2015-03-01

Judicious Use of Communication for Inherently Parallel Optimization

Andrew W. McNabb
Brigham Young University - Provo

Judicious Use of Communication for Inherently Parallel Optimization

Andrew W. McNabb

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

Doctor of Philosophy

Kevin Seppi, Chair
Christophe Giraud-Carrier
Eric Mercer
Daniel Zappala
Dan Olsen

Department of Computer Science

Brigham Young University

March 2015

ABSTRACT

Judicious Use of Communication
for Inherently Parallel
Optimization

Andrew W. McNabb
Department of Computer Science, BYU
Doctor of Philosophy

Function optimization—finding the minimum or maximum of a given function—is an extremely challenging problem with applications in physics, economics, machine learning, engineering, and many other fields. While optimization is an active area of research, only a portion of this work acknowledges parallel computation, which is now widely available. Today, anyone with a modest budget can buy a cluster with hundreds of cores, pay for access to a supercomputer with thousands of processors, or at least purchase a laptop with 8 cores. Thus, an algorithm that works well in serial but cannot be parallelized is needlessly inefficient in real-life computational environments.

We address these issues in three connected threads of development: a high-level programming framework that makes it possible to create flexible and efficient implementations of optimization algorithms; improvements to an existing algorithm, Particle Swarm Optimization, to make it take better advantage of parallel resources; and a statistical model designed to efficiently use available information in parallel optimization by inferring search directions. Each of these is an essential step toward effective parallel optimization. First, without a suitable high-level programming model, expediency leads to purely serial development with parallel issues only an afterthought. Second, PSO has proven effective for optimization and is an excellent candidate to consider for efficient parallel implementations. Third, a model for inference of search directions is useful for understanding communication in the context of parallel optimization and provides a flexible base for continuing optimization research.

Keywords: function optimization, parallel computation, Particle Swarm Optimization, Bingham distribution

Table of Contents

Introduction

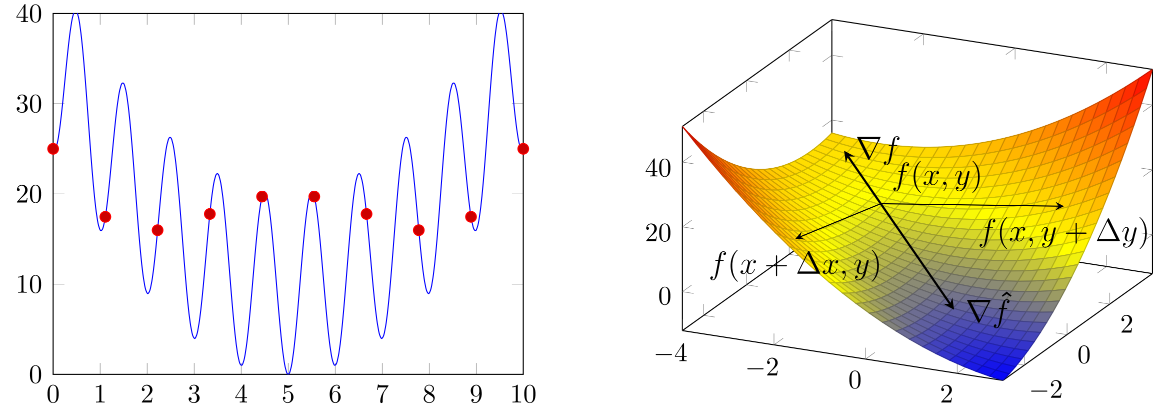
Chapter 1

Introduction

1.1 Optimization

Optimization is the problem of finding the minimum or maximum, along with its associated input value, for a given objective function—in other words, finding $x^\star = \arg\min f(x)$ and $f(x^\star)$. There are many different types of optimization problems. An objective function may have discrete- or continuous-valued inputs, or even a mix. Valid solutions may be constrained to certain areas of the function's domain, or search may be unconstrained. A local optimization problem only requires a local minimum or maximum, while a global optimization problem requires a global optimum. Analytical optimization may take advantage of closed-form derivatives, while empirical (also referred to as direct search or black box) optimization has only values at the points that it evaluates. This work is concerned with unconstrained global continuous empirical optimization.

Optimization in general is an exceptionally difficult problem. Several obvious strategies are ineffective. First, one might consider sweeping the domain of a function. Figure 1.1a demonstrates that a grid of samples from a continuous-valued function may lead to incorrect conclusions about the location of the global minimum. Furthermore, sweeping the domain is exponential in the number of dimensions. For a 50-dimensional function, taking each combination of just 2 points from each dimension (the corners of a hypercube) would require 2^{50} (over 10^{15}) evaluations. Second, one might consider estimating the gradient at a point by evaluating at a short distance away along each dimension. Figure 1.1b shows that this



(a) Points superimposed on the Rastrigin function represent observations. The points suggest that the region around $x = 5$ contains a local maximum, but $x = 5$ is in fact the global minimum.

(b) A situation where an estimated gradient $(\nabla \hat{f})$ points in the opposite direction of the function's true gradient (∇f).

Figure 1.1: Two situations where simple ideas for optimization fail badly.

estimated gradient can point in the opposite direction of the true gradient. Especially in high-dimensional spaces, a finite set of samples from a function can be extremely misleading.

Optimization algorithms are usually assessed in a serial context, and parallelization is usually added as an afterthought without fully addressing the opportunities and concerns that are present in a parallel computational environment. Research rarely considers the impact of changes on issues that affect parallel algorithms, such as communication and barriers. Furthermore, current algorithms fail to take advantage of the significantly increased resources that are available on a parallel cluster. Parallel computation should serve as the standard execution environment of optimization algorithms, influence the development of existing algorithms, and drive the design of novel optimization algorithms.

