2011-01-27

Parallel Particle Swarm Optimization and Large Swarms

Andrew W. McNabb
Brigham Young University - Provo

Parallel Particle Swarm Optimization and Large Swarms

Andrew McNabb

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Kevin Seppi, Chair
Dan Ventura
Mark Clement

Department of Computer Science

Brigham Young University

April 2011

ABSTRACT

Parallel Particle Swarm Optimization and Large Swarms

Andrew McNabb

Department of Computer Science

Master of Science

Optimization is the search for the maximum or minimum of a given objective function. Particle Swarm Optimization (PSO) is a simple and effective evolutionary algorithm, but it may take hours or days to optimize difficult objective functions which are deceptive or expensive. Deceptive functions may be highly multimodal and multidimensional, and PSO requires extensive exploration to avoid being trapped in local optima. Expensive functions, whose computational complexity may arise from dependence on detailed simulations or large datasets, take a long time to evaluate. For deceptive or expensive objective functions, PSO must be parallelized to use multiprocessor systems and clusters efficiently.

This thesis investigates the implications of parallelizing PSO and in particular, the details of parallelization and the effects of large swarms. PSO can be expressed naturally in Google's MapReduce framework to develop a simple and robust parallel implementation that automatically includes communication, load balancing, and fault tolerance. This flexible implementation makes it easy to apply modifications to the algorithm, such as those that improve optimization of difficult objective functions and improve parallel performance. Results show that larger swarms help with both of these goals, but they are most effective if arranged into sparse topologies with lower overhead from communication. Additionally, PSO must be modified to use communication more efficiently in a large sparse swarm for objective functions where information ideally flows quickly through a large swarm.

Swarm size is usually fixed at a modest number around 50, but particularly in a parallel computational environment, much larger swarms are much more effective for deceptive objective functions. Likewise, swarms much smaller than 50 are more effective for expensive but less deceptive functions. In general, swarm size should be carefully chosen using all available information about the objective function and computational environment.

Contents

Chapter 1

Introduction

Optimization, the problem of finding the maximum or minimum of a given objective function, has myriad applications across a wide range of fields. Particle Swarm Optimization (PSO) is an optimization algorithm that was inspired by experiments with simulated bird flocking [15]. This evolutionary algorithm [8] has become popular because it is simple, requires little tuning, and has been found to be effective for a wide range of problems.

Often a function that needs to be optimized takes a long time to evaluate. A problem using web content, commercial transaction information, or bioinformatics data, for example, may involve large amounts of data and require minutes or hours for each function evaluation. In a standard serial implementation of PSO, each iteration is slow for such functions because the function must be evaluated sequentially for each particle in the swarm. Alternatively, some functions may be easy to evaluate but difficult to optimize. For example, high dimensional problems may take many iterations to converge, and functions with deceptive local optima may converge prematurely. In both of these cases, PSO must be parallelized to fully utilize resources in multiprocessor systems and supercomputing clusters. Chapter 2 presents a simple and robust parallel implementation of PSO based on Google's MapReduce parallel programming framework [7].

In order to fully realize the benefits of a robust parallel implementation, the PSO algorithm itself must be tuned to maximize performance. Of primary concern are the issues of topology, which determines communication characteristics of a particle swarm, and swarm size, which affects the scalability of parallel PSO. Communication in PSO increases with the density of the swarm topology. For example, the complete (fully connected) topology requires pairwise communication between all particles, while the ring topology requires each particle to communicate with only a few neighbors. Swarm size is naturally increased in parallel particle swarms because in general, parallel PSO performs one function evaluation per processing core at any given time. With the widespread availability of multicore processors and even a modest number of particles per core, swarms running on a cluster could easily have thousands of particles. Although a few applications have successfully used larger swarm sizes [31, 29], PSO has typically been used with small swarms of 50 particles [3]. Since swarm topology and size have a greater importance in parallel PSO, Chapter 3 considers the the effects of large swarm sizes and various choices of topology and finds that contrary to conventional wisdom, the choice of swarm size is important in serial PSO as well.

1.1 Background

1.1.1 Particle Swarm Optimization

Particle Swarm Optimization simulates the motion of particles in the domain of a function. These particles search for the optimum by evaluating the function as they move. During each iteration of the algorithm, the position and velocity of each particle are updated. Each particle is pulled toward the best position it has sampled (personal best) and the best position of any particle in its neighborhood (neighborhood best). This attraction is weak enough to allow exploration but strong enough to encourage exploitation of good locations and to guarantee convergence.

Each particle's position and velocity are initialized to random values based on a function-specific feasible region. During each iteration of PSO, each particle's position and velocity are updated with respect to the current positions of its personal best and neighborhood best by simple motion formulas. We use the Constricted PSO formulas, which include terms to gradually slow particle's velocity [6]. Constricted PSO has emerged as the standard variant of PSO [3].

The topology or sociometry of a swarm indicates how information is communicated between particles. The most typical representation of swarm topology is an undirected graph, where each vertex is a particle. The neighborhood of a particle—the set of vertices with shared edges—has two intuitive interpretations. From one perspective, it indicates which neighbors (or "informants") contribute their personal bests to a particle's neighborhood best. Alternatively, it indicates which neighbors a particle sends its personal best to.

Particle swarms are usually small, with about 50 particles in either the complete (fully-connected) or ring topology [3, 21]. Although simple static topologies are most common, dynamic and adaptive topologies include TRIBES [5], stochastic star [23], an increasingly connected ring [30], Randomized Directed Neighborhoods [24], and Dynamic Multi-Swarm [17].

Large particle swarms have received little attention, but it has been observed that larger swarm sizes may improve the rate of convergence [31, 29], although adding particles to the "fully-informed PSO" has been less successful [25]. Dividing a large swarm into smaller subswarms is an alternative to using a single global swarm [13, 17].

1.1.2 MapReduce

MapReduce is a functional programming model that is well suited to parallel computation. In this model, a program consists of two high-level functions, a map function and a reduce function, which meet a few simple requirements. If a problem is formulated in this way, it can be parallelized automatically. The MapReduce implementation manages the details of parallelization so that the user-specified program does not have to [7].

Although not all algorithms can be efficiently formulated in terms of map and reduce functions, MapReduce provides many benefits over other parallel processing models. In this model, a program consists of only a map function and a reduce function. The infrastructure provided by a MapReduce implementation manages all of the details of communication, load balancing, fault tolerance, resource allocation, job startup, and file distribution. This runtime system is written and maintained by parallel programming specialists, who can ensure that the system is robust and optimized, while those who write mappers and reducers can focus on the problem at hand without worrying about implementation details.

1.2 Thesis Statement

Particle Swarm Optimization can be expressed in the MapReduce model to provide a flexible and robust parallel implementation. Large swarms, which work particularly well in this implementation, improve PSO performance for some types of objective functions. To achieve the best parallel performance for many of these functions, communication must be reduced by using specialized topologies and by modifying the algorithm to accelerate the flow of information through the swarm.

1.3 Thesis Overview

The thesis consists of two chapters. Chapter 2 considers the details of implementing PSO as a MapReduce program and considers the parallel performance of this implementation. Chapter 3 investigates the implications of large swarms of thousands of particles with various topologies and determines their effectiveness on several benchmark functions.

Chapter 2, which has been published independently [19], reformulates Particle Swarm Optimization as a sequence of map and reduce functions. It evaluates the performance of this parallel algorithm and determines when this is an effective approach.

In an iteration of Particle Swarm Optimization, each particle in the swarm moves to a new position, updates its velocity, evaluates the function at the new point, updates its personal best if this value is the best seen so far, and updates its global best after comparison with its neighbors. Except for updating its global best, each particle updates independently of the rest of the swarm.

Due to the limited communication among particles, updating a swarm can be formulated as a MapReduce operation. As a particle is mapped, it receives a new position, velocity, value, and personal best. In the reduce phase, it incorporates information from other particles in the swarm to update its global best. This implementation conforms to the MapReduce model while performing the same calculations as standard Particle Swarm Optimization. In other words, given the same random seed, the particles sample the exact same sequence of positions as in the serial implementation.

Chapter 3, portions of which have been published [18], considers the effects of increasing the number of particles in a swarm. Since parallelism is usually most effective when a problem can be split into fine-grained tasks, decomposability of parallel PSO is improved by increasing the number of particles in a swarm. This raises the question of how PSO performs with large particle swarms of thousands of particles.

The ideal topology and swarm size for Particle Swarm Optimization depends on the objective function. Mendes [21] compared the performance of PSO with a number of different benchmark functions under the assumption that the benchmark functions represent broad classes of interesting functions. Bratton and Kennedy [3] found no significant difference in performance for any swarm sizes between 20 and 100, but Perez and Basterrechea [29] and Schutte et al. [31] found that large swarms could improve performance for some functions.

The chapter compares the performance of PSO on standard benchmark functions with a variety of topologies, especially those with conservative levels of communication. This provides some basic observations about how the choice of topology affects performance with a large swarm. The chapter also considers algorithmic changes to accelerate the flow of information through a swarm.

The balance between exploration and exploitation is function-dependent. The Sphere, Griewank, and Rastrigin benchmark functions are representative of three different types of functions. Sphere is unimodal and smooth. It is most easily optimized when information flows between particles as quickly as possible. Griewank has more deceptive local minima and is poorly optimized if information flows too quickly through the swarm. In the case of Rastrigin, PSO performs poorly if the amount of communication is at either extreme. For a low dimensional Rastrigin function, PSO performs best when information flows quickly, and for a high dimensional problem, it performs best when information flows at a more moderate rate. The chapter shows that parallel PSO can be tuned to provide any needed level of information flow efficiently.

Chapter 2

Parallel PSO Using MapReduce

Published in Proceedings of CEC 2007

In optimization problems involving large amounts of data, such as web content, commercial transaction information, or bioinformatics data, individual function evaluations may take minutes or even hours. Particle Swarm Optimization (PSO) must be parallelized for such functions. However, large-scale parallel programs must communicate efficiently, balance work across all processors, and address problems such as failed nodes.

We present MapReduce Particle Swarm Optimization (MRPSO), a PSO implementation based on the MapReduce parallel programming model. We describe MapReduce and show how PSO can be naturally expressed in this model, without explicitly addressing any of the details of parallelization. We present a benchmark function for evaluating MRPSO and note that MRPSO is not appropriate for optimizing easily evaluated functions. We demonstrate that MRPSO scales to 256 processors on moderately difficult problems and tolerates node failures.

2.1 Introduction

Particle Swarm Optimization (PSO) is an optimization algorithm that was inspired by experiments with simulated bird flocking [15]. This evolutionary algorithm has become popular because it is simple, requires little tuning, and has been found to be effective for a wide range of problems. Often a function that needs to be optimized takes a long time to evaluate. A problem using web content, commercial transaction information, or bioinformatics data, for example, may involve large amounts of data and require minutes or hours for each function evaluation. To optimize such functions, PSO must be parallelized.

Unfortunately, large-scale PSO, like all large-scale parallel programs, faces a wide range of problems. Inefficient communication or poor load balancing can keep a program from scaling to a large number of processors. Once a program successfully scales, it must still address the issue of failing nodes. For example, assuming that a node fails, on average, once a year, then the probability of at least one node failing during a 24-hour job on a 256-node cluster is $1-(1-1/365)^{256} = 50.5\%$. On a 1000-node cluster, the probability of failure rises to 93.6%.

Google faced these same problems in large-scale parallelization, with hundreds of specialized parallel programs that performed web indexing, log analysis, and other operations on large datasets. A common system was created to simplify these programs. Google's MapReduce is a programming model and computation platform for parallel computing [7]. It allows simple programs to benefit from advanced mechanisms for communication, load balancing, and fault tolerance.

MapReduce Particle Swarm Optimization (MRPSO) is a parallel implementation of Particle Swarm Optimization for computationally intensive functions. MRPSO is simple, flexible, scalable, and robust because it is designed in the MapReduce parallel programming model.

Since MRPSO is intended for computationally intensive functions, we use the problem of training a radial basis function (RBF) network as representative of optimization problems which use large amounts of data. An RBF network is a simple function approximator that can be trained by PSO, with difficulty proportional to the amount of training data.

Section 2.2 reviews standard PSO and prior work in parallel PSO. Section 2.3 discusses the MapReduce model in detail, including a simple example. Section 2.4 describes how Particle Swarm Optimization can be cast in the MapReduce model, without any explicit reference to load balancing, fault tolerance, or any other problems associated with large-scale parallelization. Section 2.5 shows that MRPSO scales well through 256 processors on moderately difficult problems but should not be used to optimize trivially evaluated functions.

2.2 Particle Swarm Optimization

In Particle Swarm Optimization, a set of particles explores the input space of a function. Each particle has a position and velocity, which are updated during each iteration of the algorithm. Additionally, each particle remembers its own best position so far (personal best) and the best position found by any particle in the swarm (global best).

Initially, each particle has a random position and velocity drawn from a function-specific feasible region. The particle evaluates the function and updates its velocity such that it is drawn towards its personal best point and the global best point. This influence towards promising locations is strong enough that the particle eventually converges but weak enough that the particles explore a wide area.

The following equations are used in constricted PSO to update the position $\boldsymbol{x}$ and velocity $\boldsymbol{v}$ of a particle with personal best $\boldsymbol{p}$ and global best $\boldsymbol{g}$:

$$\boldsymbol{v}_{t+1} = \chi \left[\boldsymbol{v}_t + \phi_1 \, \mathrm{U}() \otimes (\boldsymbol{p} - \boldsymbol{x}_t) + \phi_2 \, \mathrm{U}() \otimes (\boldsymbol{g} - \boldsymbol{x}_t)\right] \tag{2.1}$$

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \boldsymbol{v}_{t+1} \tag{2.2}$$

where $\phi_1 = \phi_2 = 2.05$, U() is a vector of samples drawn from a standard uniform distribution, and $\otimes$ represents element-wise multiplication. The constriction χ is defined to be:

$$\chi = \frac{2\kappa}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|}$$

where $\kappa = 1.0$ and $\phi = \phi_1 + \phi_2$ [6].

There are several parallel adaptations of Particle Swarm Optimization. Synchronous PSO, like MRPSO, preserves the exact semantics of serial PSO [31]. In contrast, asynchronous variants do not preserve the exact semantics of serial PSO, but instead focus on better load balancing [16, 34]. Other variants propose different topologies to limit communication among particles and between groups of particles [1, 27]. Parallel PSO has been applied to applications including antenna design [12] and biomechanics [16] and adapted to solve multiobjective optimization problems [28, 27].

2.3 MapReduce

MapReduce is a functional programming model that is well suited to parallel computation. In the model, a program consists of a high-level map function and reduce function which meet a few simple requirements. If a problem is formulated in this way, it can be parallelized automatically.

In MapReduce, all data are in the form of keys with associated values. For example, in a program that counts the frequency of occurrences for various words, the key would be a word and the value would be its frequency.

A MapReduce operation takes place in two main stages. In the first stage, the map function is called once for each input record. At each call, it may produce any number of output records. In the second stage, this intermediate output is sorted and grouped by key, and the reduce function is called once for each key. The reduce function is given all associated values for the key and outputs a new list of values (often "reduced" in length from the original list of values).

The following notation and example are based on the original presentation [7].

2.3.1 Map Function

A map function is defined as a function that takes a single key-value pair and outputs a list of new key-value pairs. The input key may be of a different type than the output keys, and the input value may be of a different type than the output values:

$$\mathsf{map} : (K_1, V_1) \rightarrow \mathsf{list}((K_2, V_2))$$

Since the map function only takes a single record, all map operations are independent of each other and fully parallelizable.

2.3.2 Reduce Function

A reduce function is a function that reads a key and a corresponding list of values and outputs a new list of values for that key. The input and output values are of the same type. Mathematically, this would be written:

$$\mathsf{reduce} : (K_2, \mathsf{list}(V_2)) \rightarrow \mathsf{list}(V_2)$$

A reduce operation may depend on the output from any number of map calls, so no reduce operation can begin until all map operations have completed. However, the reduce operations are independent of each other and may be run in parallel.

Although the formal definition of map and reduce functions would indicate building up a list of outputs and then returning the list at the end, it is more convenient in practice to emit one element of the list at a time and return nothing. Conceptually, these emitted elements still constitute a list.

Function 1 WordCount Map

```
def mapper(key, value):
    for word in value.split():
        emit((word, 1))
```

Function 2 WordCount Reduce

```
def reducer(key, value_list):
    total = sum(value_list)
    emit(total)
```

2.3.3 Example: WordCount

The classic MapReduce example is WordCount, a program which counts the number of occurrences of each word in a document or set of documents. For this program, the input and output sets are:

$$\begin{aligned} K_1 &: \mathbb{N} \\ V_1 &: \text{set of all strings} \\ K_2 &: \text{set of all strings} \\ V_2 &: \mathbb{N} \end{aligned}$$

In WordCount, the input value is a line of text. The input key is ignored but arbitrarily set to be the line number for the input value. The output key is a word, and the output value is its count.

The map function, shown as Function 1, splits the input line into individual words. For each word, it emits the key-value pair formed by the word and the value 1.

The reduce function, shown as Function 2, takes a word and list of counts, performs a sum reduction, and emits the result. This is the only element emitted, so the output of the reduce function is a list of size 1.

