum F(x)=0 is obtainable for x_i, i=1,…,n.

5.2. Rosenbrock (F2)

Rosenbrock's function is also naturally non-separable and is a classic optimization problem, also known as banana function or the second function of De Jong [10]. The function has the following definition:

$$F(x) = \sum_{i=1}^{D-1}[(100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2)] \quad (5)$$

where D ≥2 is the dimension and x = (x_1, x_2, …, x_d) is a D-dimensional row vector (i.e., a 1×D matrix). Test area is usually restricted to a hypercube $-2.048 \leq x_i \leq 2.048$, i=1,…,n. Its global minimum F(x)=0 is obtainable for x_i, i=1,…,n.

5.3. Rastrigin (F3)

Rastrigin function is based on the function of De Jong with the addition of Cosine modulation in order to produce frequent local minima and is defined as follows:

$$F(x) = \sum_{i=1}^{D}[(x_i^2 - 10\cos(2\pi x_i) + 10] \quad (6)$$

where D is the dimension and x = (x_1, x_2, … , x_d) is a D-dimensional row vector (i.e., a 1×D matrix). Similarly, to make it nonseparable, an orthogonal matrix is also used for coordinate rotation. Rastrigin's function is a classical multimodal problem. It is difficult since the number of local optima grows exponentially with the increase of dimensionality [10]. Test area is usually restricted to a hypercube $-5.12 \leq x_i \leq 5.12$, i=1,…,n. Its global minimum F(x)=0 is obtainable for x_i=0, i= 1,…,n.

5.4. Schaffer (F4)

Schaffer function is also naturally nonseparable [12]. Schaffer is defined as follows:

$$F(X) = 0.5 + \frac{\{\mathrm{Sin}\left(\sqrt{x_1^2+x_2^2}\right)\}^2 - 0.5}{\{1.0+0.001\left(x_1^2+x_2^2\right)\}^2} \quad (7)$$

Test area is usually restricted to a hypercube $-100 \leq x_i \leq 100$, i=1,…,n. Its global minimum F(x)=0 is obtainable for $x_i=0$, i = 1,…,n.

6. EXPERIMENTS AND RESULTS

6.1. Experimental Setup

As mentioned in the configuration section, the cluster configuration is used to execute parallel Java applications on distributed memory platforms including clusters or network of computers. To set up the cluster network communication, the Java New I/O (NIO) device named niodev has been used. The niodev device driver uses Ethernet-based interconnect for message passing. The MPJ cluster is a 15 node cluster where every node is a single-core virtual machine with 512MB of RAM per node. MPJExpress version 0.38 has been used for the parallelization process.

The MPJ Express cluster configuration will start one MPJ Express process per node, which communicates to each other using message passing. The cluster set up has been designed to follow a master/slave paradigm. The master processor holds the queue of particles ready to be sent to the slave processors and performs all decision-making processes such as velocity/position updates and convergence checks [2]. It does not perform any function evaluations. The slave processors repeatedly evaluate the fitness function using the particles assigned to them. After receiving the fitness function value and corresponding particle number i from a slave processor, the master processor stores the particle number at the end of the queue, updates the position of the first particle, and sends the first particle back to the idle slave processor for

evaluation. The particle order changes depending on the speed with which each processor completes its function evaluations.