Most empirical optimization algorithms, such as Evolution Strategies, Simulated Annealing, and Particle Swarm Optimization manage a small sample of points, which are combined, modified, and reevaluated at each iteration to produce a new sample. Regions where the value of the function is better are favored over regions where the value is worse. Evolutionary algorithms exhibit stability of solutions as populations evolve as a group, but combination mechanisms designed for serial computation often depend on significant or even global communication between processors. Existing improvements to parallelization do not fully use available information. For example, virtually all optimization algorithms discard all sampled points at each iteration, despite the vast memory on a cluster. Approaches based on small working sets of data are simple, but a fresh non-evolutionary perspective allows for taking a more principled approach to analyzing data, extracting as much value as possible from previous samples.

1.2 Research Area Overview

Exhaustive search of a large space is impractical, so empirical optimization algorithms are generally stochastic. These algorithms perform a sequence of function evaluations, and it is helpful to model this behavior as a random walk. Some algorithms sample from one position at each time step, while population-based algorithms simulate random walks of multiple interdependent particles. We consider several influential local and global optimization algorithms but omit methods which require function derivatives (e.g., the Broyden–Fletcher–Goldfarb–Shanno method) or which are specific to constrained optimization problems (e.g., Sequential Quadratic Programming).

The simplest algorithms are referred to as local optimization algorithms and operate under the assumption that the objective function is unimodal. While this work is concerned with global optimization, which violates this assumption, we consider several influential local optimization algorithms. The Hill Climbing algorithm is a greedy algorithm that makes a change in one dimension of the input at a time and moves to the new position if it is better than the current position. Hill Climbing is based on the Gradient Descent algorithm of [1], which requires the gradient at each point. Powell's Algorithm [2] manages a list of search vectors and at each iteration performs a sequence of line searches along these directions and replaces one of the vectors. The Nelder-Mead Simplex method [3] updates a simplex defined by a set of points numbering one more than the number of dimensions. At each step, the worst point in the simplex is reflected across the centroid, and the simplex is either expanded or contracted depending on whether the reflected point is better than the worst point. Matyas [4] proposes Random Optimization, which samples a candidate value from a Normal distribution centered on the current best value and moves if the new value is better. Local optimization algorithms can be used for global optimization if a function is known to be unimodal or if run repeatedly with random restart.

Kirkpatrick et al. [5] propose Simulated Annealing, a simple and popular approach based on an algorithm by Metropolis [6] for performing simulations in statistical mechanics. Simulated Annealing performs a random walk similar to Random Optimization, with a dynamic temperature parameter that controls the probability of moving from a better to a worse value. An initial high temperature encourages exploration, and a lowering temperature increases exploitation, causing the algorithm to gradually settle to a local optimum.

Several evolution-inspired approaches perform function optimization by simulating biological evolution and using the value of the objective function as a measure of fitness. Evolution Strategies [7] simulate the evolution of a population of individuals represented as real-valued vectors. Mutation is performed by drawing from a Normal distribution (as in Random Optimization), and optional recombination may be given by randomly selecting elements from either parent [8]. Genetic Algorithms [9] also simulate evolution but use bit-string instead of real-valued representations. They were originally designed to be adaptive systems but are also used for function optimization [10], though Evolution Strategies are more commonly used for continuous function optimization. A few modern Evolution Strategies are particularly effective. The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [11] includes the parameters of the mutation distribution in the representation of each individual. Differential Evolution [12] mutates an individual by adding in the weighted difference between two randomly chosen individuals from the population. Continuous Estimation of Distribution Algorithms [13, 14] (related to Population-Based Incremental Learning) sample the next generation from probability distributions inferred from the current population. Particle

Swarm Optimization [15] creates a new generation by simulating motion of individuals in the current generation and attracting individuals to neighbors with promising solutions.

1.3 Thesis Statement

Parallel optimization algorithms are improved in terms of quality of solution and runtime performance when they are designed specifically to reduce communication and unnecessary coordination. Furthermore, high-level frameworks can provide the flexibility to consider a variety of approaches to parallel optimization, and directional statistics can provide a unique theoretical perspective for adapting search directions with low communication and coordination.

1.4 Overview

Too often, parallelization is considered an implementation detail rather than a primary concern of optimization algorithms. Parallel computation should serve as the standard execution environment of optimization algorithms, influence the development of existing algorithms, and drive the design of novel optimization algorithms. We address these issues in three connected threads of development, each of which is a separate part of this work:

1. Part I: a high-level programming framework that makes it possible to create flexible and efficient implementations of optimization algorithms such as Particle Optimization (PSO),

2. Part II: changes to an existing algorithm, PSO, to improve parallel performance by reducing communication and centralized coordination, and

3. Part III: a novel statistical model for inference of search directions that is designed to enable algorithms that exploit separability and minimize communication and coordination in a parallel environment.

Each of these is an essential step toward effective parallel optimization. First, without a suitable high-level programming model, expediency leads to purely serial development with parallel issues only an afterthought. Second, PSO has proven effective for optimization and is an excellent candidate to consider for efficient parallel implementations. Third, probability theory moves beyond the inherent limitations of parallel evolutionary algorithms, and statistical models are a strong basis for algorithms with transparent behavior.

1.4.1 Programming Framework for Parallel Optimization (Part I)

A high-level parallel programming model that is well-suited to iterative optimization algorithms increases the availability and performance of parallel optimization algorithms. Most parallel programming models are unsuitably low-level for optimization research, causing programs to be difficult to write, brittle, and hard to modify. Parallel optimization algorithms are typically implemented from scratch using low-level programming models, making it difficult to rapidly adapt ideas into prototypes for new algorithms. This is particularly troublesome because optimization is an active area of research, and algorithms are constantly modified. The complexity of parallel programming is a major obstacle to the widespread adoption of parallelization in optimization. High-level frameworks reduce the complexity of building parallel programs but are typically poorly suited to iterative algorithms.