These map and reduce functions are deceptively simple. The problem itself is inherently difficult—implementing a scalable distributed word count system with fault-tolerance and load-balancing is not easy. However all of the complexity is found in the surrounding MapReduce infrastructure rather than in the map and reduce functions. Note that the reduce function does not even output a key, since the MapReduce system already knows what key it passed in.

The data given to map and reduce functions, as in this example, are generally as fine-grained as possible. This ensures that the implementation can split up and distribute tasks. The MapReduce system consolidates the intermediate output from all of the map tasks. These records are sorted and grouped by key before being sent to the reduce tasks.

If the map tasks emit a large number of records, the sort phase can take a long time. MapReduce addresses this potential problem by introducing the concept of a combiner function. If a combiner is available, the MapReduce system will locally sort the output from several map calls on the same machine and perform a "local reduce" using the combiner function. This reduces the amount of data that must be sent over the network for the main sort leading to the reduce phase. In WordCount, the reduce function would work as a combiner without any modifications.

2.3.4 Benefits of MapReduce

Although not all algorithms can be efficiently formulated in terms of map and reduce functions, MapReduce provides many benefits over other parallel processing models. In this model, a program consists of only a map function and a reduce function. Everything else is common to all programs. The infrastructure provided by a MapReduce implementation manages all of the details of communication, load balancing, fault tolerance, resource allocation, job startup, and file distribution. This runtime system is written and maintained by parallel programming specialists, who can ensure that the system is robust and optimized, while those who write mappers and reducers can focus on the problem at hand without worrying about implementation details.

A MapReduce system determines task granularity at runtime and distributes tasks to compute nodes as processors become available. If some nodes are faster than others, they will be given more tasks, and if a node fails, the system automatically reassigns the interrupted task.

2.3.5 MapReduce Implementations

Google has described its MapReduce implementation in published papers and slides, but it has not released the system to the public. Presumably the implementation is highly optimized because Google uses it to produce its web index.

The Apache Lucene project has developed Hadoop, an Java-based open-source clone of Google's closed MapReduce platform. The platform is relatively new but rapidly maturing. At this time, Hadoop overhead is significant but not overwhelming and is expected to decrease with further development.

2.4 MapReduce PSO (MRPSO)

In an iteration of Particle Swarm Optimization, each particle in the swarm moves to a new position, updates its velocity, evaluates the function at the new point, updates its personal best if this value is the best seen so far, and updates its global best after comparison with its neighbors. Except for updating its global best, each particle updates independently of the rest of the swarm.

Due to the limited communication among particles, updating a swarm can be formulated as a MapReduce operation. As a particle is mapped, it receives a new position, velocity, value, and personal best. In the reduce phase, it incorporates information from other particles in the swarm to update its global best. The MRPSO implementation con-

$(3, \text{“}1, 2, 3, 4; 1.7639, 2.5271; 52.558, 50.444; 9.4976; 1.7639, 2.5271; 9.4976; -1.0151, -2.0254; 5.1325\text{”})$

Figure 2.1: A particle as a key-value pair

forms to the MapReduce model while performing the same calculations as standard Particle Swarm Optimization.

2.4.1 Particle Representation and Messages

In MRPSO, the input and output sets are:

$$\begin{aligned} K_1 &: \mathbb{N} \\ V_1 &: \text{set of all strings} \\ K_2 &: \mathbb{N} \\ V_2 &: \text{set of all strings} \end{aligned}$$

Each particle is identified by a numerical `id` key, and particle state is represented as a string. The state of a particle consists of its dependents list (neighbors' `id`s), position, velocity, value, personal best position, personal best value, global best position, and global best value. The state string is a semicolon-separated list of fields. If a field is vector valued, its individual elements are comma-separated. The state string is of the form:

$$\text{deps; pos; vel; val; pbpos; pbval; gbpos; gbval}$$

A typical particle is shown in Figure 2.1. This particle is exploring the function $f(\boldsymbol{x}) = x_1^2 + x_2^2$. Its components are interpreted as follows:

3	particle id
$1, 2, 3, 4$	dependents (neighbors)
$1.7639, 2.5271$	current position (x_1, x_2)

Function 3 MRPSO Map

```
def mapper(key, value):
    particle = Particle(value)

    # Update the particle:
    new_position, new_velocity = pso_motion(particle)
    y = evaluate_function(new_position)
    particle.update(new_position, new_velocity, y)

    # Emit a message for each dependent particle:
    message = particle.make_message()
    for dependent_id in particle.dependent_list:
        if dependent_id == key:
            particle.gbest_candidate(particle.pbest_position, particle.pbest_
        else:
            emit((dependent_id, repr(message)))

    # Emit the updated particle without changing its id:
    emit((key, repr(particle)))
```

Function 4 MRPSO Reduce

```
def reducer(key, value_list):
    particle = None
    best = None

    # Of all of the inputs, find the record with the best gbest_value:
    for value in value_list:
        record = Particle(value)
        if (best is None) or (record.gbest_value <= best.gbest_value):
            best = record
        if not record.is_message():
            particle = record

    # Update the gbest of the particle and emit:
    if particle is not None:
        particle.gbest_candidate(best.gbest_position, best_value)
        emit(repr(particle))
    else:
        emit(repr(best))
```

$52.558, 50.444$	velocity (x_1, x_2)
9.4976	value of $f(\boldsymbol{x})$ at the current position $(1.7639^2 + 2.5271^2)$
$1.7639, 2.5271$	personal best position (x_1, x_2)
9.4976	personal best value
$-1.0151, -2.0254$	global best position (x_1, x_2)
5.1325	global best value

MRPSO also creates messages, which are like particles except that they have empty dependents lists. A message is sent from one particle to another as part of the MapReduce operation. In the reduce phase, the recipient reads the personal best from the message and updates its global best accordingly.

2.4.2 MRPSO Map Function

The MRPSO map function, shown as Function 3, is called once for each particle in the population. The key is the `id` of the particle, and the value is its state string representation. The PSO mapper finds the new position and velocity of the particle and evaluates the function at the new point. It then calls the update method of the particle with this information. In addition to modifying the particle's state to reflect the new position, velocity, and value, this method replaces the personal best if a more fit position has been found.

The key to implementing PSO in MapReduce is communication between particles. Each particle maintains a dependents list containing the `id`s of all neighbors that need information from the particle to update their own global bests. After the map function updates the state of a particle, it emits messages to all dependent particles. When a message is emitted, its corresponding key is the `id` of the destination particle, and its value is the string representation, which includes the position, velocity, value, and personal best of the source particle. The global best of the message is also set to the personal best.

If the particle is a dependent of itself, as is usually the case, the map function updates the global best of the particle if the personal best is an improvement. Finally, the map function emits the updated particle and terminates.

2.4.3 MRPSO Reduce Function

The MRPSO reduce function, shown as Function 4, receives a key and a list of all associated values. The key is the `id` of a particular particle in the population. The values list contains the newly updated particle and a message from each neighbor. The PSO reducer combines information from all of these messages to update the global best position and global best value of the particle. The reducer emits only the updated particle.

Function 4 also works as a combiner. If no particle is found in the input value list, the function combines the list by emitting only the best message. This message is then sent to the reducer.

2.4.4 Map and Reduce in Context

When a particle is emitted by a reducer, it is fully updated. In the map phase, it updates, moves, and evaluates, and then updates its personal best. In the reduce phase, it updates its global best after receiving messages from all of its neighbors in the swarm. A map phase followed by a reduce phase performs an iteration of the swarm that is exactly equivalent to an iteration in single-processor PSO.

Observe that the MRPSO implementation does not explicitly deal with communication across nodes, load balancing, or node failures. The MapReduce formulation of the problem allows the work to be divided in small enough pieces that the MapReduce system can balance work across processors and deal with failed nodes.

2.5 Results and Remarks

2.5.1 Implementation

Our experiments involve both a serial and a parallel implementation of Particle Swarm Optimization. The two Python programs share code for particle motion and for performing evaluations of the objective function. Particle motion is a straightforward implementation of (2.1) and (2.2).

The serial PSO program creates a swarm, or list of particle objects. During each iteration, it updates the velocity, position, value, and personal best of all of the particles. It then finds the global best, updates all of the particles, and continues to the next iteration.

The MRPSO program performs the same operations as the sequential code. However, instead of performing PSO iterations internally, it delegates this work to the Hadoop MapReduce system. After creating the initial swarm, it saves the particles to a file as a list of key-value pairs, as in Figure 2.1. This file is the input for the first MapReduce operation. Hadoop performs a sequence of MapReduce operations, each of which evaluates a single iteration of the particle swarm. The output of each MapReduce operation represents the state of the swarm after the iteration of PSO, and this output is used as the input for the following iteration. In each MapReduce operation, Hadoop calls map (Function 3) and reduce (Function 4) in parallel to update particles in the swarm. Note that the code that we have shown for these two functions is the actual implementation.

2.5.2 Environment

Performance experiments were run on Brigham Young University's Marylou4 supercomputer. Marylou4 is a cluster of Dell 1955 blade servers. Each node has four 2.6 GHz Xeon cores and 8 GB of memory. In serial experiments, we reserved one processor per node, and in parallel experiments, we used all four processors on each machine. Hadoop version 0.10 in Java 1.6 was used as the MapReduce system for all experiments. Both MRPSO and serial PSO were run in a Python 2.5 interpreter. Hadoop's streaming system provided the interface between Java and Python code.

2.5.3 Methodology

MRPSO performs the same calculations as a serial implementation of PSO. With the same number of particles and iterations, MRPSO and serial PSO will achieve the same level of accuracy. Comparing the quality of solutions is useful only to verify correctness. However, the average execution time per iteration is important because it shows whether the parallel implementation is an improvement. Unless noted otherwise, the first iteration of each run was excluded from averages because they often ran slightly faster or slower than the rest of the runs.

Each swarm consists of 1,000 fully connected particles. Since each particle has a 1,000 neighbors, the dependents list is quite large. Since the sociometry is static in this case, an explicit list is not necessary. To save space, we replaced the full dependents list field in the string representation with the special string "all-1000." When a particle saw this string, it emitted messages for all 1,000 particles in the swarm as if the dependents list were included.

MapReduce has many parameters involving issues such as how to partition the input and how many tasks to run concurrently on each machine. We decided how to set these parameters after performing some initial experiments with n nodes and p processors for various values of n and p. The number of tasks per node, which indicates how many total map and reduce functions can be executing concurrently on one physical computer, was set to 4 (the number of processors per node). The number of map tasks per job, which determines how finely to partition the input, was set to p, the total number of processors. We used $\log_2 n$, but not more than 4, reduce tasks per job. We also configured the MapReduce system to use the reduce function as a combiner.

We used speedup as a measure of scalability. However, there are enough variants of speedup that the measure is worse than useless without precise clarification. Speedup is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processing elements [11]. Thus the speedup with p processors is:

$$S_p = \frac{t_1}{t_p} \tag{2.3}$$

The definition of speedup is ambiguous as to what constitutes the best sequential algorithm. Since MRPSO is a reformulation of PSO that performs the same operations, we use our standard single-processor PSO implementation as the best sequential algorithm. This implementation and the MRPSO implementation are written in the same language, share common code, and run on the same hardware.

2.5.4 RBF Network Training

We used a RBF network training function as our primary test function because it is representative of functions that use large amounts of data. An RBF network is a sum of radial basis functions and is used as a function approximator [2]. The following equation describes the activation function of an RBF network:

$$f(\boldsymbol{x}) = \sum_{i=1}^{n_{\text{bases}}} \frac{s_i}{\sqrt{2\pi}} \exp\left[\left(\sum_{j=1}^{d_{\text{input}}} \frac{w_{ij}}{625}(x_i - c_{ij})^2\right)^{\frac{1}{2}}\right] \tag{2.4}$$

The RBF network f is composed of n_{bases} Gaussian basis functions. Basis i has center $\boldsymbol{c_i}$, input weights $\boldsymbol{w_i}$ (precision), and output scale s_i.

Given a set of training points with known values, a particle swarm can minimize the sum square error function to find the parameters of the RBF network that best fits the data. Thus, the training problem becomes an optimization problem of the error function:

$$g(\boldsymbol{X}, \boldsymbol{y}) = \sum_{i=1}^{n_{\text{points}}} (f(\boldsymbol{X_i}) - y_i)^2 \tag{2.5}$$

where $\boldsymbol{X}$ are the training points and y are the corresponding values. A particle swarm finds parameters for the RBF network f in (2.4) that minimize the error function g shown in (2.5).

To a particle swarm, an RBF network with one-dimensional input and four basis functions is represented as a 12-dimensional vector with 3 parameters for each of the 4 bases, for example:

$$\begin{aligned}&(s_1, w_{11}, c_{11}, s_2, w_{21}, c_{21}, s_3, w_{31}, c_{31}, s_4, w_{41}, c_{41})\\ &= (32, 1.3, -22, 11, 37, 18, 45, 4.3, -7.8, 1.4, 11, 0.53) \qquad (2.6)\end{aligned}$$

2.5.5 RBF Results

We used PSO for the 12-dimensional problem of optimizing weights for an RBF network. The function minimized by PSO was g in (2.5); in this problem, a particle's position is a vector of weights for an RBF network. The training data was a set of 10,000 samples from the RBF network of (2.6).

We ran PSO for the serial implementation and for MRPSO with 1, 4, 8, 16, 32, 64, and 128 processors. In each case, we report the average execution time of at least 70 iterations of PSO. For MRPSO, the estimated standard deviation of execution times ranged from 2.3 seconds for 8 processors to 6.7 seconds for 128 processors. For serial PSO, the estimated standard deviation was 34 seconds (2.8% of the average time). The execution times are shown in Figure 2.2.

With 10,000 training points, each evaluation of g took about 1.2 seconds. Although this is not a very long time, it is much longer than common PSO benchmark functions. In fact, a single iteration in the serial implementation took 1,230 seconds (20.5 minutes) to complete. A training set of 10,000 data points is not large, and a function that takes 1.2 seconds to compute is not slow. However, even with this function, parallelization made a huge difference: with 64 processors, each iteration took 65 seconds.

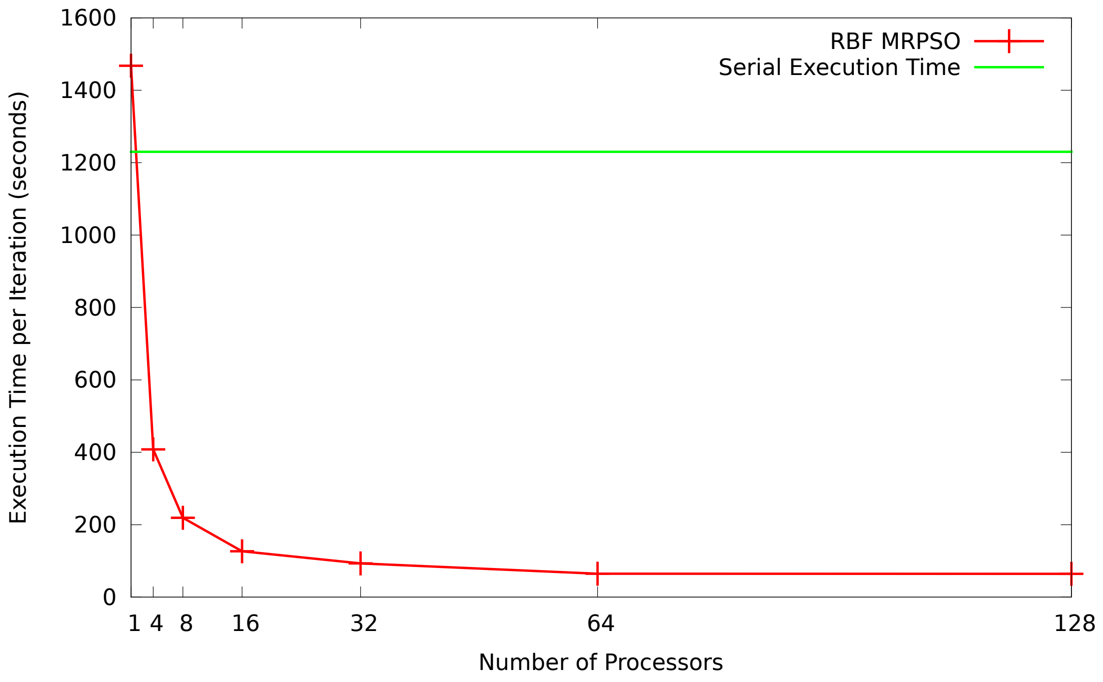

Figure 2.2: Execution times per iteration for RBF with 10,000 points

The speedup, as calculated using (2.3), is shown in Figure 2.3. Improvement was dramatic until 64 processors, but beyond this point, implementation and communication overhead hindered further improvement. For each iteration with 128 processors, the amount of computation per processor was only 9.6 seconds.

2.5.6 Sphere

A MapReduce runtime system introduces overhead due to job startup, communication, and sorting. Additional overhead is incurred by MRPSO's inter-particle messages. If the function being optimized is simple enough that particle communication takes longer than function evaluation, then MRPSO should not be used.