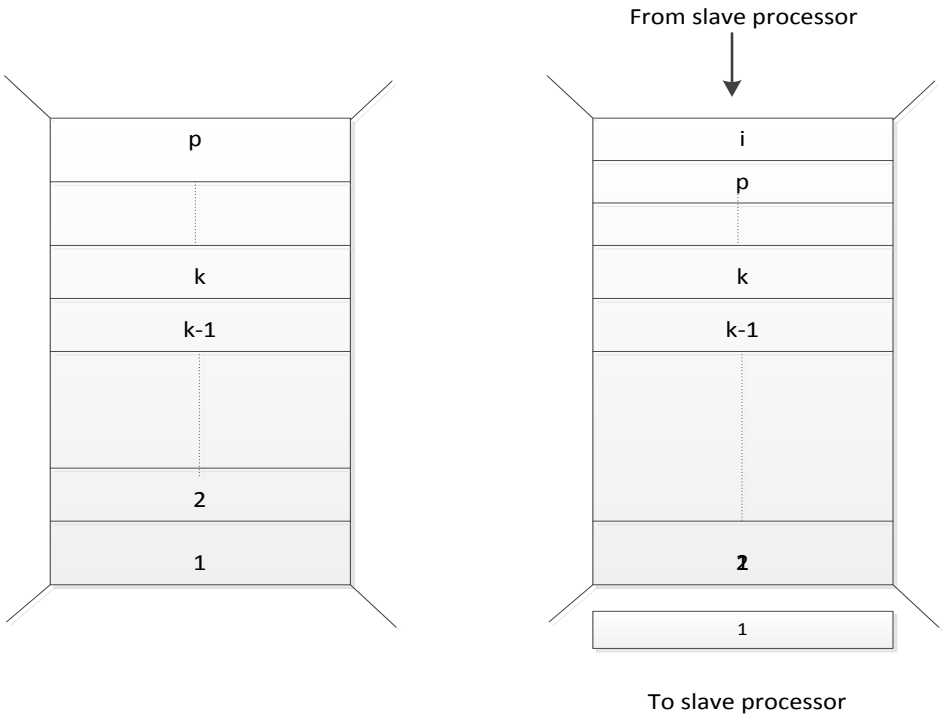

Figure 6.1: Parallel asynchronous PSO

The tasks performed by the master and slave processors are as follows:

Master processor

1. Initializes all constants and particle positions and velocities;

2. Holds a queue of particles for slave processors to evaluate;

3. Updates particle positions and velocities based on available information;

4. Sends the position x_k^i of the next particle in the queue to an available slave processor;

5. Receives fitness function values from slave processors;

6. Checks convergence.

Slave processor

7. Receives a particle position from the master processor;

8. Evaluates the fitness function f(x) at the given particle position x_k^i ;

9. Sends a cost function value to the master processor.

Once the initialization step has been performed by the master processor, particles are sent to the slave processors to evaluate the fitness function. Communication between master and slave processors is achieved using a point-to-point communication scheme and the communication time in the asynchronous case is hidden within the computation time of the slave processors. Because the master processor can communicate with only one slave processor at a time, each slave processor remains idle for a short period of time while waiting to connect to the master processor after completing a function evaluation [2].

6.1.1. General Settings

This application has some general settings that are:

1. Initialization: Uniform random initialization within the search space.

2. Termination: Terminate when reaching the maximum number of function evaluations (Max FEs).

3. Global optimum: All problems have the global optimum within the given bounds, so there is no need to perform search outside of the given bounds for these problems. The optimum function values are 0.0 for all problems.

4. Inertia Weight: Throughout the application this is constant to 0.95.

5. Global Increment and Particle Increment are also constant to 0.9.

6. Number of Runs: 30.

For the experiment four benchmark functions have been used to evaluate the performance of the parallelized PSO algorithms. A two-dimensional search space is considered for the benchmark functions, and therefore, the exploration of particles. Number of runs are considered as 30 so that the sample observations drawn from a sample population forms a normal distribution as stated by central limit theorem [8]. For the test case 1 Gbest PSO algorithm has been considered and for test case 2 Lbest PSO algorithm has been considered. For the Lbest PSO, particle indices have been used to bind the particles to a neighborhood.

6.2. Results

This section describes various test cases that are performed to evaluate the performance of basic PSO (abbreviated as Basic in the tables below) and distributed PSO i.e., synchronous case and asynchronous case (abbreviated as Sync and Async respectively in the tables below). A standard of fourteen particles is considered throughout the experiment while the number of iterations is being altered. 30 independent runs were performed and the average fitness, best fitness, and the average time per iteration in ms are reported.