The MapReduce model is a good basis for optimization algorithms due to its simplicity and familiarity. A MapReduce program is written at a high level without any explicit communication, which is managed completely by the framework. This allows programs to focus on the essence of an algorithm without becoming overwhelmed by bookkeeping. Furthermore, this model is familiar within the broad area of machine learning. MapReduce has proven popular for machine learning algorithms such as k-means [16], logistic regression [17], back-propagation [17], independent component analysis [17], expectation maximization (EM) [17], support vector machines [17], and genetic algorithms [18].

Part I presents a MapReduce framework called Mrs [1] that is well-suited to optimization algorithms. Chapter 2 formulates Particle Swarm Optimization as a MapReduce program. Although the MapReduce programming model is a good fit for PSO, the per-iteration overhead of Hadoop, designed for big data rather than computationally intensive applications, is prohibitively expensive for iterative algorithms. Most existing MapReduce implementations exhibit poor performance for iterative programs such as optimization algorithms, and the standard MapReduce model cannot express some important parallel optimization algorithms such as asynchronous parallel PSO. Chapter 3 introduces the Mrs framework, and Chapter 4 augments the MapReduce model with major improvements for iterative algorithms, including support for asynchronous algorithms.

The asynchronous MapReduce programming model removes communication bottlenecks that affect optimization algorithms and other iterative programs. Each iteration of an algorithm in MapReduce consists of at least one map phase that performs the evaluation of the objective function and a reduce phase that performs communication. In standard MapReduce, the reduce step cannot proceed until all map tasks have completed. However, in an optimization algorithm, starting some tasks in the next iteration early may be desirable. In the asynchronous iterative MapReduce model, when a task proceeds to the next iteration early, then missed messages would be rerouted to tasks in later iterations. This model is implemented in Mrs.

1.4.2 Reconsidering Particle Swarm Optimization in a Parallel Context (Part II)

Part II considers changes to PSO to improve its performance in a parallel computational environment. PSO is a particularly convenient algorithm for exploring parallel issues in optimization. It is a relatively simple existing algorithm, making it a good test bed for new ideas, and its swarm topology is very closely tied to communication in a parallel implementation. Chapters 5 and 6 introduce two changes to PSO, speculative evaluation and

[1] A MapReduce program is often prefixed by "MR," which can be pronounced "mister" (for example, "MRWordCount"). As a play on words, the new implementation is called "Mrs."

a subswarm-based swarm topology respectively, to explore approaches to the efficient use of resources and communication parallel optimization. Chapter 7 emphasizes the importance of evaluating PSO algorithms from a parallel perspective and reviews the fundamental issues of parallel PSO that algorithm designers must consider when proposing a improvement to PSO.

Speculative evaluation, described in Chapter 5, accelerates convergence for objective functions which are slow to evaluate and which show little benefit from increased swarm sizes. In speculative evaluation, at each step, the objective function is evaluated not only at each particle's current location, but also at several likely locations of the particle at the following iteration. Speculative evaluation allows the algorithm to proceed two iterations at a time with the exact same computations and numerically identical answers as in the standard algorithm. Our experiments show that for some objective functions, additional processors are more effective when used for speculative evaluation than for increasing the swarm size.

In the new "apiary" topology, described in Chapter 6, particles are organized into semi-independent subswarms and occasionally send messages to neighbors in other subswarms. By comparison, in PSO with islands used in parallel genetic algorithms, each processor manages and evaluates a set of particles which occasionally migrate between processors to increase diversity. However, migration is unnecessary in parallel PSO, where communication is not defined by membership in a group but rather by messages sent to neighbors. Unlike existing islands-based approaches in PSO [19, 20], a new topology does not require global coordination, heavy communication, or other complicated schemes that are inefficient in parallel. Our work shows that communication in an apiary is compatible with asynchronous iteration which improves parallel performance, especially for large clusters or function with heterogeneous evaluation times. For the apiary topology in an asynchronous MapReduce programming model, the PSO map function is very similar to that of MapReduce PSO with a standard topology. The map function performs a predetermined number of iterations and outputs the updated state of the apiary along with "message" records. These messages are automatically routed to a subsequent reduce task associated with the appropriate subswarm. The apiary approach is simple, economical, and improves the performance of PSO both in serial and in parallel.

Chapter 7 argues that the standard practice of evaluating a PSO variant by reporting function values with respect to the number of function evaluations is inadequate for evaluating PSO in a parallel environment. Results that demonstrate an improvement only for serial PSO are no longer relevant. A variant of PSO should be evaluated, first by how its performance scales independently of communication, and then by the task interactions and communication that it requires. This information, combined with details about a particular objective function and computational environment, determine the parallel behavior of the PSO variant. While this discussion is focused on Particle Swarm Optimization, the issues and results have implications for parallel optimization algorithms in general.

1.4.3 Inference of Search Directions (Part III)

In a few limited cases, optimization is embarrassingly parallel. For example, for an arbitrary unknown objective function, no algorithm performs better than random search on average, by the No Free Lunch theorem[21]. Random search running independently on multiple processors does not require any communication, so if no exploitable properties are known to hold for a given objective function, communication cannot offer any benefit. Likewise, optimization is embarrassingly parallel if it is known to be separable with respect to a given set of basis vectors, or orientation. An n-dimensional function is *separable* if it can be expressed as $f(\mathbf{x}) = \sum_{i=1}^{n} f_i(x_i)$. Separability is a well-known property in optimization and is shared by many benchmark functions and some real-life problems. If a function is known to be separable, then optimization is embarrassingly parallel along individual dimensions. However, the definition of separability is fairly restrictive and does not necessarily apply directly to an objective function of interest.