The sphere function is: $f(\boldsymbol{x}) = x_1^2 + x_2^2 + \cdots + x_n^2$. Like all of the standard benchmarks, it is easily evaluated. In our serial PSO, it took less than a millisecond per evaluation on 12-dimensional sphere with 1,000 particles. For parallelization to be useful, there would have to be almost no additional overhead. In MapReduce, the overhead to process each particle

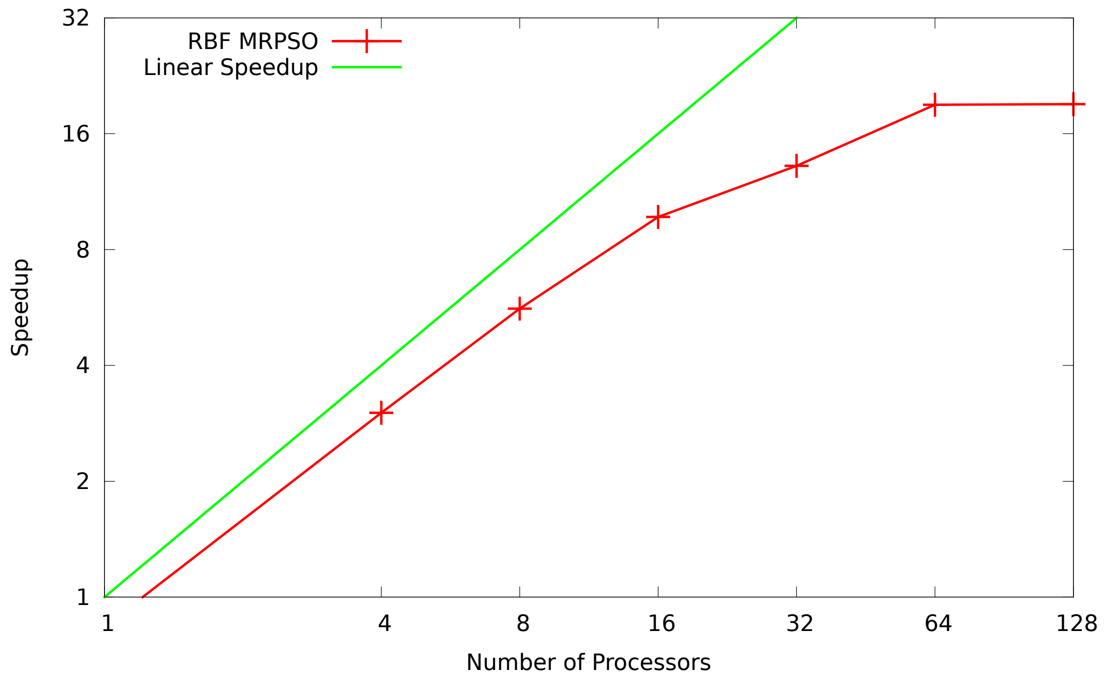

Figure 2.3: Speedup for RBF with 10,000 data points

would certainly be expected to take much longer than a millisecond (on the cluster, round trip time alone is 0.084 milliseconds). In Hadoop 0.10, the amount of time to evaluate sphere was dominated by the time needed by the MapReduce runtime system.

Figure 2.4 shows the speedup of MRPSO with 1,000 particles on 12-dimensional sphere. The baseline is a standard serial implementation which completed each iteration in 0.867 seconds on average. Even at its best, MRPSO took 41.5 seconds per iteration, which is many times slower than a millisecond. MRPSO should not be used for easily evaluated functions.

The sphere function serves another useful purpose by measuring the amount of MRPSO overhead. Since each evaluation of sphere takes less than a millisecond to compute, the function is essentially a null operation compared to the total execution time. Overlaying the graph of RBF execution times with the graph of sphere execution times shows how much of the time was spent on function evaluation and how much was spent on overhead. In Figure 2.5, the time between the two curves approximates the amount of RBF computation

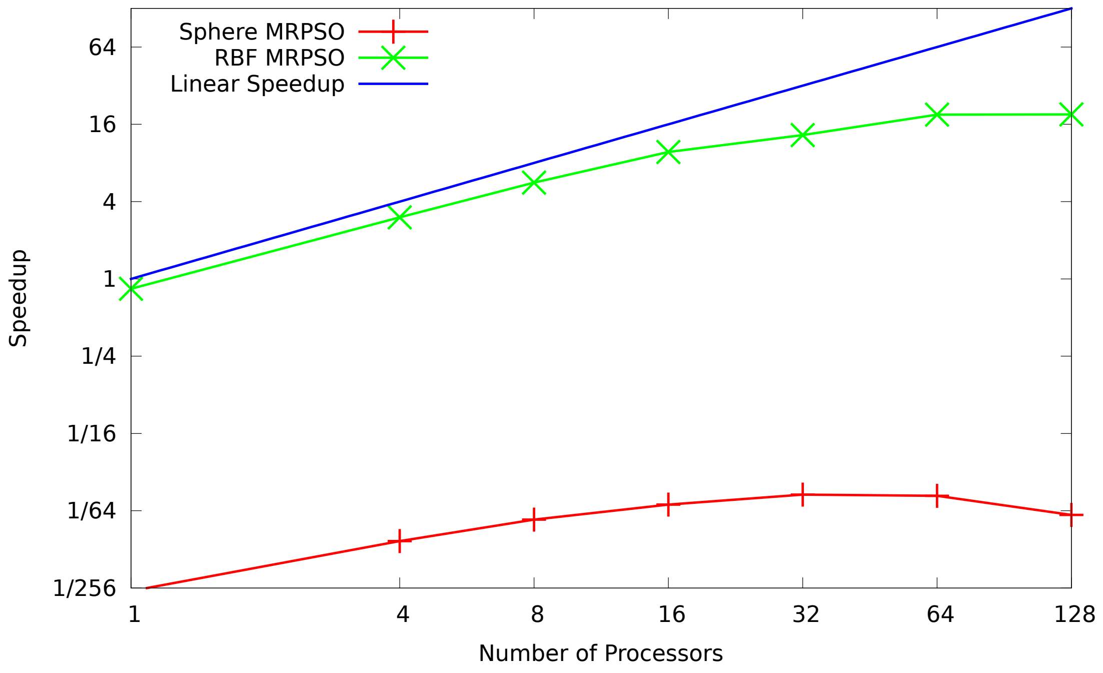

Figure 2.4: Speedup for the sphere function compared to the speedup for RBF. Using MRPSO for sphere would not be appropriate.

time, while the line beneath the lower curve shows the amount of overhead. As the number of processors increases to 128, the two curves nearly meet. After this point, each additional processor increases overhead more than it contributes to computation. Some of the overhead represented by this curve is unavoidable, but much of it will decrease as Hadoop continues to improve.

2.5.7 RBF With 1,000,000 Points

The earlier RBF experiments used 10,000 training points and took 1.2 seconds to compute one function evaluation. Although MRPSO scaled well for this function, it was not particularly long-running function. However, with 100 times as many data points, the RBF network error function from (2.5) takes 100 times longer to run. At 120 seconds per function evaluation, training an RBF network with 1,000,000 training points is noticeably slow. Over 10 serial PSO experiments, the average time per iteration was 120,000 seconds (33 hours), with an estimated standard deviation of 710 seconds (12 minutes).

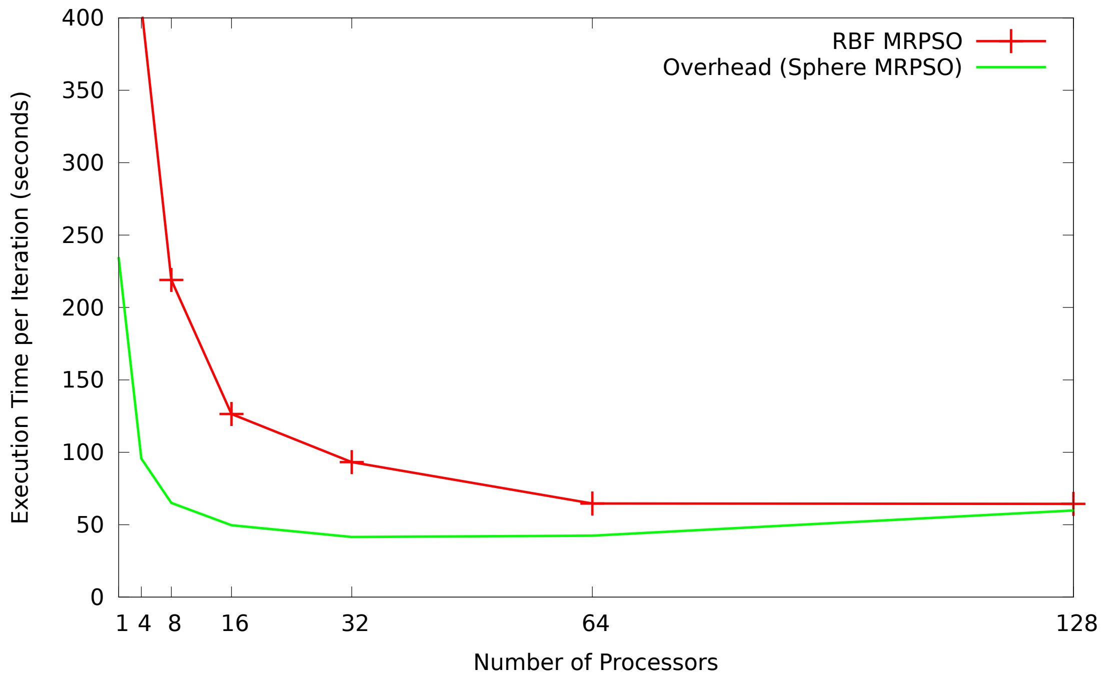

Figure 2.5: Execution times per iteration for RBF with 10,000 data points and implementation overhead as measured by sphere.

MRPSO experiments were similar to the previous experiments. However, the first iteration was not dropped because of the sparsity of data. Also, the 256-processor experiments were run with 500 map tasks instead of 256 because of the need for load balancing, as discussed below.

Figure 2.6 shows the speedup of RBF network training in MRPSO. Note that the RBF nearly matches linear speedup through 128 processors. The speedup with 16 processors is 14.9, and the speedup with 128 processors is 101.

2.5.8 Load Balancing

In each experiment with up to 128 processors, the number of map tasks was equal to the total number of processors. In these experiments, the MapReduce system performed static load balancing. It split the input into similarly sized tasks and assigned a task to each processor.

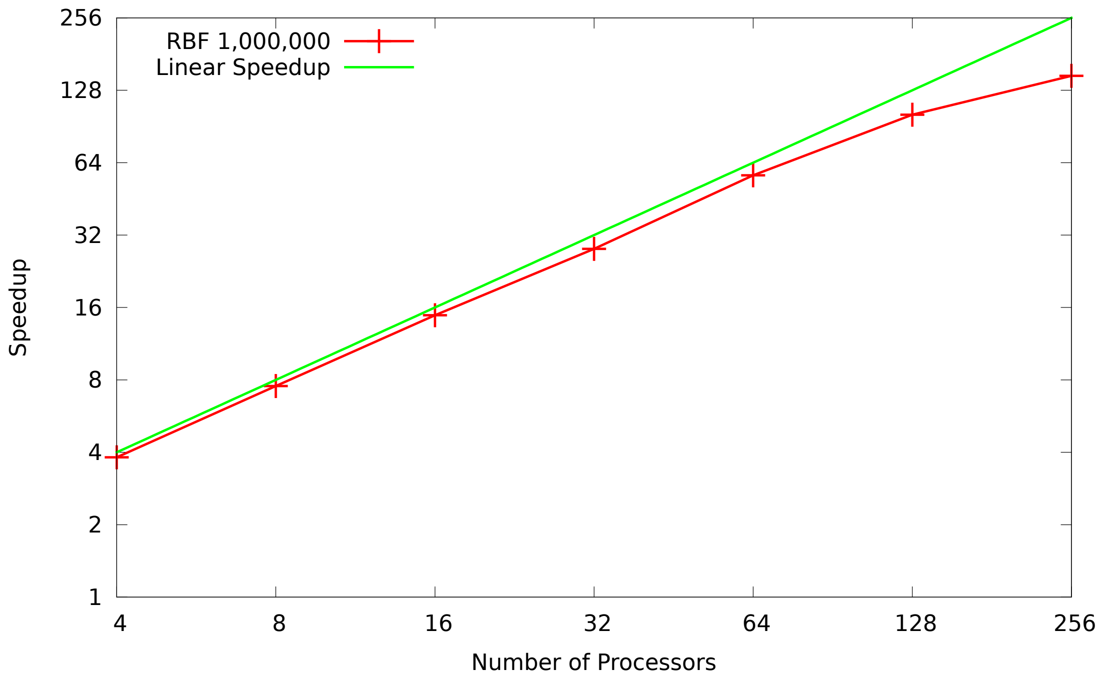

Figure 2.6: Speedup for RBF with 1,000,000 data points

Alternatively, the number of map tasks can be set to the total number of particles. In this case, there would be 1,000 map tasks, with exactly one particle in each map task. However, experimentation showed that this increased overhead.

With 256, we found that it was relatively common for a machine to experience a network error and lose a map task. When this happened, the MapReduce system recognized the fault and restarted the task. The iteration completed successfully, but the reduce tasks could not begin until the last map task completed. With 256 processors, iterations finished in less than 600 seconds in the normal case, but took more than 1,000 seconds in the event of a failure.

To reduce the variance, the number of map tasks was set to 500 instead of 256. Since there were more map tasks than processors, Hadoop performed dynamic load balancing, and if a task failed, the reassigned map task would complete more quickly because it included 2 particles instead of 4. After making this change, the slowest iteration time was 865 seconds rather than 1,078 seconds. The more processors in use, the greater the need for dynamic load balancing.

2.6 Future Work and Conclusions

If an MRPSO swarm has fewer particles than the number of available processors, then the extra processors are idle. With more particles than the number of processors, the MapReduce system can dynamically balance the load. This suggests that with thousands of processors, MRPSO would perform best with a very large number of particles.

The number of messages emitted by the map function is proportional to the size of the particle's dependents list. Because of this, each particle should have a limited number of neighbors. MRPSO makes it easy to control the swarm sociometry, but it is still not clear which sociometries work best in which contexts, and very little work has been done on large swarms. Experiments with more sparsely connected sociometries such as rings, directed rings, and tribes in MRPSO might show how to reduce communication and improve optimization. [14]

Since a particle's dependents list is part of its state, it can be updated during either the map or the reduce phase. Dynamic changes to the dependents list might affect the performance of PSO.

MRPSO makes no assumptions about whether the sociometry is static or dynamic. If the sociometry is assumed to be static, then the map function could refrain from emitting unnecessary messages. In this case, a message would only be emitted in iterations where a particle updates its personal best. This might reduce the average communication overhead.

In summary, we have shown that Particle Swarm Optimization can be naturally adapted to the MapReduce programming model. With a function that took 2 minutes to evaluate, MapReduce Particle Swarm Optimization scaled well through 256 processors. MRPSO addresses the problems that face highly parallel programs because it builds on a system that is specifically designed be robust.

Chapter 3

The Importance of Swarm Size in Serial and Parallel PSO

Prepared for journal submission (portions were published in Proceedings of CEC 2009 [18])

Particle Swarm Optimization (PSO) is a popular evolutionary optimization algorithm. Unfortunately, not enough research has considered the unique challenges associated with parallelization of PSO. Both the widespread availability of multi-core processors and the needs of expensive and difficult optimization problems make parallelization unavoidable.

PSO is usually used with swarms of about 50 particles. However, larger swarms allow for more efficient parallelization. Even with only one particle per core, swarms running on a cluster can have thousands of particles. Not only is swarm size an important issue in parallelization, but it can also have a large effect in serial PSO. When greater exploration is needed, an increased swarm size can reduce premature convergence, and when greater exploitation is desired, smaller swarms of 15 or 20 particles may be more efficient. Although 50 remains a good starting point, swarm size should be tuned for optimal performance.

In the case of parallel PSO, we consider both expensive optimization problems, where the cost of performing the function evaluation outweighs the cost of communication, and also inexpensive optimization problems, where communication may severely limit performance.

3.1 Introduction

In recent years, processor technology has advanced more by increasing the number of cores than by improving the processing speed of each core. Unfortunately, most PSO papers have not addressed parallel computation, and many successful variants of PSO such as TRIBES and FIPS may not have been designed with parallelization in mind. With the widespread availability of multi-core processors, we suspect that most researchers and practitioners currently have access to dozens or hundreds of cores, and the trend toward parallel computing will only increase in the future. Both the availability of cores and the needs of expensive and difficult optimization problems make parallelization unavoidable.

Swarm size is perhaps the most important issue in parallel PSO because it is difficult to decompose the algorithm into more tasks than the number of particles. Even with only one particle per core, swarms running on a cluster can have thousands of particles. Although PSO has typically been used with swarms of about 50 particles [3], there is evidence that larger swarm sizes may improve the rate of convergence for some problems [31, 29]. The inescapable realities of parallel computing suggest that the issue of swarm size should be revisited. In this paper, we consider swarm size first from a general perspective and then in the context of parallelization.

Section 3.2 addresses the issue of swarm size independent of parallelization. The No Free Lunch Theorems for Optimization [35] hint that regardless of whether PSO is performed in serial or parallel, an increased swarm size might be beneficial for some objective functions while a decreased swarm size might be better for others. Indeed, tuning the swarm size can provide dramatic improvements by changing the tradeoff between exploration and exploitation. For example, PSO is more effective with large swarms for highly multimodal functions and with small swarms for smooth functions.