6.3. Test Case 1: Number of Iterations = 10,000

In this test case, tests are performed by changing the number of iterations and keeping the number of dimensions = 2, number of particles = 14, inertia weight = 0.95, global increment = 0.9, particle increment = 0.9, number of runs = 30. Table 6.1 shows the average fitness for every algorithm for ten thousand iterations.

Table 6.1: Fitness values of Alpine for 10,000 iterations

	Average Fitness	Best Fitness	Average Time(ms)/iteration
Basic	7.59815E-06	0.0	608.00
Sync	2.30371E-20	0.0	729.20
Async	1.22298E-20	0.0	731.00

As seen from Table 6.1, asynchronous produces the best results followed by the Synchronous and Basic algorithms. All of the algorithms were able to reach the optimal value of 0.0 for the Alpine benchmark function when run for 10,000 iterations. Looking at the average time per iteration, Basic PSO has the shortest execution time followed by Synchronous and Asynchronous PSO. As noticed, the average time difference between Synchronous is only 2 seconds.

Table 6.2: Fitness values of Rastrigin for 10,000 iterations

	Average Fitness	Best Fitness	Average Time(ms)/iteration
Basic	4.71354E-04	0.0	430.00
Sync	4.37782E-05	0.0	641.80
Async	3.74999E-05	0.0	653.71

As seen from Table 6.2, Asynchronous PSO has best value closest to the optimal value on the Rastrigin benchmark function. Average execution time for Basic PSO is less when compared to the Synchronous case.

Table 6.3: Fitness values of Rosenbrock for 10,000 iterations

	Average Fitness	Best Fitness	Average Time(ms)/iteration
Basic	7.64946E-06	0.0	2041.00
Sync	6.42339E-06	0.0	2659.52
Async	4.01656E-06	0.0	2659.08

Considering 10,000 iterations for the Rosenbrock benchmark, Asynchronous PSO produced a far better result compared to the other two PSO algorithms. The average best fitness value for Basic PSO algorithm is 7.649E-06 and for Asynchronous algorithm is 4.01656E-06, which represents a significant increase towards the optimal value on the Rosenbrock benchmark function. The average execution time per iteration is shortest for Basic PSO followed by Synchronous and Asynchronous PSO. Also, all the algorithms were able to achieve the optimal value of 0.0.

As seen from Table 6.4, Asynchronous PSO produced the best results of 2.39429E-06 followed by Synchronous and Basic PSO. All the algorithms were able to reach the optimal value of zero for the Schaffer benchmark function when run for 10,000 iterations. Considering the average execution time per iteration, Basic PSO has the minimum execution time followed by Synchronous and Asynchronous PSO.

Table 6.4: Fitness values of Schaffer for 10,000 iterations

	Average Fitness	Best Fitness	Average Time(ms)/iteration
Basic	8.40341E-05	0.0	480.00
Sync	1.99571E-06	0.0	658.92
Async	2.39429E-06	0.0	656.68

As seen from Table 6.4, Asynchronous PSO produced the best results of 2.39429E-06 followed by Synchronous and Basic PSO. All the algorithms were able to reach the optimal value of zero for the Schaffer benchmark function when run for 10,000 iterations. Considering the average execution time per iteration, Basic PSO has the minimum execution time followed by Synchronous and Asynchronous PSO.

6.4. Test Case 2: Number of Iterations = 100,000

In this test, the number of iteration is set to 100,000 and keeping the number of dimensions = 2, number of particles = 14, inertia weight = 0.95, global increment = 0.9, particle increment = 0.9, number of runs = 30.

Table 6.5: Fitness values of Alpine for 100,000 thousand iterations

	Average Fitness	Best Fitness	Average Time(ms)/iteration
Basic	2.07154E-06	0.0	6739.00
Sync	2.68304E-22	0.0	18826.57
Async	3.96508E-23	0.0	19523.86

As seen from Table 6.5, Asynchronous PSO has the best fitness value when compared with the other two algorithms. There was a significant increase in the average fitness from Basic PSO to Asynchronous PSO. When compared to 10,000 iterations of the Alpine benchmark function, the average fitness has changed from 1.22298E-20 to 3.96508E-23, indicating the average fitness value is getting closer to the 0.0 as the number of iterations increases. In both test cases of 10,000 and 100,000 iterations, the Alpine benchmark function was able to reach the optimum value of 0.0. The execution time for 100,000 iterations has increased tremendously when compared to 10,000 iterations.