Even when a function is not separable, it may be *partially separable* along certain dimensions or small groups of dimensions. Furthermore, it may be partially separable in some areas of the search space. Sensitivity to scale and rotation is a common challenge in optimization, and some algorithms attempt to adapt to the search space and function landscape. CMA-ES and differential evolution both take advantage of an assumption that vectors between good values are promising directions for future candidate positions. Such approaches can be interpreted as performing a kind of dimensionality reduction to find latent separability in a function relative to different coordinate space. Unfortunately, parallel evolutionary algorithms inherently require communication every iteration for crossover, and there are limits to how much this parallelism can be improved through modifications to evolutionary algorithms such as those considered in Part II.

Part III develops a statistical model for inference of search directions that is designed for building inherently parallel optimization algorithms. Chapter 8 presents this model and demonstrates its behavior in inferring search directions. As part of this model, the chapter introduces a new conjugate prior of the Bingham distribution that makes it possible to perform Bayesian inference on directions. The prior distribution in the model favors search directions clustered around the direction that is orthogonal to those being used by other processors. The posterior distribution is updated by observations of successful directions, such as those with evidence of separability. This statistical model provides a foundation for building a wide range of models and algorithms for parallel optimization.

1.5 Publications

Many of the chapters in this work have been published previously as peer reviewed conference papers. The following list provides citations for each of these papers:

- Chapter 2: A. McNabb, C. Monson, and K. Seppi. Parallel PSO using MapReduce. In *Proc. IEEE Congress on Evolutionary Computation*. 2007.

- Chapter 3: A. McNabb, J. Lund, and K. Seppi. Mrs: MapReduce for Scientific Computing in Python. In *Proc. Python for High Performance and Scientific Computing*. 2012.

- Chapter 5: M. Gardner, A. McNabb, and K. Seppi. Speculative Evaluation in Particle Swarm Optimization. In *Proc. Parallel Problem Solving from Nature*, 2010.

- Chapter 6: A. McNabb and K. Seppi. The Apiary Topology: Emergent Behavior in Communities of Particle Swarms. In *Proc. Parallel Problem Solving from Nature*. 2012.

- Chapter 7: A. McNabb and K. Seppi. Serial PSO results are irrelevant in a multi-core parallel world. In *Proc. IEEE Congress on Evolutionary Computation*. 2014.

Part I

Programming Framework for Parallel Optimization

Parallel optimization research requires the flexibility to experiment with a variety of approaches to parallelization. Unfortunately, algorithms implemented in low-level libraries are not flexible enough to allow for major changes in design. In Chapter 2, we demonstrate that the MapReduce model is a natural fit for Particle Swarm Optimization. However, implementations of MapReduce designed for big data have high per-iteration overhead that makes them unnecessarily inefficient for iterative algorithms such as optimization algorithms. Chapter 3 introduces a lightweight MapReduce framework designed for iterative algorithms that is flexible enough for optimization research. Chapter 4 augments the MapReduce model to better support iterative algorithms, including asynchronous iteration. These changes improve convenience and performance which enable experimentation with various approaches to parallel PSO, such as those developed in Part II.

Chapter 2

Parallel PSO Using MapReduce

Published in Proceedings of CEC 2007 [22]

This chapter describes how to formulate parallel PSO within the MapReduce model. In the process, it demonstrates the convenience of MapReduce for parallel PSO and lays the foundation for PSO variants in subsequent chapters. These algorithms, while varying widely in the details of communication, are all based on the approach used by standard parallel PSO in MapReduce.

Abstract

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 [23]. 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 $\mathbf{x}$ and velocity $\mathbf{v}$ of a particle with personal best $\mathbf{x}^P$ and global best $\mathbf{g}$:

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

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

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 $\phi = \phi_1 + \phi_2$ [24].

There are several parallel adaptations of Particle Swarm Optimization. Synchronous PSO, like MRPSO, preserves the exact semantics of serial PSO [25]. In contrast, asynchronous variants do not preserve the exact semantics of serial PSO, but instead focus on better load balancing [26, 27]. Other variants propose different topologies to limit communication among particles and between groups of particles [28, 29]. Parallel PSO has been applied to applications including antenna design [30] and biomechanics [26] and adapted to solve multiobjective optimization problems [29, 31].

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 [23].

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.

Program 1 WordCount Map

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

Program 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 Program 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 Program 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 conforms to

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

Figure 2.1: A particle as a key-value pair

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' `ids`), 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 $f(\mathbf{x}) = x_1^2 + x_2^2$. 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)

Program 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_value)
        else:
            emit((dependent_id, repr(message)))

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

Program 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 $f(\mathbf{x})$ 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 Program 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 Program 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.

Program 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 (Program 3) and reduce (Program 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 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 [32]. 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 [33]. The following equation describes the activation function of an RBF network:

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

The RBF network f is composed of n_{bases} Gaussian basis functions. Basis i has center $\mathbf{c_i}$, input weights $\mathbf{w_i}$ (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:

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

where **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:

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

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.