Section 3.3 considers swarm size and topology for parallel PSO, where the benefits of large swarms are particularly pronounced. In this section, objective functions are classified according to their effects on parallelization. For expensive optimization problems, discussed in Section 3.3.1, the cost of performing the function evaluation outweighs the cost of communication. In this case, a sparse swarm topology would not be significantly more efficient than a dense topology. In contrast, swarm topology can be an important issue for inexpensive optimization problems. Sections 3.3.2 and 3.3.3 discuss the case of inexpensive optimization problems with sparse and dense topologies respectively.

3.2 Topology and Swarm Size in PSO

3.2.1 Particle Swarm Optimization

Particle Swarm Optimization simulates the motion of particles in the domain of an objective function. These particles search for the optimum by evaluating the function as they move. During each iteration of the algorithm, the position and velocity of each particle are updated. Each particle is pulled toward the best position it has sampled, known as the *personal best*, and toward the best position of any particle in its neighborhood, known as the *neighborhood best*. This attraction is weak enough to allow exploration but strong enough to encourage exploitation of good locations and to guarantee convergence.

Constricted PSO is generally considered the best variant [3]. Each particle's position $\boldsymbol{x}_0$ and velocity $\boldsymbol{v}_0$ are initialized to random values based on a function-specific feasible region. During iteration t, the following equations update the i^{th} component of a particle's position $\boldsymbol{x}_t$ and velocity $\boldsymbol{v}_t$ with respect to the personal best $\boldsymbol{p}_{t-1}$ and neighborhood best $\boldsymbol{n}_{t-1}$ from the preceding iteration:

$$v_{t,i} = \chi \left[v_{t-1,i} + \phi^P u^P_{t-1,i}(x^P_{t-1,i} - x_{t-1,i}) + \phi^N u^N_{t-1,i}(x^N_{t-1,i} - x_{t-1,i}) \right] \tag{3.1}$$

$$x_{t,i} = x_{t-1,i} + v_{t,i} \tag{3.2}$$

where x^P is the personal best, x^N is the neighborhood best, ϕ^P and ϕ^N are usually set to 2.05, $u^P_{t,i}$ and $u^N_{t,i}$ are samples drawn from a standard uniform distribution, and the constriction constant $\chi = \frac{2}{|2-\phi-\sqrt{\phi^2-4\phi}|}$ where $\phi = \phi^P + \phi^N$ [6].

The neighborhoods within a particle swarm are defined by the swarm topology, also known as the sociometry. The choice of topology can have a significant effect on the performance of PSO [21]. Topologies also determine the amount of communication between particles, which is especially important for parallel implementations of PSO. Topologies are discussed in greater detail in Section 3.2.2.

3.2.2 Topology Representation

The topology of a swarm indicates how information is communicated between particles. The most typical representation of swarm topology is an undirected graph, where each vertex is a particle. Swarm topologies are usually defined informally, using a combination of prose and diagrams. Although insightful, this approach is inherently imprecise and does not scale to more complex and dynamic topologies. In particular, the concept of a neighborhood is ambiguous when extended to directed graphs. In the "informants" interpretation, the neighborhood indicates which neighbors contribute their personal bests to a particular particle's neighborhood best. In the "message sending" interpretation, it indicates which neighbors a particular particle sends its personal best to. This ambiguity makes it more difficult to discuss how topologies affect communication, which has a significant cost in parallel PSO,

To address the drawbacks of defining topologies informally, we will build on the formal definitions from graph theory. A *directed graph* is a pair $G = (V, E)$, where V is a nonempty finite set and E is a (possibly empty) subset of $V \times V$. The elements of V are the *vertices* of G and the elements of E are the *edges* (or arcs) of G. An edge (u, v) indicates a link from vertex u to vertex v [22].

The *topology* in iteration i is a directed graph $T_i = (P_i, E_i)$ where the vertex set $P_i = \{p_0, p_1, \ldots, p_{n-1}\}$ is the set of particles. Note that the topology in one iteration may be different from the topology in another iteration as particles and edges are added or removed. In a *static topology*, the topology is the same in all iterations, while in a *dynamic topology*, the topologies may vary between iterations.

The *neighborhood* of a particle p in iteration i is the set $N_{i,p} = \{p_j \in P_i \mid (p, p_j) \in E_i\}$. In other words, it is the set of particles to which p sends its personal best. This representation is preferable to the "informants" model because it makes the communication more explicit. A particle may be a member of its own neighborhood, meaning that it considers its own personal best when updating its neighborhood best. Using the definition of a neighborhood, the edge set is constructed by combining the individual neighborhoods according to the equation $E_i = \bigcup_{p \in P} N_{i,p}$.

Many simple topologies can be more generally and succinctly described by a neighborhood function than by a verbose listing of the edge set. A *neighborhood function* ν is a function of a particle index and swarm size that gives a set of indices indicating the neighbors. The neighborhood N_{i,p_j} of the particle p_j in iteration i is related to the neighborhood function ν by the equation $N_{i,p_j} = \bigcup_{k \in \nu(j,n)} p_k$.

3.2.3 Common Topologies

The complete topology K_n is a static topology of n particles where each particle sends its personal best to all other particles in the swarm. This topology is also known by the names fully connected, gbest, global topology, or star[1]. A complete topology is described by the neighborhood function:

$$\nu_K(i, n) = \{0, 1, \ldots, n - 1\} \tag{3.3}$$

The ring topology $Ring_{n,1}$ is a static topology with n particles where each particle sends its personal best to itself and one neighbor on either side. The ring topology is also known as lbest or local topology. This topology has been generalized as $Ring_{n,k}$, where each particle sends its personal best to k neighbors on each side, although the $Ring_{n,1}$ variant is

[1]The word "star" has been used historically in the PSO community for K_n. We and others feel that this choice is unfortunate because graph theory uses the term to denote the complete bipartite graph $K_{1,k}$.

more common. The neighborhood function for $Ring_{n,k}$ is:

$$\nu_{Ring}(i, n, k) = \{(i - k) \bmod n, \ldots, i, \ldots, (i + k) \bmod n\}$$

Static topologies may be randomly generated with irregular connections between vertices. A study of 3,289 generated topologies found that the average vertex degree and the level of clustering were the graph statistics with the greatest effect on swarm performance [21].

In addition to fixed topologies, there are also many dynamic and adaptive topologies, including Randomized Directed Neighborhoods [24] and Dynamic Multi-Swarm [17]. This paper does not focus on adaptive topologies, such as TRIBES [5], because they tend to require global state, which is difficult to parallelize. The concept of islands with migration, inspired by work in Genetic Algorithms, leads to parallelizable adaptive topologies [27, 1, 4].

The random topology $Rand_{n,k}$ is a dynamic topology where each particle randomly picks k different neighbors each iteration. Its neighborhood function is:

$$\nu_{Rand}(i, n, k) = \{i, U_1, U_2, \ldots, U_k\}$$

where U_j is a uniform random integer between 0 and $n - 1$. These random numbers are drawn independently in each iteration and for each particle.

Topologies may be characterized both by size and by the level of connectedness. A dense topology such as K_n results in faster propogation of the neighborhood best, while a sparse topology such as $Ring_{n,1}$ communicates this information more slowly [26]. As a result, sparse topologies tend to exhibit more explorative behavior while dense topologies tend to be more exploitative [21].

3.2.4 Selecting an Appropriate Swarm Size

The ideal topology and swarm size for Particle Swarm Optimization depend on the objective function. Researchers have devised various benchmark functions and have found that the ideal topology for one function may perform very poorly for another function. The No Free Lunch Theorems for Optimization show that this is true in general—if an algorithm performs well on average for one class of functions then it must do poorly on average for other problems [35].

An attempt to standardize PSO found that although K_{50} converged to the global optimum more quickly than $Ring_{50,1}$ for many benchmark functions, it was also more likely to permaturely converge to local optima for other functions. The study found no significant improvement for any other swarm size between 20 and 100 and concluded with a recommendation to use $Ring_{50,1}$ as a starting point. The authors acknowledged that choosing the ideal topology requires thorough experimentation for the particular problem [3]. Other authors have explained that a large swarm with a sparse topology propogates information slowly [26].

The results in this section show the relationship between swarm size and the tradeoff between exploration and exploration. The Sphere, Griewank, and Rastrigin benchmark functions are representative of three different types of functions. Sphere is unimodal and smooth. It is most easily optimized when information flows between particles as quickly as possible. Although the $Ring_{n,1}$ topology will eventually converge, the K_n topology is much more efficient. Griewank has deceptive local minima, although if swarms avoid being trapped, they eventually find the global minimum. Rastrigin is much less smooth, and for high-dimensional problems, PSO is extremely susceptible to premature convergence.

The remainder of this section reviews the behavior of various sizes of swarms with the K_n and $Ring_n$ topologies. Performance is evaluated with respect to the Sphere, Griewank, and Rastrigin benchmark functions. All plots represent the average over at least 20 runs. PSO performance is usually measured by running the algorithm multiple times and plotting the best value attained (averaged among the runs) against the cumulative number of function evaluations. To compare a wide range of swarm sizes, we set a fixed number of function evaluations and show the average value reached by PSO with the given computational budget.

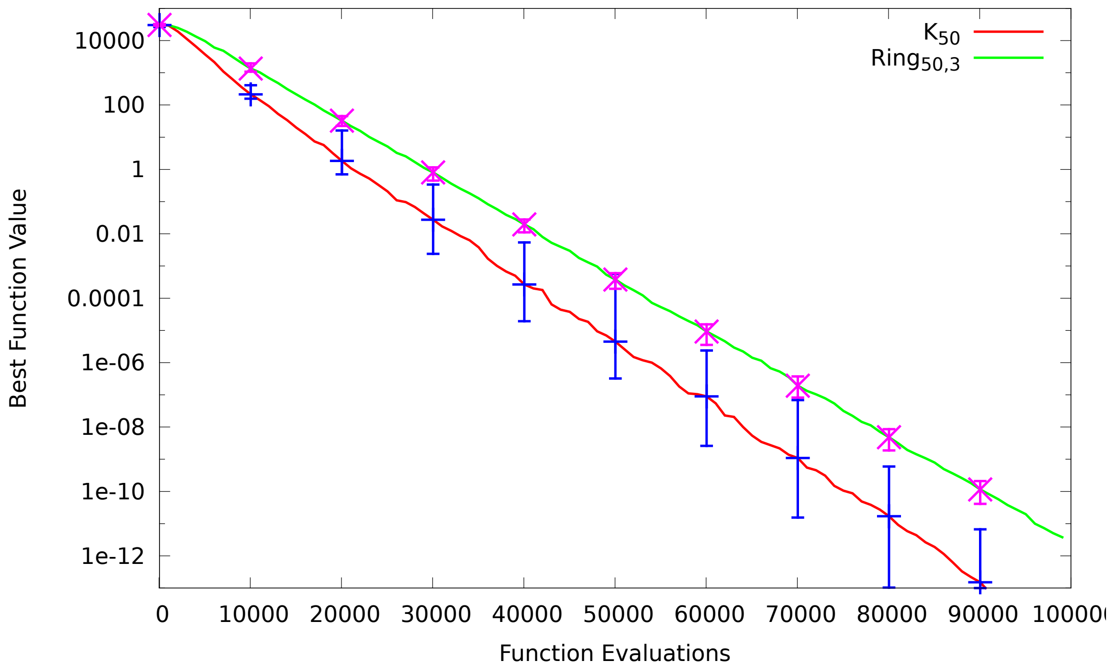

Figure 3.1: PSO performance on Sphere for conventional-sized K_{50} and $Ring_{50,1}$ swarms (the error bars show percentiles 10, 50, and 90, and the y-axis represents the log of the best function value). The complete topology is more exploitative than the ring topology and performs better for the unimodal Sphere function.

For some objective functions, such as Griewank, once a particle enters a smooth region around the global optimum, the swarm quickly converges. Thus, in a series of repeated runs, the average does not represent the behavior of the algorithm well because it is skewed by a single successful swarm. For such functions, it is more enlightening to plot the percentage of the runs in which the swarm reached a given threshold [21].

3.2.5 Sphere

The simplest benchmark function is the *Sphere* or parabola, expressed by the function $f_S(\boldsymbol{x}) = \sum_{i=1}^{D} x_i^2$. Particles are initialized in the interval $[-50, 50]^D$. For the 50-dimensional variant, Figure 3.1 shows the best value in the swarm at each iteration of PSO with both K_{50} and $Ring_{50,3}$. The K_{50} swarm is more effective because its particles share information as quickly as possible, while information in $Ring_{50,3}$ takes time to propogate from one particle

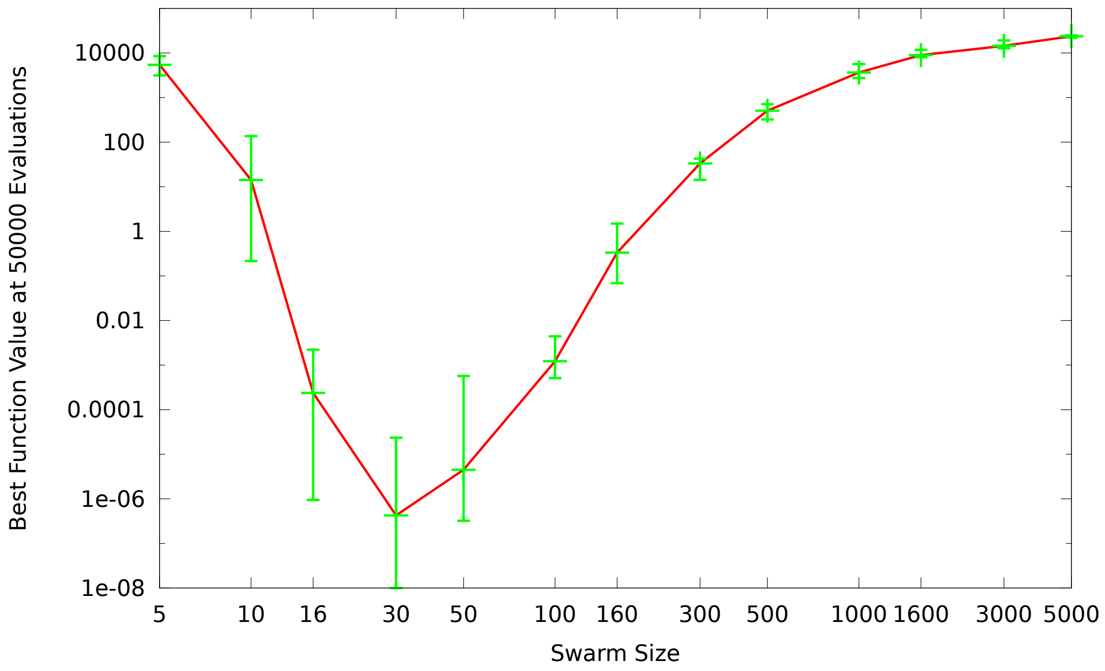

Figure 3.2: PSO performance on Sphere for K_n with increasing swarm sizes (with a log scale for both axes). The number of function evaluations per iteration grows with the number of particles, so increasing the number of particles decreases the number of iterations performed.

to the next. Note, however, that the K_{50} swarm exhibits greater variability as a result of this information flow.

Figure 3.2 shows the behavior of complete swarms relative to function evaluations as the number of particles increases. Larger swarms exhibit less variability, but increased swarm sizes do not improve performance for Sphere.

Figure 3.3 demonstrates that the ideal swarm size increases with the dimensionality of the objective function. For 20-dimensional Sphere, the ideal swarm size is about 16, but for 100-dimensional Sphere, the 50 particle swarm was most effective. For smooth unimodal objective functions such as Sphere, smaller swarms, which are more exploitative than explorative, are generally more efficient than larger swarms. Figure 3.4 compares two different swarms for 20-dimensional Sphere. Although 50 particles is the ideal swarm size for 100-dimensional Sphere, the 16-particle swarm achieves comparable values for 20-dimensional Sphere in less than half as many function evaluations as the 50-particle swarm.

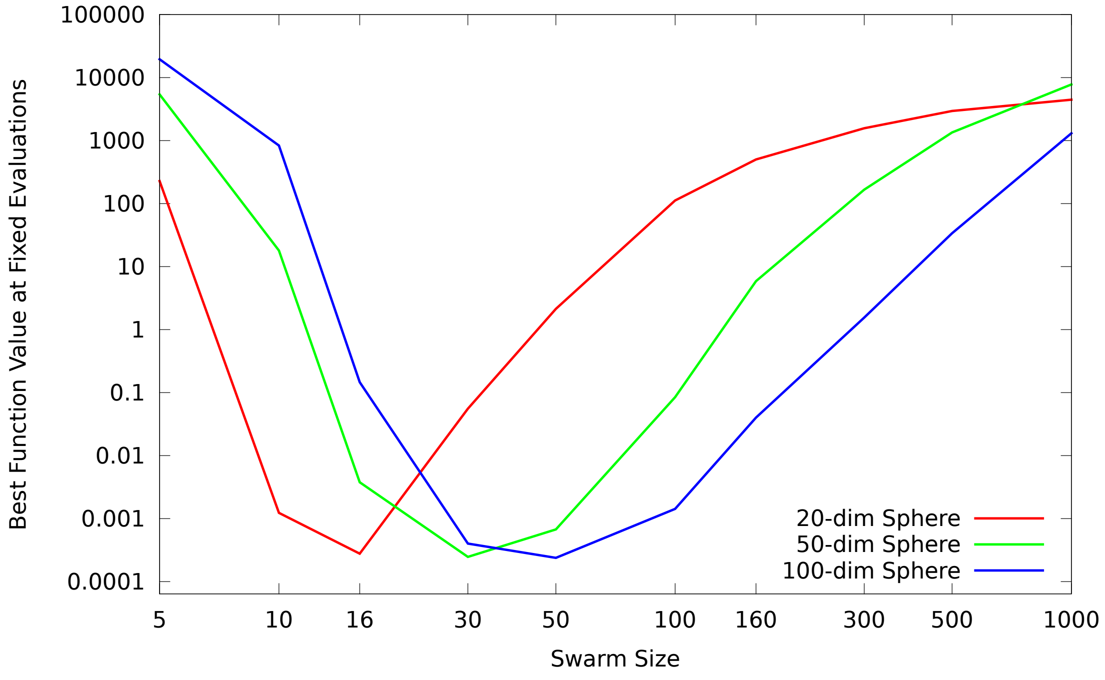

Figure 3.3: The effect of increasing swarm sizes as the problem dimensionality varies. The fixed number of evaluations for each problem (6,000, 38,000, and 180,000 respectively) was chosen to conveniently scale the plot. All three curves have the same basic shape, but each objective function has its own ideal swarm size.