Table 6.6: Fitness values of Rastrigin for 100,000 iterations

	Average Fitness	Best Fitness	Average Time(ms)/iteration
Basic	6.24671E-05	0.0	6500.00
Sync	4.64737E-06	0.0	28277.37
Async	6.30698E-06	0.0	29014.07

As seen from Table 6.6, Asynchronous PSO has the best value closest to the optimal value on the Rastrigin benchmark function. The average execution time for Basic PSO is less when compared to Synchronous and Asynchronous PSO. When compared to Asynchronous PSO, the execution time of Synchronous case is better by 12 seconds for every iteration. From Table 6.2, we can see the average fitness value has an exponential increase when the number of iteration increased from 10,000 to 100,000 iterations. Also, there was an exponential increase in execution time for 100,000 iterations compared to 10,000 iterations.

As seen from Table 6.7, Asynchronous PSO produced the best results of 3.89245E-07 followed by Synchronous and Basic PSO. All the algorithms were able to reach the optimal value of 0.0 for the Rosenbrock benchmark function when run for 100,000 iterations. When compared with Rastrigin, the average fitness value for 10,000 iterations, 100,000 iterations has the best fitness value of 3.89245E-07. Considering the

average execution time per iteration, Basic PSO has the minimum execution time followed by Synchronous and Asynchronous PSO.

Table 6.7: Fitness values of Rosenbrock for 100,000 iterations

	Average Fitness	Best Fitness	Average Time(ms)/iteration
Basic	5.38134E-06	0.0	45771.00
Sync	5.01774E-07	0.0	89340.64
Async	3.89245E-07	0.0	90568.40

Table 6.8: Fitness values of Schaffer for 100,000 iterations

	Average Fitness	Best Fitness	Average Time(ms)/iteration
Basic	3.90423E-05	0.0	4998.00
Sync	1.66979E-07	0.0	14824.15
Async	2.82962E-07	0.0	14892.28

As noticed from Table 6.8, Asynchronous PSO has the best average fitness value when compared to Basic and Synchronous PSO. Also, the average fitness value for 100,000 iterations has a far better value when compared to the fitness value of 10,000 iterations.

Table 6.9: Percentage improvement of fitness value for Alpine benchmark

Alpine	Percentage improvement [%]
Basic	72.74
Sync	98.84
Async	99.68

As seen from Table 6.9, the fitness value for Alpine bfunction is approaching the optimal value as the number of iterations is increased from 10,000 to 100,000. Also, there is a significant increase in the fitness value when the number of iterations is increased.

Table 6.10: Percentage improvement of fitness value for Rastrigin benchmark

Rastrigin	Percentage improvement [%]
Basic	86.75
Sync	89.38
Async	83.18

As seen from Table 6.10, the fitness value is approaching the optimal value with increasing numbers of iterations. The fitness value of Rastrigin has increased over 83% for all algorithms.

Table 6.11: Percentage improvement of fitness value for Rosenbrock benchmark

Rosenbrock	Percentage improvement [%]
Basic	29.65
Sync	92.19
Async	90.31

As seen from Table 6.11, Synchronous PSO has the highest percentage fitness increase followed by Asynchronous and Basic PSO.

Table 6.12: Percentage improvement of fitness value for Schaffer benchmark

Schaffer	Percentage improvement [%]
Basic	53.54
Sync	91.63
Async	88.18

As seen Table 6.12, there was a significant improvement in fitness with increasing number of iterations for the Schaffer benchmark function.