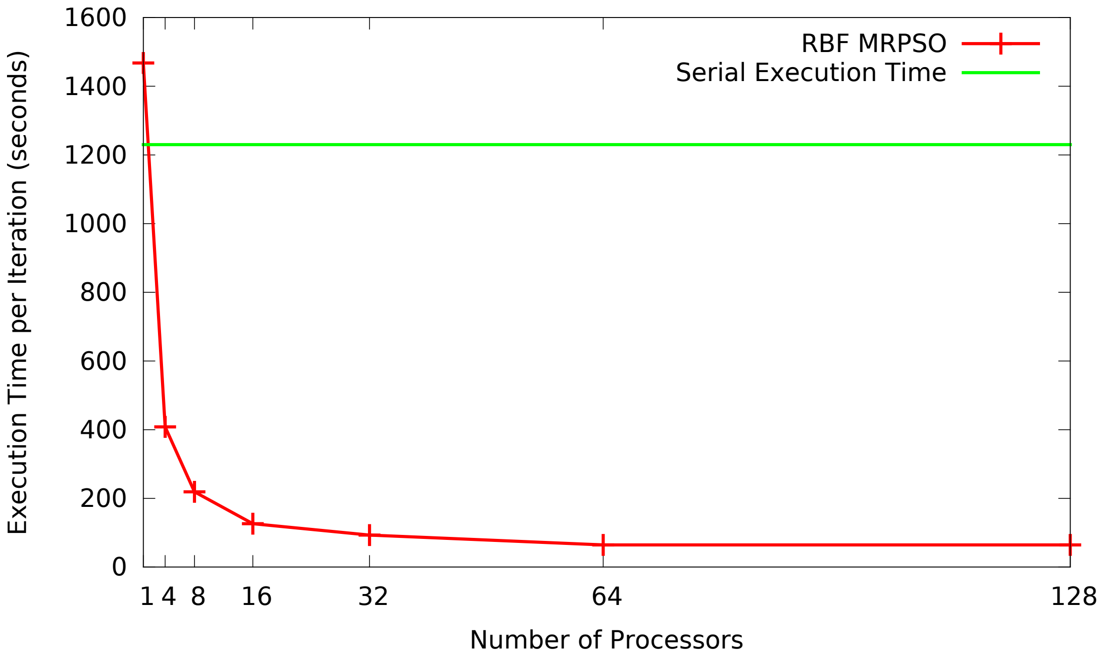


Figure 2.2: Execution times per iteration for RBF with 10,000 points

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: $f(\mathbf{x}) = x_1^2 + x_2^2 + \cdots + x_n^2$. 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

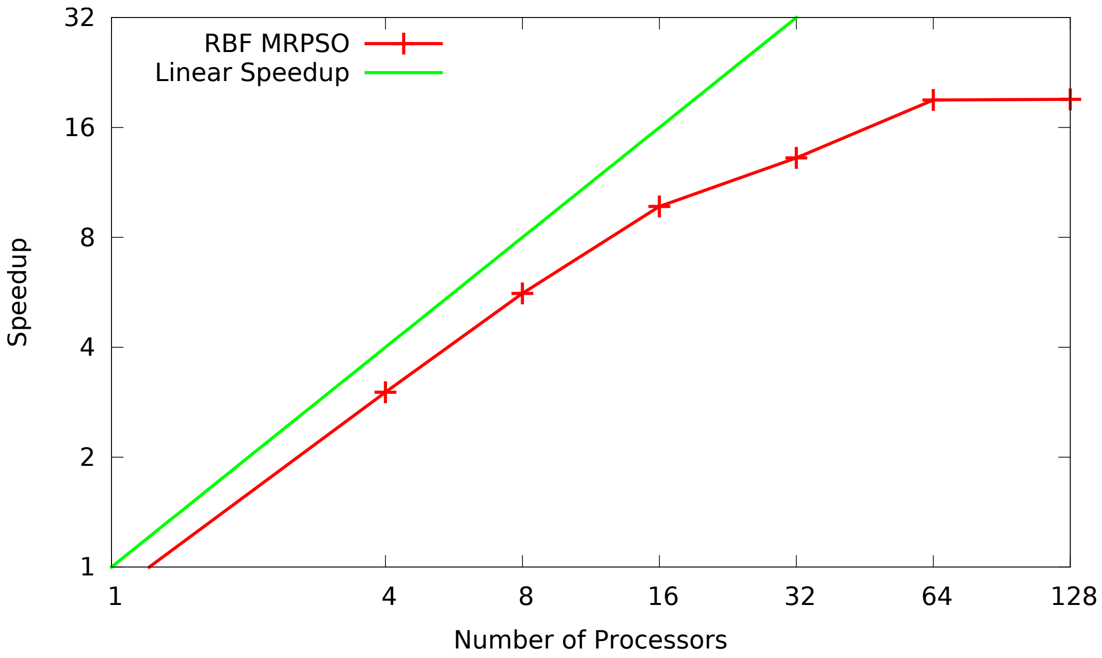


Figure 2.3: Speedup for RBF with 10,000 data points

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 time, while the line beneath the lower curve shows the amount of overhead. As the number