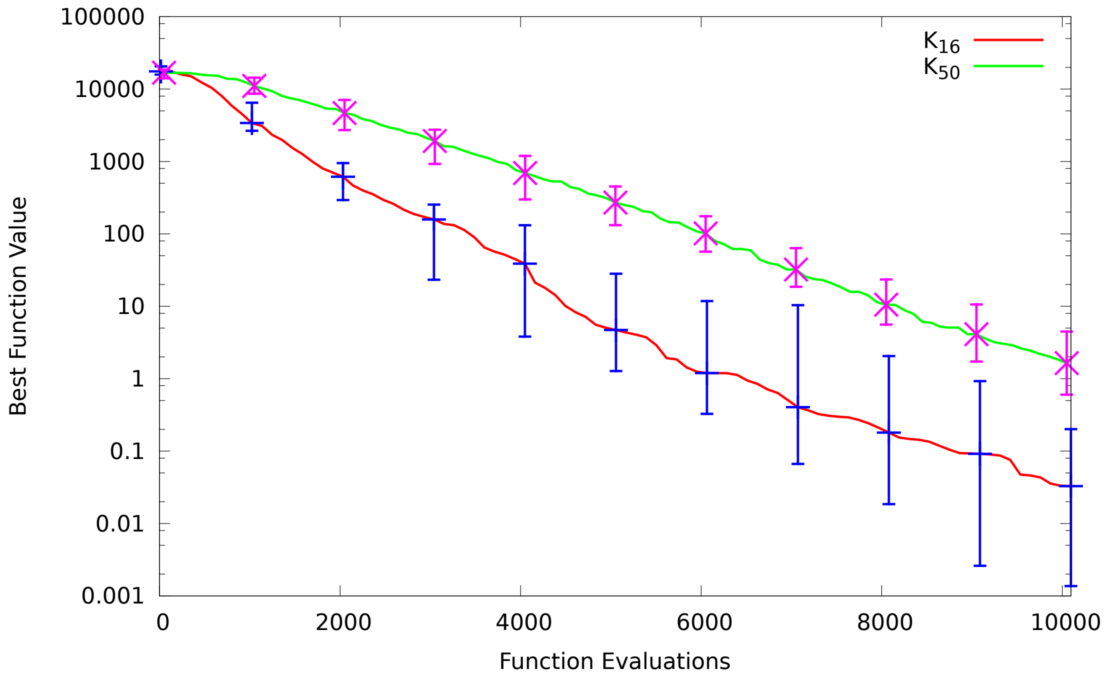

Figure 3.4: For 20-dimensional Sphere, PSO is much more effective with small swarms.

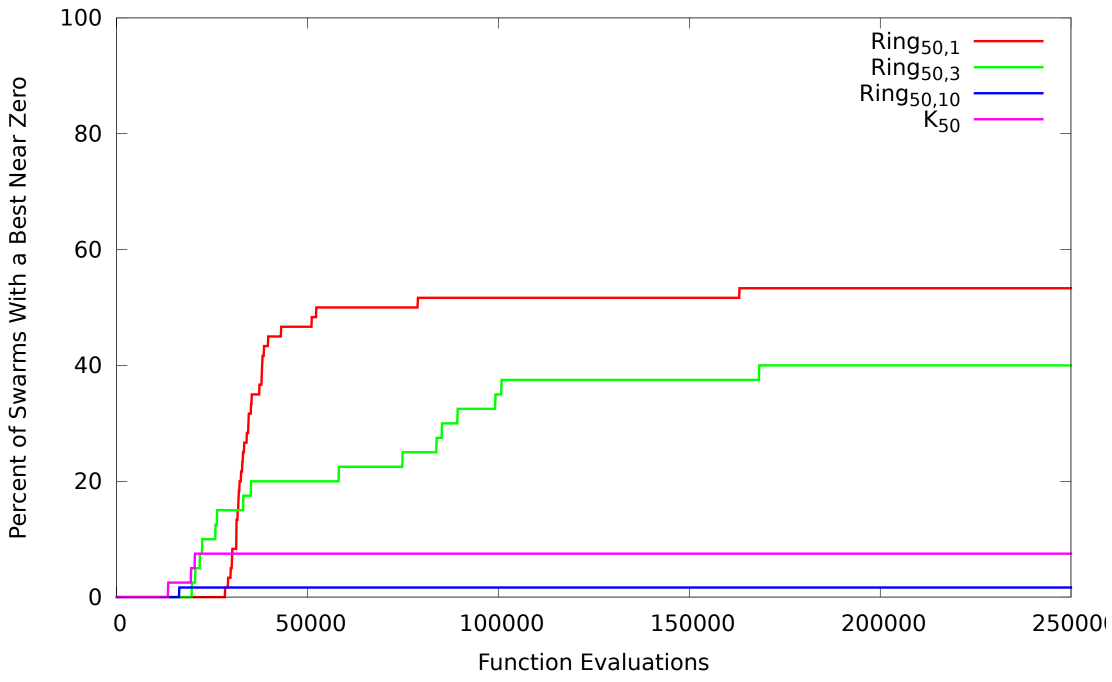

Figure 3.5: PSO performance on Griewank for conventional-sized K_{50} and $Ring_{50,k}$ swarms. The highly connected swarms tend to get stuck in local minima because information flows too quickly.

3.2.6 Griewank

The *Griewank* benchmark function is defined by the equation $f_G(\boldsymbol{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{D} x_i^2 - \Pi_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right)$. We use the 15-dimensional variant with the feasible region $[-600, 600]^{15}$ (Griewank is more challenging with fewer dimensions than with more dimensions). Griewank has deceptive local minima, but if swarms avoid getting trapped, they tend to quickly converge to the global minimum. As described in Section 3.2.4, this is plotted as the percentage of runs in which the algorithm reached a given threshold, in this case, 10^{-6}. As shown in Figure 3.5, the K_n topology is highly susceptible to premature convergence, and PSO is less likely to converge prematurely in a more loosely linked topology.

Although it performs better than K_{50}, even $Ring_{50,1}$ tends to get stuck in local minima. Figure 3.6 shows that adding more particles increases exploration, and $Ring_{n,1}$ does not prematurely converge for any $n \geq 500$. Although most swarms of size 160 converged, they did not do so consistently. Swarms with greater than 500 particles are less efficient

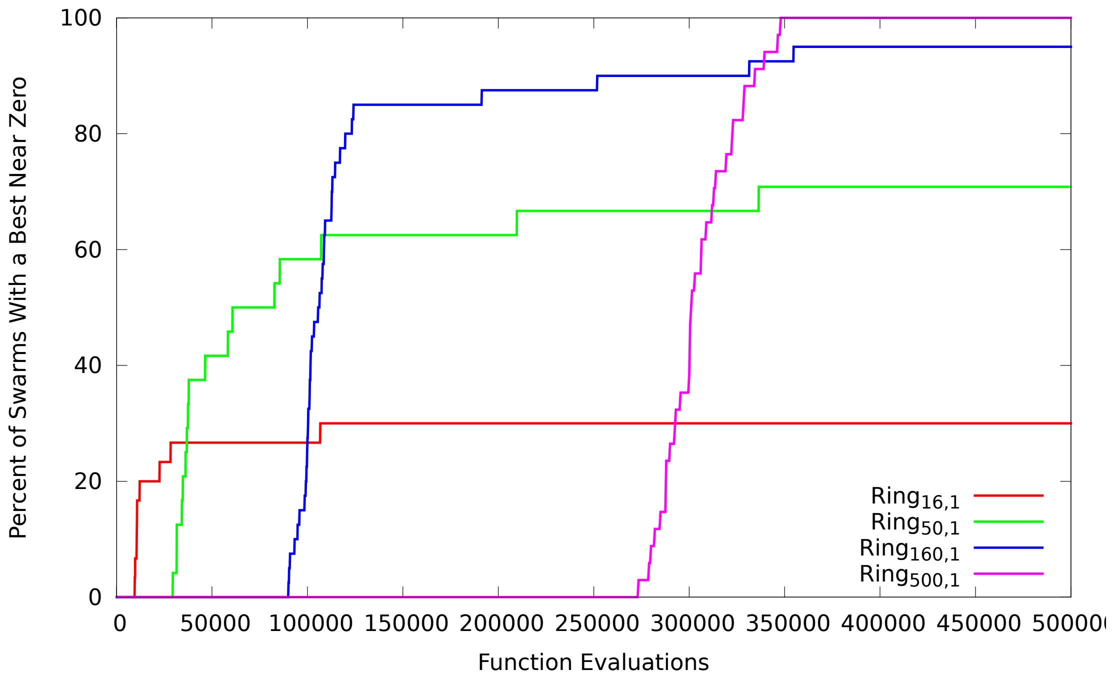

Figure 3.6: PSO performance on Griewank for $Ring_{n,1}$ with increasing swarm sizes. Larger swarms are more explorative and less likely to get stuck in local optima.

because they require many more evaluations per iteration. For serial PSO, the swarm size should be large enough to avoid premature convergence but no larger.

Similar experiments with more densely connected ring topologies showed dramatically worse performance. Apparently information must fundamentally move slowly through the swarm for PSO to successfully optimize Griewank.

3.2.7 Rastrigin

The more complicated *Rastrigin* function is given by $f_R(\boldsymbol{x}) = \sum_{i=1}^{D} (x_i^2 - 10\cos(2\pi x_i) + 10)$. We use the 50-dimensional variant with the feasible region $[-5.12, 5.12]^{50}$. For Rastrigin, PSO is sensitive to the speed of information flow through the swarm, and the ideal amount of communication changes with the number of dimensions. With the 30-dimensional variant of Rastrigin, as with Sphere, the ring topology is less effective because information propogates slowly, and the best topology is K_n. However, with a higher-dimensional problem, such as the 50-dimensional Rastrigin shown in Figure 3.7, the complete topology exhibits high

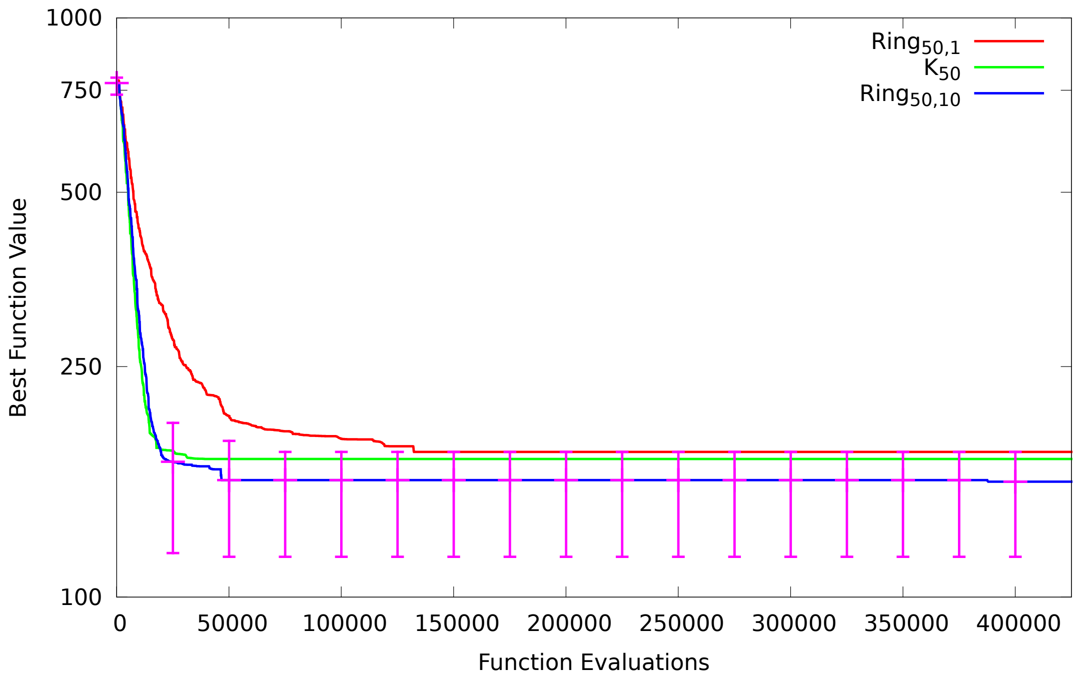

Figure 3.7: PSO performance on Rastrigin for conventional-sized $Ring_{50,1}$, K_{50}, and $Ring_{50,10}$ swarms (for clarity, only the last is shown with percentiles 10 and 90). The moderately connected $Ring_{50,10}$ outperforms the sparse $Ring_{50,1}$ and K_{50} topologies.

variability and is susceptible to premature convergence. In this case, PSO performs better with the $Ring_{50,10}$ topology than with either the dense K_{50} topology (with fast information flow) or the sparse $Ring_{50,1}$ (with slow information flow). With 50-dimensional Rastrigin, we generally use the $Ring_{n,\frac{n}{20}}$ topology.

Regardless of the topology, all of the 50-particle swarms are trapped in local minima. There are many variants of PSO that address the problem of premature convergence by encouraging exploration. Figure 3.8, which shows the performance of PSO with the $Ring_{n,\frac{n}{20}}$ topology for various values of n, demonstrates that simply adding particles can increase exploration. Since a smaller swarm can do more iterations with fewer function evaluations, it initially outperforms a larger swarm. However, it converges to a local optimum and gets passed up by the larger swarm. Even though adding particles reduces the number of iterations performed, the increased exploration leads to an overall improvement for Rastrigin. In contrast to the Sphere function, discussed in Section 3.2.5, setting swarm size beyond the

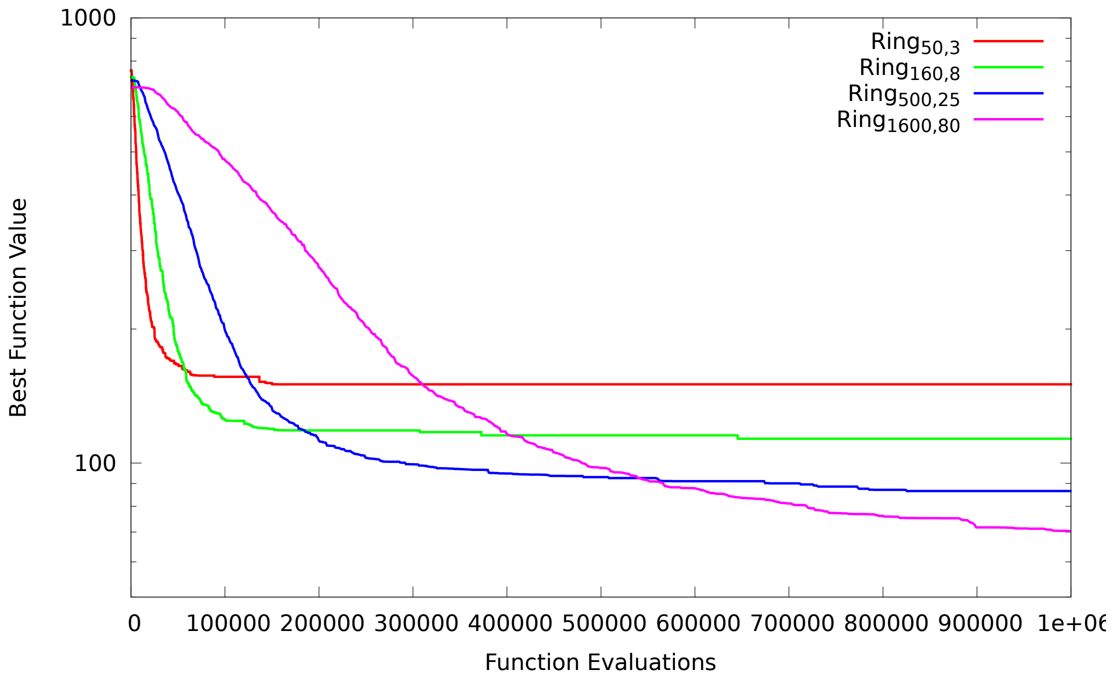

Figure 3.8: PSO performance on Rastrigin for $Ring_{n,\frac{n}{20}}$ with increasing swarm sizes.

traditional range improves performance on Rastrigin, even in a sequential implementation of PSO.