Overall, Asynchronous PSO was able to produce better fitness values compared to Basic and Synchronous PSO algorithms on all four benchmark functions considered. All of the algorithms were able to achieve higher fitness values when the number of

iterations are increased, which clearly indicates that fitness values are improving as the number of iterations are increasing. There was a tradeoff between the execution time and the optimal values. Execution time has exponential increased when the number of iterations was increased. This is due to the latebcy factor involved for the small cluster configuration.

6.5. Test Case 3: Different Neighborhood Structures

6.5.1. Star Structured Neighborhood

In this test case, the star neighborhood is considered. There are three neighborhoods in total in which each neighbor has five particles. For every iteration, the best value from the neighborhood is chosen. Tables 6.17, 6.18, 6.19, and 6.17 show the average fitness for Basic PSO when the star neighborhood is considered.

Table 6.13: Fitness values of Basic PSO for Alpine benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	4.51E-06	0.0	565.00
100,000	8.34E-07	0.0	6002.00

Table 6.14: Fitness values of Basic PSO for Rastrigin benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	5.40E-03	0.0	437.00
100,000	2.10E-05	0.0	4514.00

Table 6.15: Fitness values of Basic PSO for Rosenbrock benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	2.93E-04	0.0	3765.00
100,000	1.10E-05	0.0	45638.00

Table 6.16: Fitness values of Basic PSO for Schaffer benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	2.49E-03	0.0	466.00
100,000	4.16E-06	0.0	4915.00

As seen from Tables 6.13, 6.14, 6.15 and 6.16, the average fitness value has always increased when the number of iterations has increased from 10,000 to 100,000. Rastrigin has the highest percentage increase in the average fitness towards the optimal value when iteration count has been increased. Rastrigin has 99.61% increase

followed by Schaffer, Rosenbrock and Alpine with 96.55%, 99.16%, and 81.52% increases, respectively.

6.5.2. Circular Structured Neighborhood

In this test case, the circular neighborhood is considered. In total there are five neighborhoods in which each neighborhood has three particles. Particles in the neighborhood communicate with each other and the best value is chosen for every iteration. Table 6.17 shows the Average fitness for Basic PSO algorithm when circular neighborhood is considered.

Table 6.17: Fitness values of Basic PSO for Alpine benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	5.22E-06	0.0	592.00
100,000	8.51E-07	0.0	6648.33

Table 6.18: Fitness values of Basic PSO for Rastrigin benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	3.32E-01	0.0	436.00
100,000	2.20E-05	0.0	4520.00

Table 6.19: Fitness values of Basic PSO for Rosenbrock benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	4.92E-04	0.0	3468.00
100,000	1.20E-05	0.0	43142.50

Table 6.20: Fitness values of Basic PSO for Schaffer benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	6.51E-03	0.0	473.00
100,000	8.10E-06	0.0	5017.00

As seen from Tables 6.17, 6.18, 6.19 and 6.20, the average fitness value has always increased when the number of iterations has increased from 10,000 to 100,000. Rastrigin have the highest percentage increase in the average fitness value towards the optimal value followed by Schaffer, Rosenbrock and Alpine when the iteration count has been increased.

6.5.3. Wheel Structured Neighborhood

In this test case, the wheel structured neighborhood is considered. There are three neighborhoods in total in which each neighbor has five particles. In the wheel social structure, individuals in a neighborhood are isolated from one another. One

particle serves as the focal point, and all information is communicated through the focal particle. The focal particle compares the performances of all particles in the neighborhood, and adjusts its position towards the best neighbor. If the new position of the focal particle results in better performance, then the improvement is communicated to all the members of the neighborhood. Tables 6.21, 6.22, 6.23, and 6.24 show the average fitness for Basic PSO algorithm when the wheel neighborhood is considered.