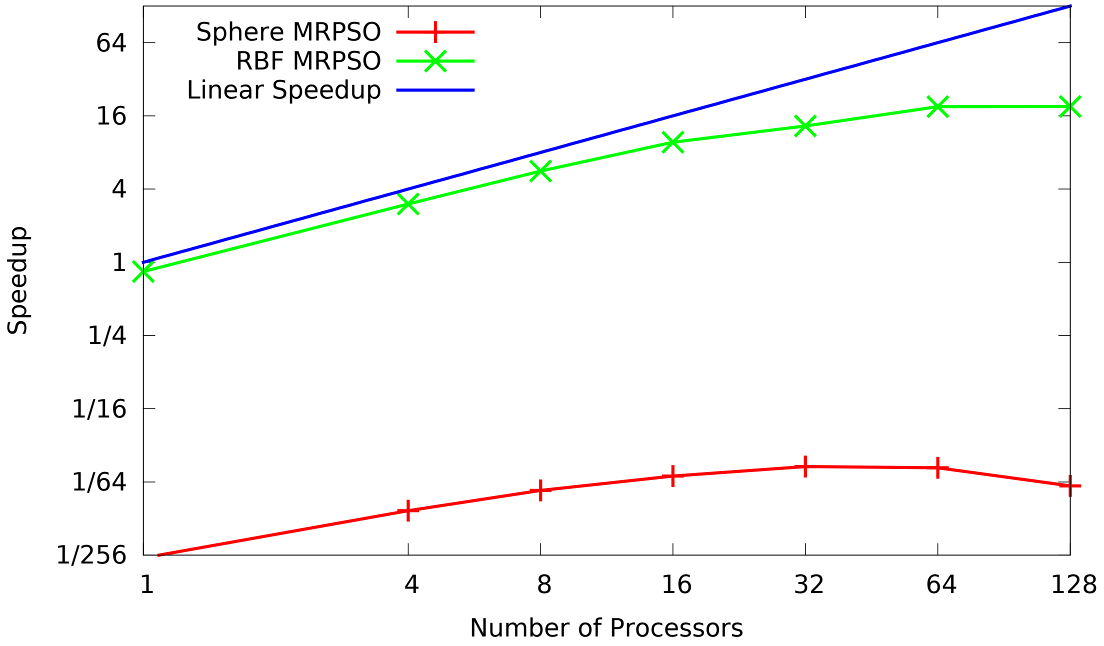


Figure 2.4: Speedup for the sphere function compared to the speedup for RBF. Using MRPSO for sphere would not be appropriate.

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).

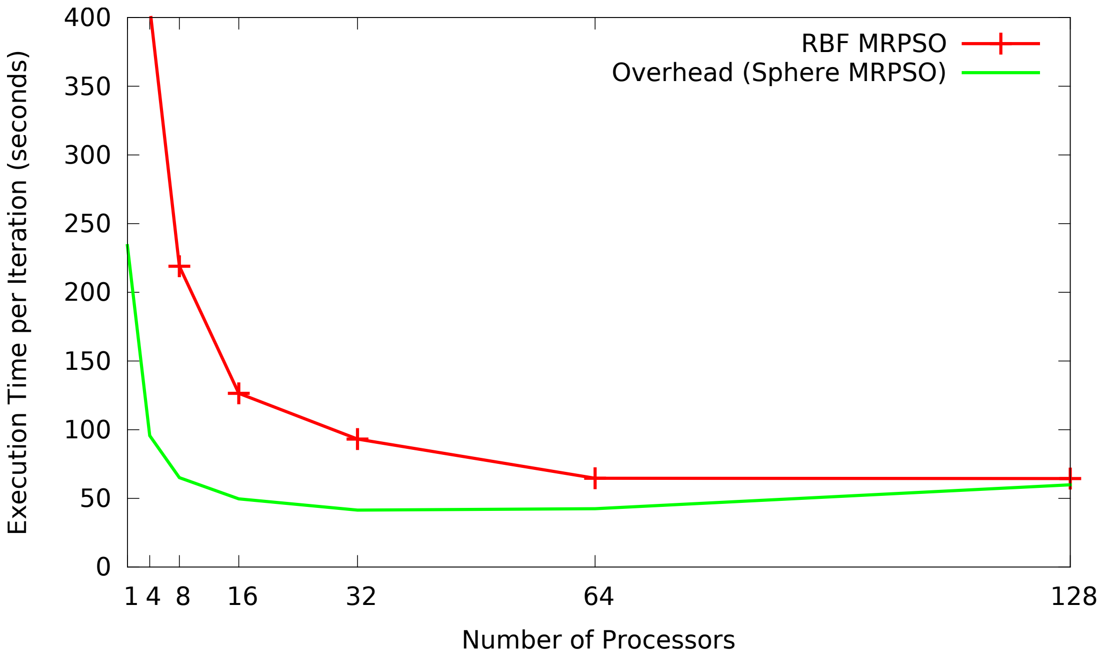


Figure 2.5: Execution times per iteration for RBF with 10,000 data points and implementation overhead as measured by sphere.

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.

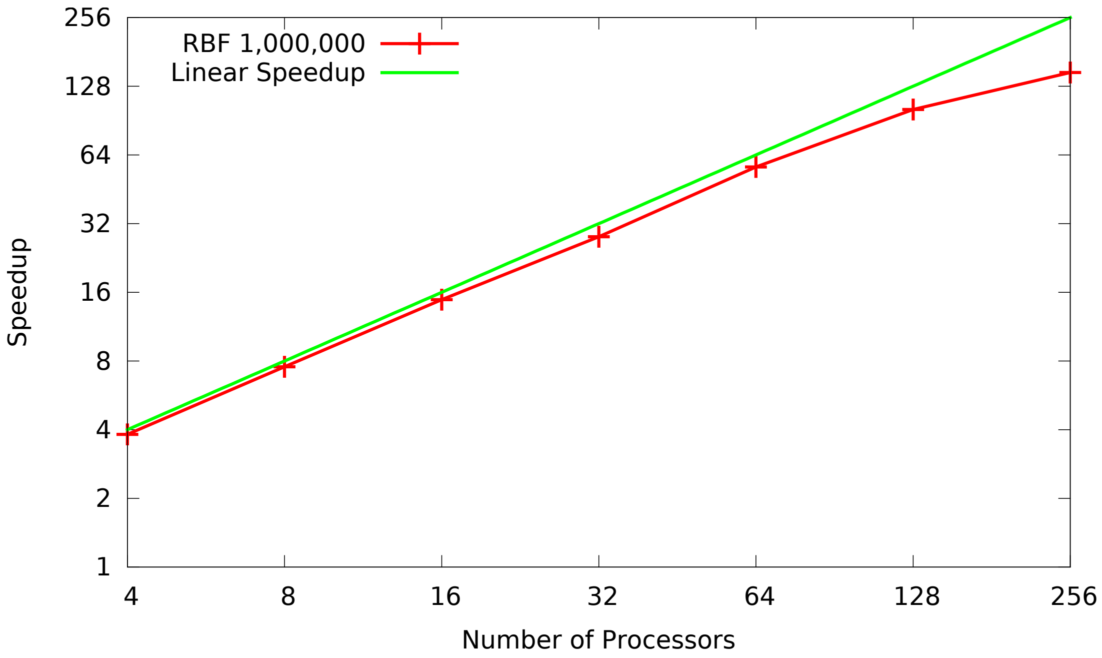


Figure 2.6: Speedup for RBF with 1,000,000 data points

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. [34]