3.3 Topology and Swarm Size in Parallel PSO

In Section 3.2, we discussed topology and swarm size in general. However, parallelization gives a different perspective. After a brief review of parallel PSO and communication, we will discuss how these issues affect the choice of topology and swarm size.

In order for an algorithm to be parallelized, its operations must be decomposed into tasks [11]. Particle Swarm Optimization may be decomposed into a task for each function evaluation [10]. In other words, there is one task per particle per iteration. Other strategies may be possible, such as decomposition of the objective function itself and speculative decomposition [11, 9]. However, we will refer to the common case as parallel PSO. Parallel PSO has been applied to many applications such as antenna design [12] and biomechanics [16].

The two most successful approaches to parallel PSO are the synchronous variant, which produces the exact same results as the standard algorithm, and the asynchronous variant, which improves parallel performance by loosening the requirement that particles iterate in lockstep. Note that variations in tuning parameters and topologies may also affect parallel performance but may be applied to both synchronous and asynchronous parallel PSO.

Synchronous parallel implementations of PSO reproduce the standard serial algorithm exactly. This approach was first described analytically by Belal and El-Ghazawi [1] and first implemented by Schutte et al. [31]. In a typical master-slave algorithm, the master assigns tasks to slave processors, and in parallel PSO, each task consists primarily of a function evaluation. Updating the particle's position and value may also be included in the task [1], or this work may be performed in serial by the master [31]. Before proceeding to the next iteration, particles communicate, and each particle updates its neighborhood best. Whether this communication step happens sequentially on the master or in parallel, each particle must receive communication from its neighbors before proceeding. The benefits of the synchronous PSO include its simplicity, repeatability, and comparability with standard PSO, which may be essential in research applications.

Asynchronous parallel PSO [34, 16] is a modification to the standard algorithm which removes the synchronization point at the end of each iteration. Instead, particles iterate independently and communicate asynchronously. In a typical master-slave implementation of asynchronous parallel PSO, the master updates each particle's personal best, neighborhood best, velocity, and position immediately after receiving the function value from the slave processor. Since this update occurs while other particles are still being evaluated, it may use information from the previous iteration for some neighbors.[2] In a partially asynchronous implementation, particles might wait for some but not all neighbors to complete before

[2]Asynchronous parallel PSO has been compared to the "asynchronous updates" variant of serial PSO [16]. However, serial PSO with asynchronous updates differs from standard PSO in that particles use newer information, but asynchronous parallel PSO differs from standard PSO in that particles use older information.

proceeding [32]. In some master-slave implementations, particles never get more than one iteration ahead of others [34, 16]. However, in a fully distributed implementation, particles might never wait for information, and one particle could complete many more iterations than another particle [33]. The main effect of asynchronous evaluation is that processors spend less time idle—this trait is particularly valuable when processors are heterogeneous or function evaluation times are varied [34, 16]. Asynchronous parallel PSO behaves differently than the standard algorithm and may even produce different results between runs. Most reports conclude that asynchronous communication produces similar numerical results to the standard algorithm, but the question has not yet been thoroughly addressed [34, 16].

The ideal swarm topology in parallel PSO may depend on a variety of factors, including the number of processors available, the amount of time required for each evaluation of the benchmark function, and the costs associated with communication. For the sake of repeatability, the results in this paper are produced by synchronous parallel and serial implementations of PSO, but the discussions are relevant for both synchronous and asynchronous parallel PSO. For example, even though asynchronous parallel PSO avoids blocking for communication, slow communication can make the algorithm require more iterations to converge [32].

PSO algorithms are usually evaluated on their performance with respect to the number of function evaluations as in Section 3.2, but this is not appropriate for parallel PSO, where function evaluations are performed concurrently. In fact, parallel performance is highly implementation-dependent. Although specific implementations of parallel PSO may be evaluated by criteria such as speedup or wallclock time per iteration, this paper focuses on the implementation-independent effects of topology and swarm size on PSO. In this section, PSO performance is evaluated with respect to the number of iterations rather than the number of function evaluations, and topologies are characterized by the number of messages between particles per iteration. This information determines the parallel performance when combined with implementation-specific details such as the time required for each function evaluation, the number of processors, the implementation overhead per iteration, and the communication overhead per message. Note that the performance of PSO with respect to the number of iterations is independent of the specific implementation, and it is in fact more efficient to perform a series of repeated experiments by parallelizing sequential executions of PSO than by running a parallel implementation.

Section 3.3.1 discusses swarm topology in the situation where function evaluations are expensive relative to communication. This case occurs when communication costs are inexpensive, such as in a centralized master-slave implementation, or when the objective function takes a long time to evaluate. Section 3.3.2 and 3.3.3 consider the behavior of parallel PSO in cases where function evaluations are inexpensive relative to communication, such as in a distributed implementation of PSO.

3.3.1 Expensive Function Evaluations

Section 3.2 showed how large swarm sizes can improve the convergence of PSO even with respect to the number of function evaluations. However, in a parallel implementation of PSO with an expensive objective function, the time used for evaluating the function outweighs the overhead from communication. If the number of particles is the same as the number of processors, then the time needed to perform a full iteration of parallel PSO is approximately the same as the time of a single function evaluation. The precise wallclock time of each iteration is implementation-specific, but even in an implementation with high communication overhead, this approximation is reasonable for expensive objective functions [20].

In this section, we are not concerned with implementation-specific run times, but rather with the effects of topology and swarm size from the perspective of parallelization. As there are inexpensive functions with complex landscapes and expensive functions with simple landscapes, the behavior of PSO with respect to the function is the main issue. For this purpose, benchmark functions are a useful and efficient tool for understanding the effects of PSO with expensive objective functions even though the benchmark functions are

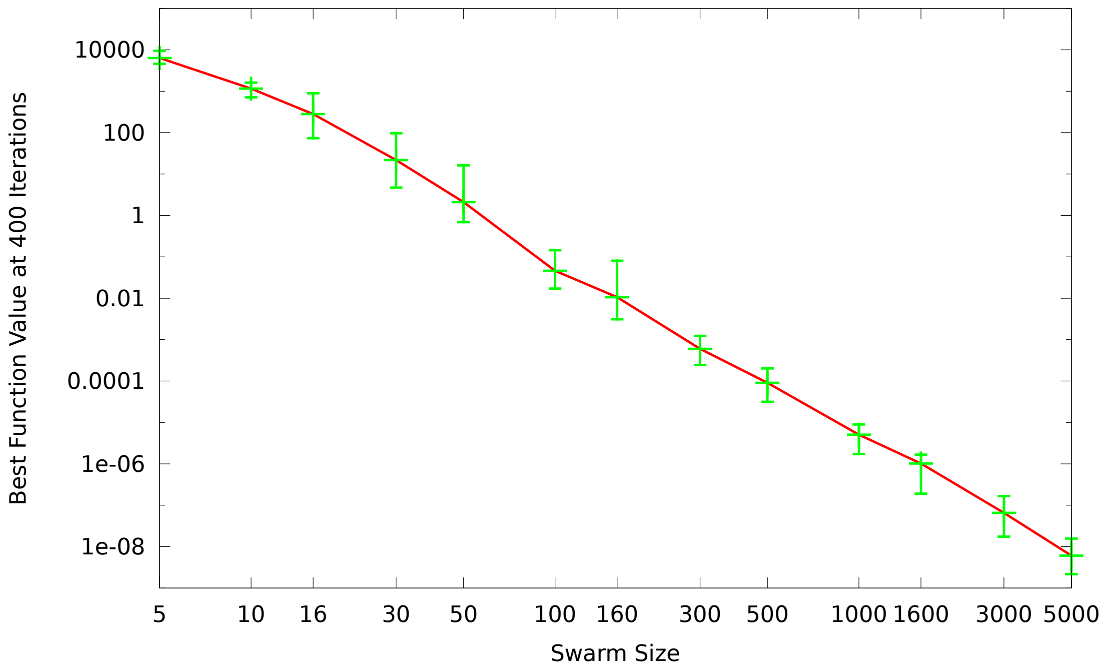

Figure 3.9: Performance of PSO on Sphere after 400 iterations for K_n with various swarm sizes.

themselves inexpensive. For example, a plot of performance with respect to iterations for a smooth unimodal function with five minutes per evaluation would be similar to that of Sphere.

The Sphere benchmark function is extreme in that it has no deceptive local minima, so it does not particularly benefit from exploration. In Section 3.2.5, Figure 3.2 showed that for 50-dimensional Sphere, the ideal swarm size in serial PSO is around 30. However, Figure 3.9 plots the best function value after a fixed number of iterations instead of after a fixed number of function evaluations. This demonstrates that if expensive evaluations are performed in parallel, adding particles always improves performance, even for an extreme function like Sphere. Note that this is not necessarily true for objective functions with low computational cost, where overhead from communication may negate this benefit.

When judged by function evaluations, as in Sections 3.2.6 and 3.2.7, small swarms converged prematurely for the Griewank and Rastrigin functions. In parallel PSO, where performance is plotted against iterations instead of function evaluations, the benefit of large

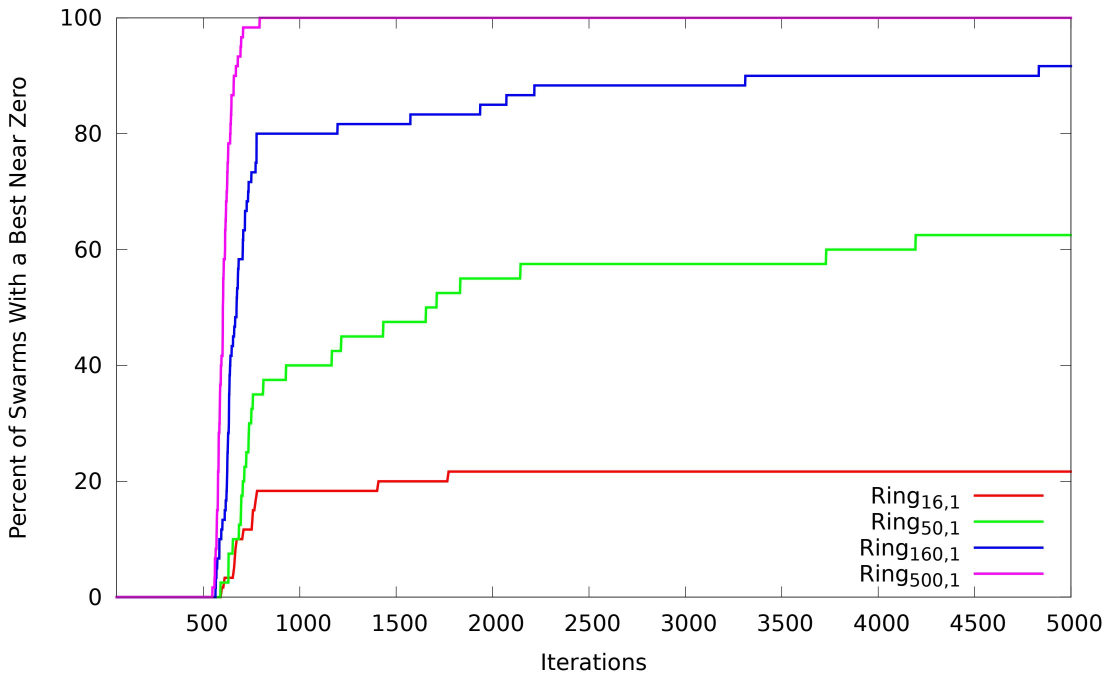

Figure 3.10: Success rate on Griewank with respect to iterations for $Ring_{n,1}$ with various swarm sizes.

swarms is even more pronounced. For Griewank, compare Figure 3.6 (serial) to Figure 3.10 (parallel). For Rastrigin, compare Figure 3.8 (serial) to Figure 3.11 (parallel). These figures show that in a parallel environment, large swarms are not only less prone to premature convergence, but they also attain comparable values in fewer iterations.

3.3.2 Inexpensive Function Evaluations With Sparse Topologies

In a parallel implementation of PSO, communication overhead may significantly affect performance. For functions that take a long time to evaluate, this communication may be negligible. However, for more easily evaluated functions, limiting communication may be critical for performance [20].

Functions such as Griewank, which is best optimized with $Ring_{n,1}$, inherently demand low communication. The ring topology only needs to send 2 messages per particle for a total of $2n$ messages per iteration. Since the communication overhead is naturally low for such functions, they are naturally efficient in a parallel implementation of PSO.

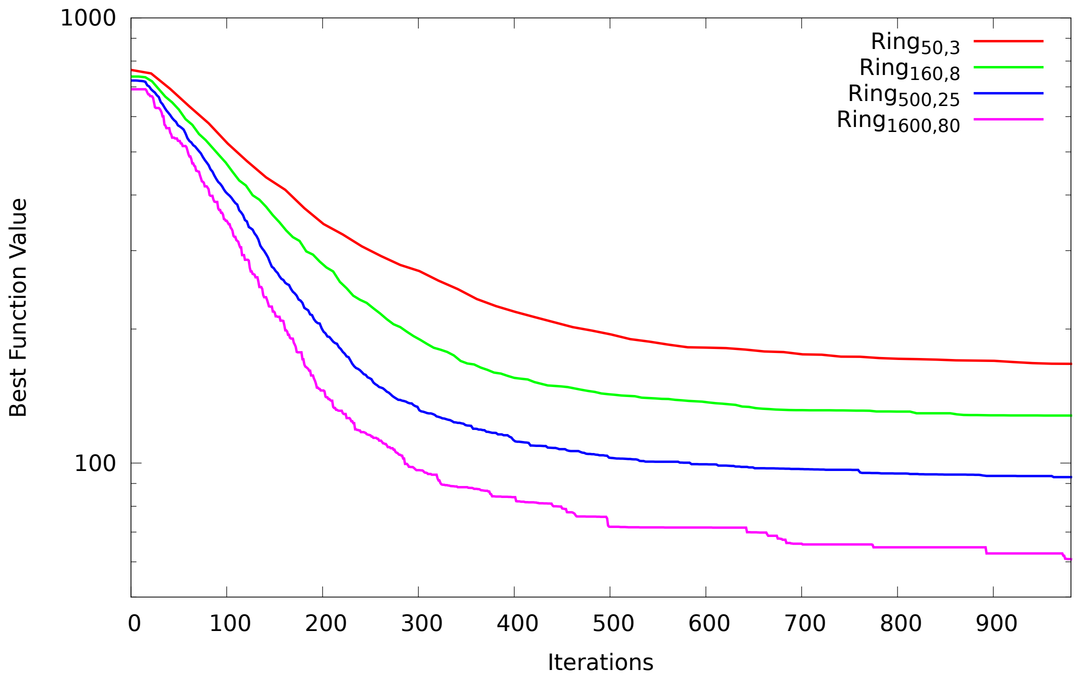

Figure 3.11: PSO performance on Rastrigin with respect to iterations with various topologies.

Another way to parallelize PSO is to perform independent runs, or subswarms, on different processors and take the best result after the runs have completed. Formally, $Subswarms_{n,k}$ is a static topology consisting of n independent components of k fully-connected particles. This requires no modification of the PSO implementation and works surprisingly well. If local communication is inexpensive, then the subswarms topology is extremely efficient because it involves no external communication.

"Communicating subswarms" occasionally share neighborhood bests between subswarms. If external communication is expensive, this can be effective because messages are not sent every iteration. The communicating subswarms topology $CommunicatingSwarms$ is dynamic: it is $Subswarms_{n,k}$ during most iterations but is K_{nk} every ith iteration. More sophisticated ways to connect multiple subswarms have been proposed [13, 17], but communicating subswarms are particularly simple to parallelize.