Table 6.21: Fitness values of Basic PSO for Alpine benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	1.14E-05	0.0	551.00
100,000	9.77E-07	0.0	5244.00

Table 6.22: Fitness values of Basic PSO for Rastrigin benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	4.98E-01	0.0	440.00
100,000	3.92E-05	0.0	4504.00

Table 6.23: Fitness values of Basic PSO for Rosenbrock benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	5.13E-03	0.0	3992.00
100,000	2.09E-04	0.0	44595.00

Table 6.24: Fitness values of Basic PSO for Schaffer benchmark

Iterations	Average Fitness	Best Fitness	Average Time(ms)/iteration
10,000	9.42E-02	0.0	469.00
100,000	3.25E-03	0.0	4924.00

As seen from Tables 6.21, 6.22, 6.23, and 6.24, the average fitness has been approaching the optimal values for the benchmark functions as the number of iterations has increased. Rastrigin have the highest percentage increase in the average fitness value towards the optimal value followed by Schaffer, Rosenbrock and Alpine. Of all the neighborhood structures considered, the star structure neighborhood has the average fitness value close to the optimal value due to larger interconnectivity of neighborhood particles.

7. CONCLUSION AND FUTURE WORK

PSO is an extremely simple algorithm that has proven to be effective for the optimization of a wide range of functions. It is a stochastic optimization method used to solve complex engineering problems. Recent advances in computer and network technologies provide an option to do parallelization and helps in solving complex engineering problems. In this paper, parallelization has been applied to PSO to determine its efficiency. Based on the results obtained, we can conclude that the parallelization of PSO has provided better results. We were able to compare the results on standard benchmark functions and determine the parallelization efficiency. Overall, the parallelization using Asynchronous PSO has provided the best results compared to Synchronous and Basic PSO.

Since this paper serves only as a preliminary study into the parallelization of the PSO algorithm, much more comprehensive experiments need to be conducted. For example, only 2-dimensional benchmark functions were investigated. However, the complexity arising from higher dimensionality needs to be looked at. Furthermore, since our MPJ cluster only consisted of 15 nodes, more intensive experiments with a larger number of nodes need to be performed. Further parameters to look at are the PSO specific values such as the swarm size, global increment, local increment, etc.

8. REFERENCES

[1] D. Ortiz-Boyer, "A Crossover Operator for Evolutionary Algorithms Based on Population Features," Journal of Artificial Intelligence Research, vol. 24, 2005.

[2] A.Engelbrecht, Computational Intelligence: An Introduction, John Wiley & Sons, Ltd, 2007.

[3] K. Byung I and R. Jeffrey, "Evaluation of Parallel Decomposition Methods for Biomechanical Optimizations," Computer Methods in Biomechanics and Biomedical Engineering, vol. 7, no. 4, 2004.

[4] Z. Huang, "Message Passing Java:Performance Results for Java and GridTcp," 2005. [Online]. Available: http://depts.washington.edu/dslab/reports/zhiji_sp05.pdf.

[5] M. Baker, "MPJ Express: An Implementation of MPI," [Online]. Available: http://www.mpj-express.org.

[6] K. Byung-I and G. Alan, "Parallel asynchronous particle swarm optimization," International Journal For Numerical Methods In Engineering, vol. 67, pp. 578-595, 2006.

[7] J.Schutte and J. Reinbolt, "Parallel Global Optimization with the Particle Swarm Algorithm," International Journal of Numeric Methods Engineering, vol. 61, pp.

2296–2315, 2007.

[8] M. Clerc, Particle Swarm Optimization, John Wiley & Sons, 2005.

[9] Eberhart and J. Kennedy, "Particle Swarm Optimization," IEEE International Conference on Neural Networks, vol. 6, no. 1995, 2005.

[10] R. Luis, "Stability of the Supply Chain Using System Dynamics Simulation and the Accumulated Deviations from Equilibrium," Modelling and Simulation in Engineering, 2011.

[11] M. Marcin and S. Czesław, "Test functions for optimization needs," 2005. [Online]. Available: www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf.

[12] A. Razana, "The Impact of Social Network Structure in Particle Swarm Optimization for Classification Problems," International Journal of Soft Computing, vol. 4, no. 4, pp. 151-156, 2009.