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

Mrs: MapReduce for Scientific Computing in Python

Published in Proceedings of PyHPC 2012 [35]

This chapter describes a flexible and convenient MapReduce system. Its flexibility makes it suitable for a wide range of parallel optimization algorithms, and its convenience enables rapid prototyping and experimentation with novel algorithms. Chapter 4 builds on the basic system to adapt the MapReduce model to give better performance for computationally intensive iterative programs such as optimization algorithms without sacrificing convenience and flexibility.

Abstract

The MapReduce parallel programming model is designed for large-scale data processing, but its benefits, such as fault tolerance and automatic message routing, are also helpful for computationally-intensive algorithms. However, popular MapReduce frameworks such as Hadoop are slow for many scientific applications and are inconvenient on supercomputers and clusters which are common in research institutions.

Mrs is a Python-based MapReduce framework that is well suited for scientific computing. We present comparisons of programs and run scripts to argue that Mrs is more convenient than Hadoop, the most popular MapReduce implementation. We also demonstrate that Mrs outperforms Hadoop for several types of problems that are relevant to scientific computing. In particular, Mrs demonstrates per-iteration overhead of about 0.3 seconds for

Particle Swarm Optimization, while Hadoop takes at least 30 seconds for each MapReduce operation, a difference of two orders of magnitude.

3.1 Introduction

MapReduce [23] has quickly become a popular paradigm for large scale data intensive analysis and has also been applied to computationally intensive programs. It has been used for iterative algorithms such as k-means [16], logistic regression [17], backpropagation [17], independent component analysis [17], expectation maximization [17], support vector machines [17], genetic algorithms [18], and particle swarm optimization (PSO) [36]. The popularity of MapReduce may be attributed to its simplicity and availability.

Unfortunately, most current MapReduce frameworks exhibit poor performance in scientific applications [16–18, 36] and are ill suited to the computational environments that are most important for scientific computing. Many universities and research institutions have supercomputers, but these are not tied to any particular parallel processing technology. Most popular MapReduce frameworks are designed for large-scale data processing in datacenters and require a dedicated cluster and extensive configuration. Such frameworks use technologies that make MapReduce unnecessarily complex to program and difficult to run on supercomputers and clusters that are common in research institutions. Furthermore, performance is not optimized for computationally intensive applications, particularly iterative algorithms.

Mrs is a lightweight Python-based MapReduce implementation. It is designed to make MapReduce programs easy to write, easy to run, and fast. Python helps make these design goals possible. Mrs programs are easy to write because of the convenience and readability of Python. The Mrs API is also designed to avoid the need for unnecessary boilerplate. Mrs programs are easy to run because it relies only on the Python standard library and works with any job scheduler or filesystem. Mrs programs are fast because Mrs is the product of multiple significant rewrites to improve efficiency and reduce overhead, and Python makes such restructuring manageable. Furthermore, Python provides powerful approaches for accelerating programs without sacrificing simplicity, such as running in PyPy or integrating with custom C modules. All of these strengths contribute to making Mrs an effective platform for scientific computing.

Mrs offers both subjective improvements and performance improvements over Hadoop, the most popular MapReduce framework. We evaluate the ease of programming and readability by comparing a Mrs program written in Python with a functionally equivalent Hadoop program written in Java. We show the simplicity in running a Mrs job on a supercomputer with a PBS job scheduler relative to a Hadoop job in the same environment. We also demonstrate the performance advantages of Mrs over Hadoop using three applications: WordCount in Project Gutenberg, a collection of 31,173 documents; a computationally intensive estimator of π ranging from 1 to 10^9 samples; and Particle Swarm Optimization, an empirical function optimization algorithm. In all cases, Hadoop exhibits significant overhead. Despite the inherent performance advantage of Java over Python, the Mrs program maintains a significant performance advantage when task times are less than around 32 seconds, which is extended to around 40 seconds when using a C module in the innermost loop and using the PyPy interpreter. This performance advantage is particularly significant in the context of iterative algorithms, where overhead is incurred each iteration.

Section 3.2 reviews the MapReduce programming model and other MapReduce implementations. Section 3.3 discusses the context of scientific computing and the specific needs that it requires of a MapReduce implementation. Section 3.4 describes the programming model and the design of Mrs, including the advantages and challenges of using Python to implement a MapReduce system. Section 3.5 presents results showing the benefits of Mrs over Hadoop.

3.2 Background and Related Work

MapReduce is a functional programming model that is well suited to parallel computation [23]. In the model, a program consists of a high-level map function and reduce function which process key-value pairs. If a problem is formulated in this way, it can be parallelized automatically by the MapReduce framework.

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).

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:

$$\textsf{map} : (K_1, V_1) \to \textsf{list}((K_2, V_2))$$

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:

$$\textsf{reduce} : (K_2, \textsf{list}(V_2)) \to \textsf{list}(V_2)$$

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.

Figure 3.1 shows the task dependencies in a MapReduce operation. Since the map function only takes a single record, all map operations are independent of each other and fully parallelizable. 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

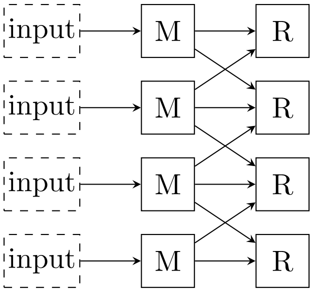


Figure 3.1: Task dependencies of the map (M) and reduce (R) operations in a MapReduce program.

reduce operations are independent of each other and may be run in parallel. The data given to map and reduce functions are generally fine-grained to ensure 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.