Figure 3.12 shows two $CommunicatingSwarms$ topologies with 4000 total particles in comparison with related K_n swarms (K_{400} is the same as $CommunicatingSwarms_{1,400}$

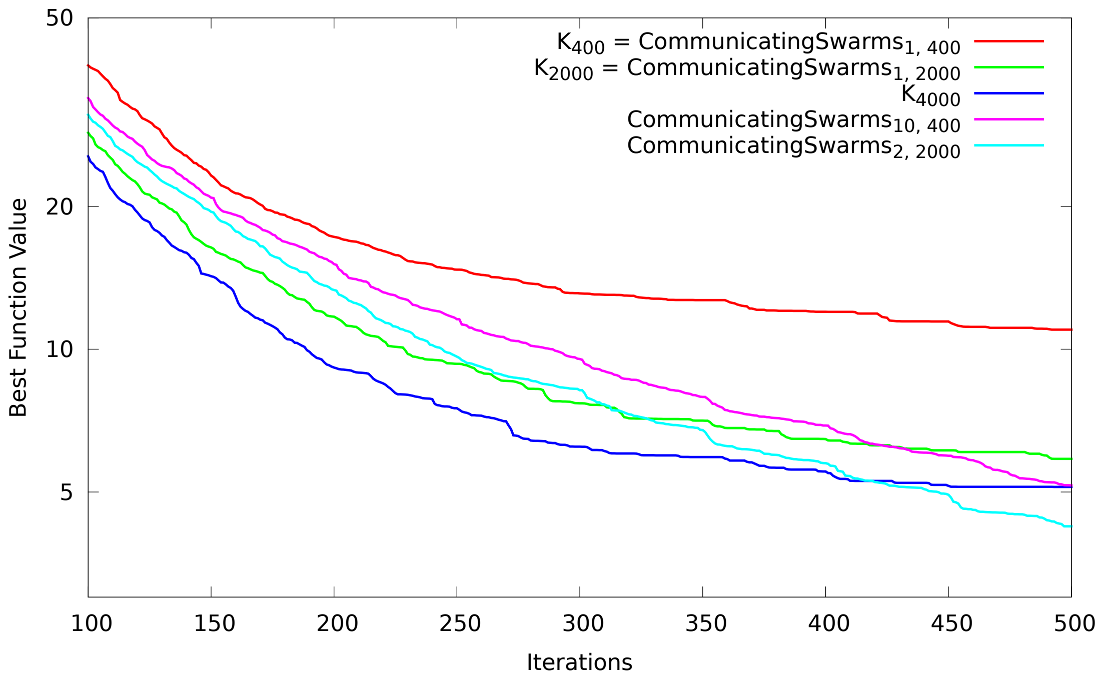

Figure 3.12: PSO performance on 20-dimensional Rastrigin with two different communicating subswarms topologies of 4000 particles. Unsurprisingly, increasing the number of subswarms improves performance, but the $CommunicatingSwarms_{10,400}$ and $CommunicatingSwarms_{2,2000}$ are even promising compared to the K_{4000} swarm with the same total number of particles.

and K_{2000} is the same as $CommunicatingSwarms_{1,2000}$). These subswarms communicate every 50 iterations, and as a result, there is an interesting dip after every 50th iteration. Unsurprisingly, communicating subswarms outperform a single swarm of the smaller size. Interestingly, a few small subswarms even seem promising in comparison with a single large swarm with the same total number of particles.

3.3.3 Inexpensive Function Evaluations With Dense Topologies

Functions like Sphere and low-dimensional Rastrigin are best optimized when information flows more quickly through the swarm. In this section, 20-dimensional Rastrigin is used as an example of a functions where PSO performs best with greater communication. Unfortunately, using a fully connected K_n topology meets this need but requires $n-1$ messages per particle for a total of $n^2 - n$ messages per iteration. In this section, we will focus on reducing

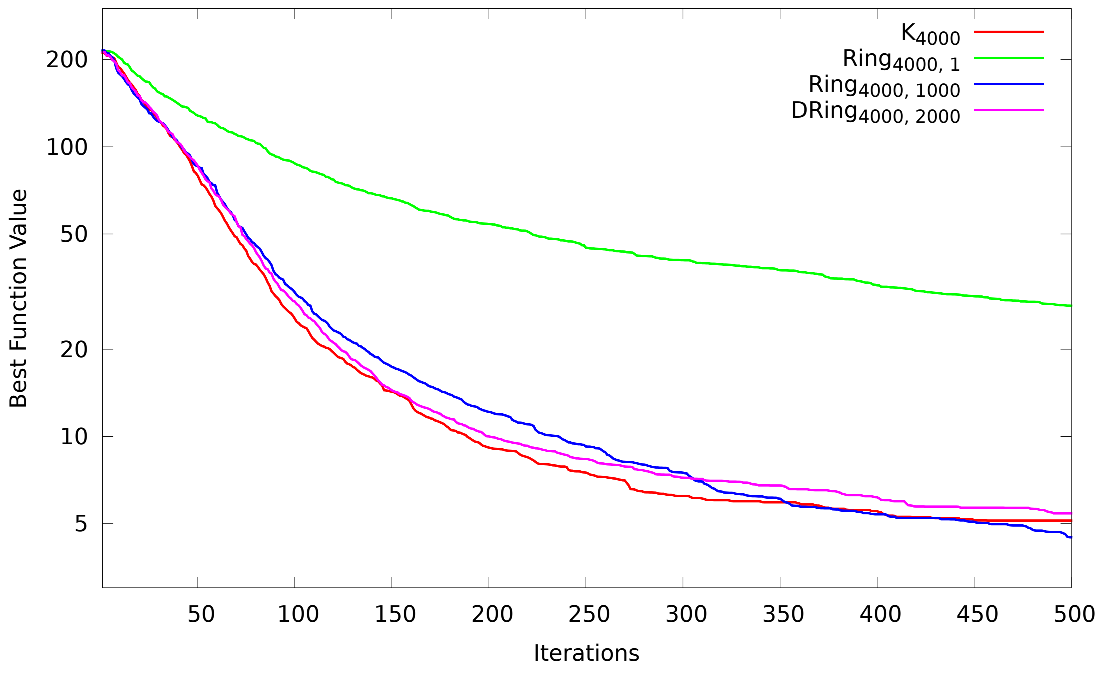

Figure 3.13: PSO performance on low-dimensional Rastrigin with ring topologies.

communication for such functions when communication is expensive or function evaluation is inexpensive.

The $Ring_{n,k}$ topology requires $2kn$ messages per iteration. If $k = \lceil \frac{1}{2}(n-1) \rceil$ (i.e., on each side, a particle communicates with half of the swarm), then $Ring_{n,k}$ is identical to K_n. The k parameter can be tweaked to compromise the tradeoff between information flow and communication overhead. Figure 3.13 shows a low-dimensional Rastrigin with a few ring topologies, demonstrating that communication can be halved without affecting performance.

Section 3.2.2 describes the random topology, in which each particle randomly picks k different neighbors each iteration. The $Rand_{n,k}$ topology has kn messages per iteration. When k is large, particles send messages to most other particles in the swarm, making the topology similar to K_n. A topology called stochastic star, which differs primarily by using the informants model in its definition, also has this similarity to K_n [23].

Recall that in a dynamic topology, a particle's neighborhood may change between iterations. The PSO algorithm is ambiguous about how to update a particle's neighbor-

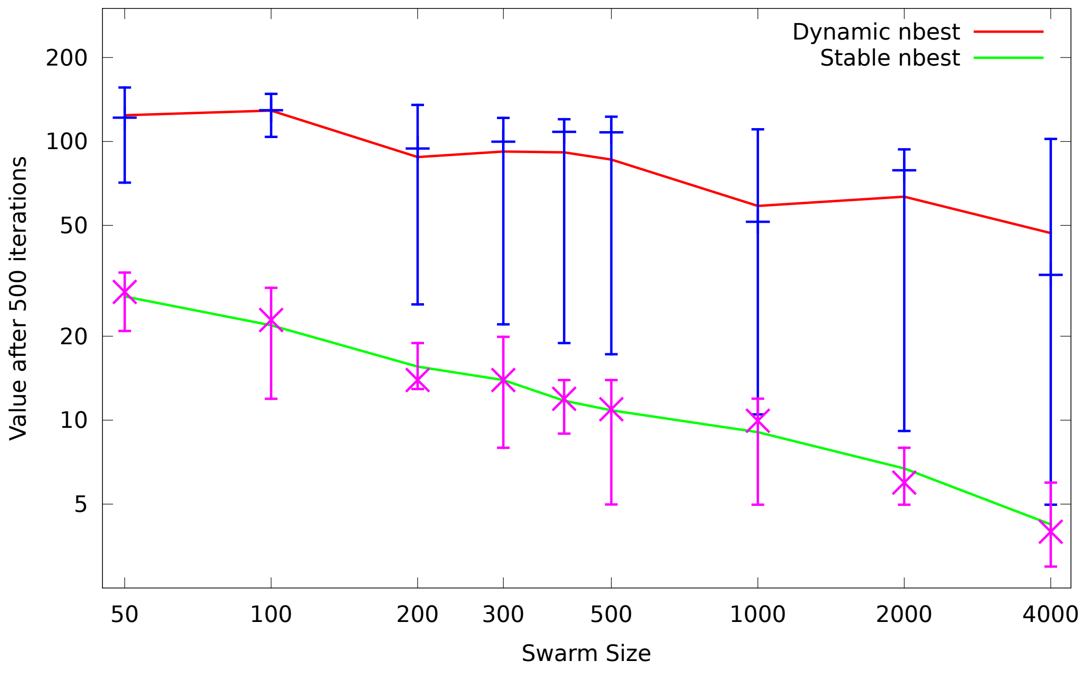

Figure 3.14: PSO performance for dynamic and stable neighborhood bests on 20-dimensional Rastrigin. The primary plots show the average performance over 20 independent runs, and the error bars show the median and the 10th and 90th percentiles.

hood best in this situation. With a *dynamic neighborhood best*, the particle would set its neighborhood best by taking the best personal best in its neighborhood, throwing out the prior neighborhood best. With a *stable neighborhood best*, the particle would retain the prior neighborhood best, replacing it only if some particle in its neighborhood has a better personal best.

Figure 3.14 shows the performance of the $Rand_{n,\frac{n}{2}}$ topology on the 20-dimensional Rastrigin benchmark function. With a stable neighborhood best, performance improves steadily with the number of particles. However, a swarm with a dynamic neighborhood best exhibits high variance and terrible overall performance. These results show that with a dynamic neighborhood best, a particle wastes the information from neighbors because it gets tugged toward an ever-changing neighborhood best instead of focusing on a productive part of the space.

Figure 3.15 shows the performance of $Ring_{n,1}$ and $Rand_{n,2}$ on the 20-dimensional

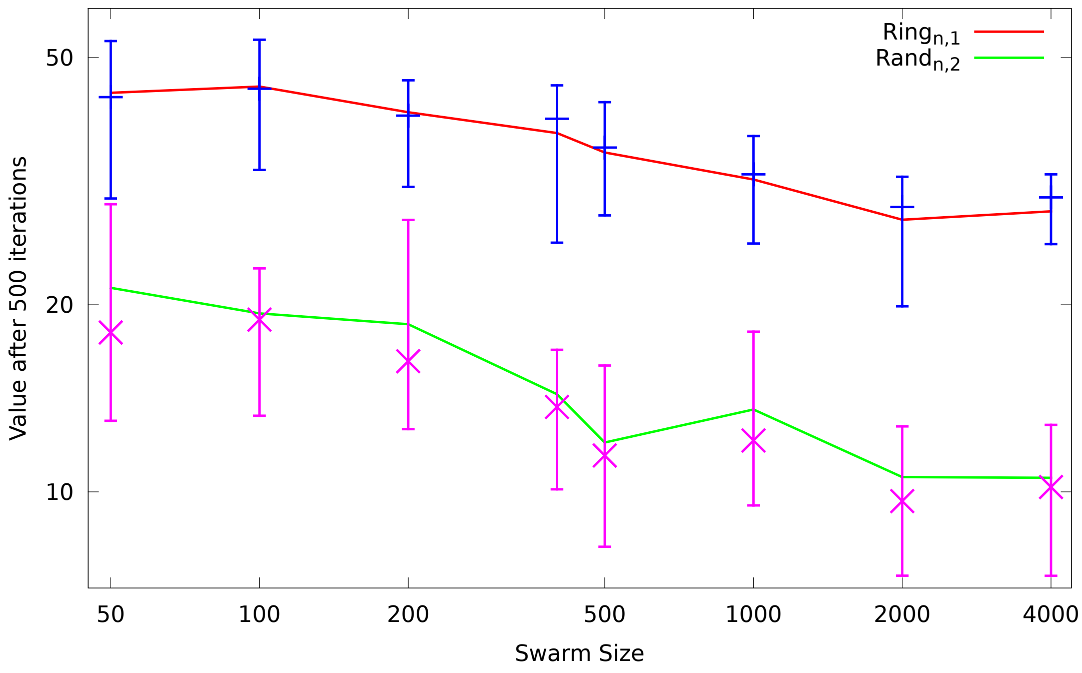

Figure 3.15: PSO performance for $Ring_{n,1}$ and $Rand_{n,2}$ on low-dimensional Rastrigin. The primary plots show the average performance, and the error bars show the median and the 10th and 90th percentiles.

Rastrigin benchmark function. The two topologies require the exact same amount of communication as each particle sends its personal best to two neighbors. The random topology performs better, even with a small swarm, and adding additional particles for $Rand_{n,2}$ has a greater effect than for $Ring_{n,1}$.

Figure 3.16 compares the performance of K_n, $Rand_{n,\frac{n}{5}}$ (20% communication), $Rand_{n,\frac{n}{20}}$ (5% communication), and $Rand_{n,2}$ on the 20-dimensional Rastrigin benchmark function. Note that with the exception of $Rand_{n,2}$ they give approximately the same results despite the random topologies requiring significantly less communication.

Sphere and low-dimensional Rastrigin are better optimized with more shared information. Although the $Rand_{n,k}$ topology is able to reduce communication dramatically without affecting performance, information propogates too slowly through the swarm when k is small. For example, $Rand_{n,2}$ performs worse than the more communicative random topologies in Figure 3.16. However, a modification of the PSO algorithm restores information flow without

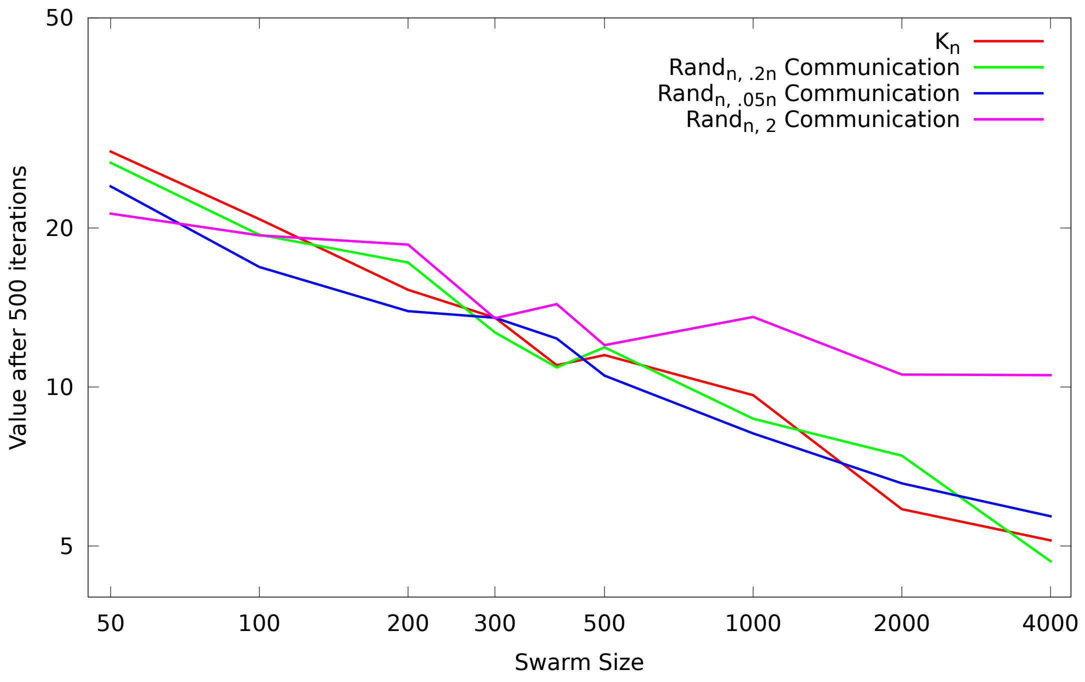

Figure 3.16: PSO performance for K_n, $Rand_{n,\frac{n}{5}}$, and $Rand_{n,\frac{n}{20}}$ on Rastrigin.

actually increasing the communication between particles. In *Hearsay PSO*, particles convey what they have heard and not just what they have seen. Specifically, they have a "transitive best" and send their neighborhood best to neighbors rather than their personal best.

Figure 3.17 shows that Hearsay PSO with $Rand_{n,2}$ outperforms standard PSO on Sphere using $Rand_{n,2}$ and $Ring_{n,1}$ topologies with the same amount of communication. Figure 3.18 makes the same comparisons for low-dimensional Rastrigin and shows that for large swarm sizes, Hearsay PSO using $Rand_{n,2}$ is competitive with Standard PSO using K_n, both of which slightly outperform Standard PSO with $Rand_{n,2}$. Figure 3.19 compares a few topologies and shows that when the number of messages is the primary cost, Hearsay PSO with $Rand_{n,2}$ outperforms standard PSO with most other topologies.

3.4 Conclusions and Future Work

We have shown that large swarms can improve the performance of Particle Swarm Optimization on several standard benchmark functions. Large swarms inherently delay convergence,

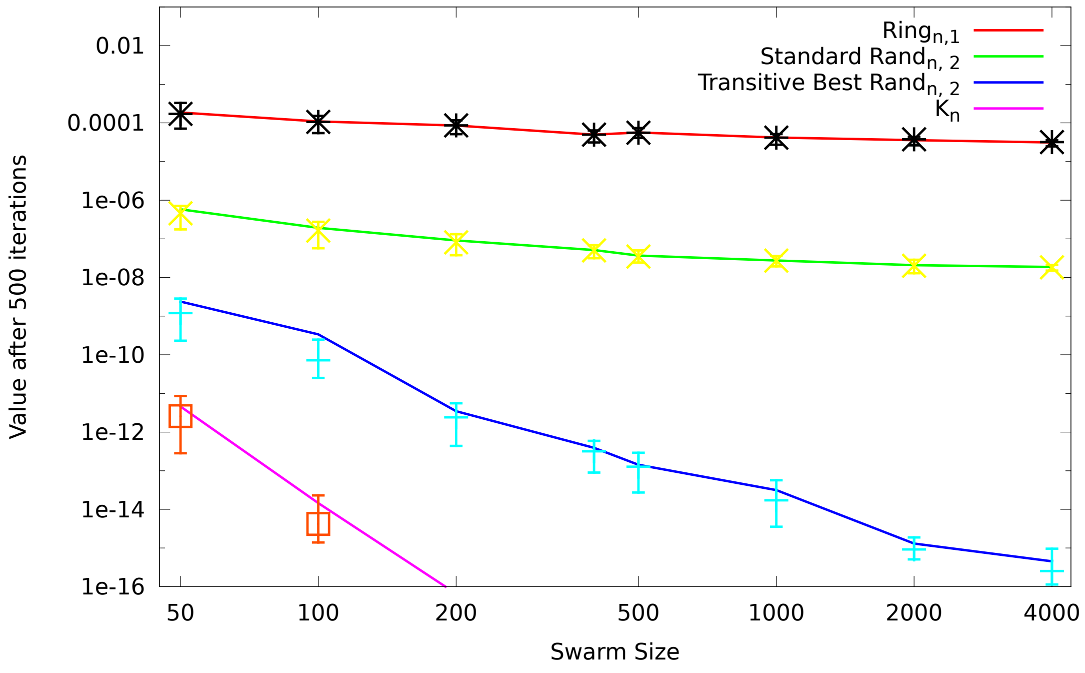

Figure 3.17: PSO performance on Sphere for transitive best $Rand_{n,2}$, $Rand_{n,2}$, and $Ring_{n,2}$. The primary plots show the average performance, and the error bars show the median and the 10th and 90th percentiles.

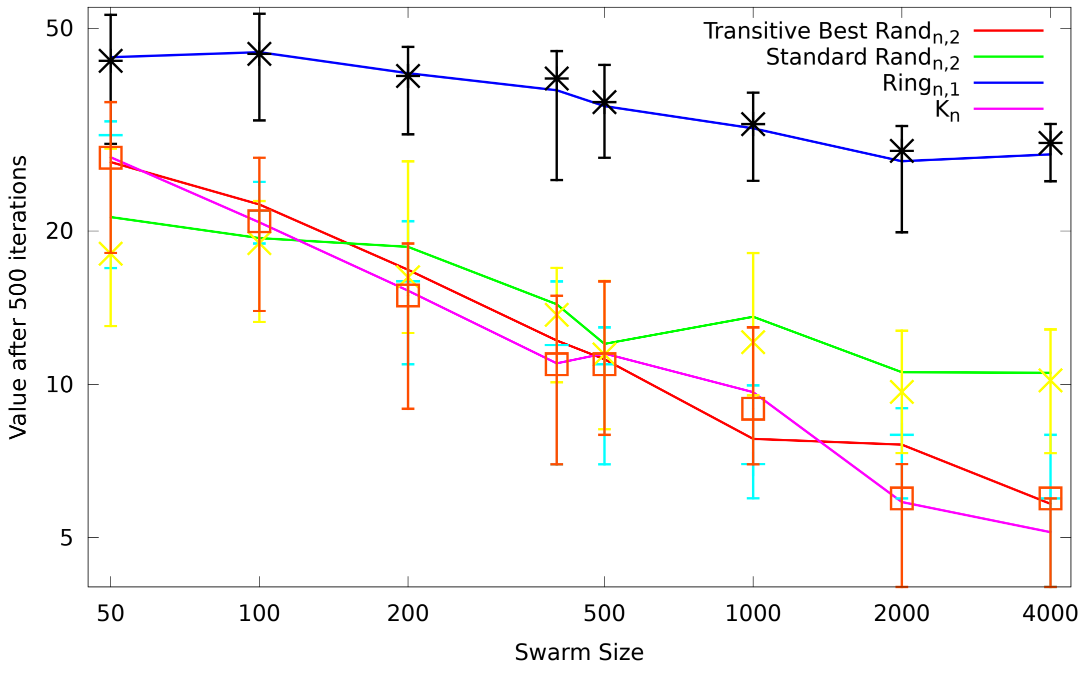

Figure 3.18: PSO performance on 20-dimensional Rastrigin for transitive best $Rand_{n,2}$, $Rand_{n,2}$, and $Ring_{n,2}$. The primary plots show the average performance, and the error bars show the median and the 10th and 90th percentiles.

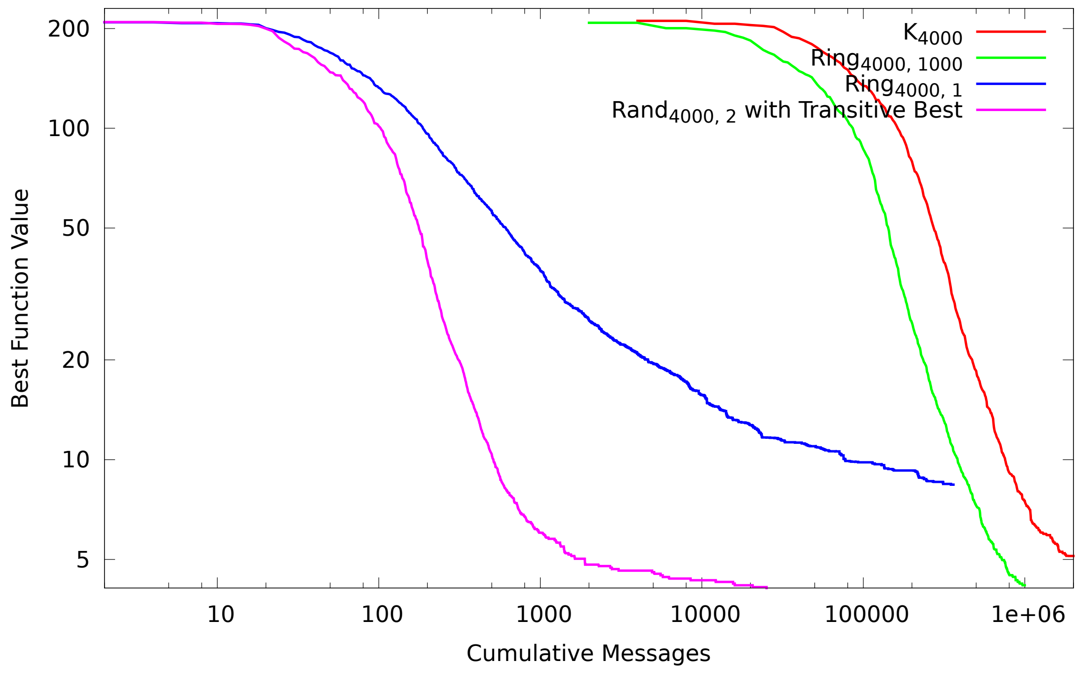

Figure 3.19: PSO performance on 20-dimensional Rastrigin with respect to communication.

which can be helpful in optimizing deceptive and moderately deceptive functions such as Griewank and Rastrigin. These results apply to both sequential and parallel environments.

However, performance gains are particularly compelling in the context of parallel computation. Where faster convergence is desirable but where communication is expensive relative to function evaluations, we have proposed adaptations of PSO. We have given a more formal approach to topologies, making it easier to describe complex and dynamic topologies. Communicating subswarms provide a simple way to perform parallel PSO for inexpensive objective functions without requiring a great amount of communication. This is helpful in the case where local communication is inexpensive (such as between cores in a multicore processor) and external communication is more expensive (like between multicore processors in a cluster). Random topologies with a stable neighborhood best increase the flow of information without destabilizing motion. Hearsay PSO allows particles to communicate more than just their personal best, which helps information spread more quickly. In a situation where a fully connected topology would otherwise be used, this makes it possible to use a more sparsely connected topology, reducing the communication overhead of PSO with a modest increase in time to convergence.

In the future, we expect to extend our work with large swarms. Large scale parallel computing environments open the door to very large swarms. A cluster of a thousand 8-core processors could allow optimization with perhaps millions of particles. Based on our experiences so far it seems likely that we will discover new issues that will drive further algorithmic advances. For example, we expect that large swarms, like larger societies, will demand richer communication for the swarm to work effectively. Sharing information such as history may allow particles to move more intelligently without increasing communication overhead. We have only begun to develop topologies that might be applicable in parallel computing environments.

As optimization is highly function-dependent, future work also includes an improved understanding of how PSO's performance is affected by characteristics of the objective function. Experiments with a greater variety of benchmark functions as well as objective functions from specific applications will further improve our understanding of how PSO behaves and how to use it most effectively.

Chapter 4

Conclusions

This thesis was intended to show that a MapReduce formulation of PSO provides a flexible and robust parallel implementation, that the larger swarm sizes encouraged by parallel PSO can improve performance, and that adaptations to PSO can achieve the effects of dense topologies with more efficient levels of communication. Chapter 2 introduces MapReduce PSO and describes its robustness, and Chapter 3 considers the effects of swarm size and topology, particularly in parallel PSO. That the topologies and adaptations of Chapter 3 were easily achievable in MapReduce PSO demonstrates the flexibility of the approach.

Chapter 2 describes how Particle Swarm Optimization can be framed in the MapReduce parallel programming model. Using off-the-shelf MapReduce frameworks, PSO researchers and practitioners can create a parallel PSO implementation without worrying about the details of communication, load balancing, or fault tolerance. Although several papers have discussed parallel PSO, all appear to describe early prototypes, and very few have published source code at all. In contrast, there are many MapReduce implementations available, and the basic MapReduce algorithm can be adapted to create custom variants of parallel PSO, as was done in Chapter 3. When PSO researchers create such algorithms using MapReduce, they produce robust implementations while delegating the communication, load balancing, and fault tolerance to the underlying framework.

Chapter 3 discusses the role of swarm size in Particle Swarm Optimization. While particle swarms are traditionally small, parallel PSO induces larger swarm sizes. Results in this chapter show that even in serial PSO, where an increase in swarm size increases the time required for each iteration, large swarms can improve the performance of PSO by increasing exploration. In parallel PSO, increasing the swarm size does not necessarily lengthen the time to compute each iteration, so the benefits are even more pronounced. For objective functions which are best optimized with densely connected swarms but where communication costs are expensive, a good compromise is provided by a random topology combined with transitive communication of the best known value. Given information about the computational environment and some basic prior knowledge of the landscape of the objective function, this chapter suggests how to make reasonable choices of topology and swarm size.

Multicore processors and computing clusters are now widely available. PSO researchers need to be more aware of the implications of parallel computation for algorithm design, and practitioners need to have more readily available tools and techniques that are effective in a parallel environment. This work has already encouraged a new approach to parallel PSO using speculative evaluation which was built in MapReduce framework [9]. As awareness of the special requirements and capabilities of parallelization increases, we expect that greater attention will be given to parallel computation in all aspects of PSO research.

References

[1] M. Belal and T. El-Ghazawi. Parallel Models for Particle Swarm Optimizers. *International Journal of Intelligent Computing and Information Sciences*, 4(1):100–111, 2004.

[2] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[3] D. Bratton and J. Kennedy. Defining a Standard for Particle Swarm Optimization. In *Proceedings of the IEEE Swarm Intelligence Symposium*, pp. 120–127, 2007.

[4] Jui-Fang Chang, Shu-Chuan Chu, John F. Roddick, and Jeng-Shyang Pan. A Parallel Particle Swarm Optimization Algorithm with Communication Strategies. *Journal of Information Science and Engineering*, 21:809–818, 2005.

[5] Maurice Clerc. TRIBES - Un exemple d'optimisation par essaim particulaire sans paramètres de contrôle. In *Optimisation par Essaim Particulaire*, Paris, France, 2003.

[6] Maurice Clerc and James Kennedy. The Particle Swarm—Explosion, Stability, and Convergence in a Multidimensional Complex Space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Sixth Symposium on Operating System Design and Implementation*, pp. 137–150, 2004.

[8] R. Eberhart and Y. Shi. Comparison Between Genetic Algorithms and Particle Swarm Optimization. In *Evolutionary Programming VII*, pp. 611–616, Springer, 1998.

[9] Matthew J. Gardner, Andrew W. McNabb, and Kevin D. Seppi. Speculative Evaluation in Particle Swarm Optimization. *Parallel Problem Solving from Nature XI*, pp. 61–70, 2010.

[10] Dennis Gies and Yahya Rahmat-Samii. Particle Swarm Optimization for Reconfigurable Phase-Differentiated Array Design. *Microwave and Optical Technology Letters*, 38(3):168–175, 2003.

[11] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, Harlow, England, second edition, 2003.

[12] Nanbo Jin and Yahya Rahmat-Samii. Parallel Particle Swarm Optimization and Finite-Difference Time-Domain (PSO/FDTD) Algorithm for Multiband and Wide-Band Patch Antenna Designs. *IEEE Transactions on Antennas and Propogation*, 53(11):3459–3468, 2005.

[13] Johannes Jordan, Sabine Helwig, and Rolf Wanka. Social Interaction in Particle Swarm Optimization, the Ranked FIPS, and Adaptive Multi-Swarms. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, pp. 49–56, ACM, 2008.

[14] J. Kennedy. Small Worlds and Mega-Minds: Effects of Neighborhood Topology on Particle Swarm Performance. In *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 3, p. 1938, 1999.

[15] James Kennedy and Russell C. Eberhart. Particle Swarm Optimization. In *International Conference on Neural Networks IV*, pp. 1942–1948, Piscataway, NJ, 1995.

[16] Byung-Il Koh, Alan D. George, Raphael T. Haftka, and Benjamin J. Fregly. Parallel Asynchronous Particle Swarm Optimization. *International Journal of Numerical Methods in Engineering*, 67:578–595, 2006.

[17] J.J. Liang and P.N. Suganthan. Dynamic Multi-Swarm Particle Swarm Optimizer. In *Proceedings of the 2005 Swarm Intelligence Symposium*, pp. 124–129, 2005.

[18] Andrew McNabb, Matthew Gardner, and Kevin Seppi. An Exploration of Topologies and Communication in Large Particle Swarms. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 712–719, 2009.

[19] Andrew W. McNabb, Christopher K. Monson, and Kevin D. Seppi. MRPSO: MapReduce Particle Swarm Optimization. In *Proceedings of the Ninth annual conference on Genetic and evolutionary computation*, p. 177, 2007.

[20] Andrew W. McNabb, Christopher K. Monson, and Kevin D. Seppi. Parallel PSO using MapReduce. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 7–14, 2007.

[21] Rui Mendes. *Population Topologies and Their Influence in Particle Swarm Performance*. Ph.D. thesis, Escola de Engenharia, Universidade do Minho, 2004.

[22] Russell Merris. *Graph Theory.* John Wiley & Sons, New York, 2001.

[23] Vladimiro Miranda, Hrvoje Keko, and Álvaro Jaramillo Duque. Stochastic Star Communication Topology in Evolutionary Particle Swarms. *International Journal of Computational Intelligence Research*, 4(2):105–116, 2008.

[24] Arvind S. Mohais, Christopher Ward, and Christian Posthoff. Randomized Directed Neighborhoods with Edge Migration in Particle Swarm Optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation*, volume 1, pp. 548–555, 2004.

[25] Marco A. Montes de Oca and Thomas Stützle. Convergence Behavior of the Fully Informed Particle Swarm Optimization Algorithm. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, pp. 71–78, ACM, 2008.

[26] Marco A. Montes de Oca, Thomas Stützle, Mauro Birattari, and Marco Dorigo. Frankenstein's PSO: a Composite Particle Swarm Optimization Algorithm. *Transactions on Evolutionary Computation*, 13(5):1120–1132, 2009.

[27] Sanaz Mostaghim, Jürgen Branke, and Hartmut Schmeck. Multi-Objective Particle Swarm Optimization on Computer Grids. Technical Report 502, AIFB Institute, 2006.

[28] K. E. Parsopoulos, D. K. Tasoulis, and M. N. Vrahatis. Multiobjective Optimization Using Parallel Vector Evaluated Particle Swarm Optimization. In *Proceedings of the IASTED International Conference on Artificial Intelligence and Applications*, pp. 823–828, 2004.

[29] J.R. Perez and J. Basterrechea. Particle Swarm Optimization for Antenna Far-Field Radiation Pattern Reconstruction. In *Proceedings of the 36th European Microwave Conference*, pp. 687–690, 2006.

[30] Mark Richards and Dan Ventura. Dynamic Sociometry in Particle Swarm Optimization. In *Proceedings of the Sixth International Conference on Computational Intelligence and Natural Computing*, pp. 1557–1560, 2003.

[31] J. F. Schutte, J. A. Reinbolt, B. J. Fregly, R. T. Haftka, and A. D. George. Parallel Global Optimization with the Particle Swarm Algorithm. *International Journal for Numerical Methods in Engineering*, 61(13):2296–2315, 2004.

[32] Ian Scriven, David Ireland, Andrew Lewis, Sanaz Mostaghim, and Jürgen Branke. Asynchronous Multiple Objective Particle Swarm Optimisation in Unreliable Distributed

Environments. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 2481–2486, 2008.

[33] Ian Scriven, Andrew Lewis, David Ireland, and Junwei Lu. Decentralised Distributed Multiple Objective Particle Swarm Optimisation Using Peer to Peer Networks. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pp. 2925–2928, 2008.

[34] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. A Parallel Particle Swarm Optimization Algorithm Accelerated by Asynchronous Evaluations. In *Proceedings of the 6th World Congresses of Structural and Multidisciplinary Optimization*, 2005.

[35] David H. Wolpert and William G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.