These map and reduce functions are sometimes deceptively simple. Even for applications which are simple on the surface, it is inherently difficult to implementing a scalable distributed system with fault-tolerance and load-balancing. In the MapReduce model all of this complexity is found in the surrounding MapReduce framework rather than in the map and reduce functions.

Although not all algorithms can be efficiently formulated in terms of map and reduce functions, MapReduce provides benefits over many other popular parallel processing systems. 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. Those who write mappers and reducers can focus on the problem at hand without worrying about implementation details.

A more complex program may consist of multiple MapReduce stages combined together. In an iterative MapReduce program, the output of each reduce task is the input to a subsequent map task. Figure 3.2 shows the task dependencies in an iterative MapReduce operation.

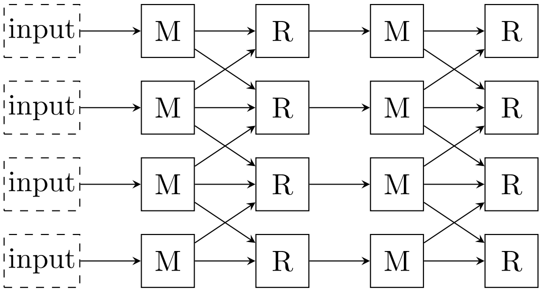


Figure 3.2: Task dependencies of the map (M) and reduce (R) operations in an iterative MapReduce program.

Iterative programs are more sensitive to the overhead of the MapReduce implementation. The per-iteration overhead is multiplied by the number of iterations, which can number in the tens or hundreds of thousands in some applications.

Most MapReduce implementations have targeted large-scale data processing, though a few systems have focused on improving performance for computationally intensive programs. Google has described some details of its MapReduce implementation in published papers and slides, but the system is private. The Apache Lucene project developed Hadoop, an Java-based open-source MapReduce framework implementation. Hadoop is the largest and most well known MapReduce implementation, and though it is primarily designed for large-scale data processing, it has also been used for computationally intensive tasks. HaLoop [37] is an example of a project intended to improve the performance of Hadoop for iterative programs. Twister [38] is an alternative MapReduce implementation designed to improve performance of iterative programs with some sacrifice of fault tolerance. Hadoop remains the most well known and widely available MapReduce system.

3.3 MapReduce in Scientific Computing

Most MapReduce systems are designed to operate in racks of computers dedicated to MapReduce processing, but scientific applications use a wide range of computing resources. A company such as Google or Yahoo with large datacenters might have many thousands of MapReduce systems and thousands of jobs per day, and dedicated systems in datacenters are appropriate in this situation. However, many different types of clusters are used for scientific computing. Shared clusters, which are generally large supercomputers, use batch scheduling systems to coordinate jobs submitted by a wide variety of users. Private clusters, consisting of a smaller number of commodity workstations or temporarily provisioned cloud nodes, are used by a single user at a time and do not require a scheduler. Shared and private clusters are different from dedicated MapReduce clusters. A shared cluster has many users, each of which has unique software requirements. Supercomputers provide resources that are expected to meet the needs of the majority of users, and any individual user cannot expect MapReduce infrastructure to be available. Likewise, most private clusters have no support staff to set up software. In many cases on both shared and private clusters, an individual MapReduce user must perform installation, configuration, and maintenance of the infrastructure they require.

Since MapReduce systems like Hadoop are designed to operate on dedicated machines, these frameworks implement functionality that may duplicate and conflict with the native equivalents that are provided on a supercomputer, such as a job scheduler and a custom distributed filesystem. Most MapReduce frameworks include a job scheduler, but shared clusters already provide a scheduler such as PBS, and private clusters may not require a scheduler at all. A redundant or unnecessary job scheduler does not introduce irreconcilable conflicts but does add complexity in configuration, maintenance, and running jobs. Likewise, a distributed filesystem is redundant with a supercomputer's high-availability centralized storage. Requiring the use of a particular distributed filesystem adds great complexity in maintenance and may be less robust in this context than the existing storage system. After all, a distributed filesystem expects nodes to be up all the time, but a supercomputer's scheduler kills processes as soon as a job completes. The distributed filesystem may lose all of its data nodes and all associated data within a few seconds. It is inappropriate for a MapReduce system on a supercomputer to use a specialized distributed filesystem or scheduler.

A MapReduce system that is well suited to scientific computing on supercomputers will meet different requirements than a MapReduce system designed for large-scale corporate data processing. Such a system should be easy to install and maintain and should play well with existing infrastructure. Jobs should be easy to submit to a batch scheduling system. Programs should require minimal boilerplate to allow for rapid development. Many scientific programs are dynamic research code rather than stable production software, so the system should make it easy to develop and debug programs on a single workstation or small cluster while scaling up to supercomputers.

The Python programming language presents both advantages and challenges in the context of scientific MapReduce. On the one hand, Python is a full-featured and popular language that maximizes developer productivity. It has a large scientific community, and it is popular for developing both prototypes and production code. It has a full-featured standard library, and most third-party libraries are easy to install in a home directory. Python interfaces with other languages, such as C, C++, and Java (through Jython). PyPy is an alternative Python interpreter that provides high performance, especially for numerical programs. On the other hand, as a highly dynamic language, it does not prioritize performance above all other concerns. However, with careful attention to the language's limitations, it is entirely possible to write an efficient MapReduce implementation in Python.

3.4 The Design and Architecture of Mrs

Mrs is a lightweight MapReduce implementation that works well for scientific computing. It is designed to be simple for both programmers and users. The API includes reasonable but overridable defaults in order to avoid any unnecessary complexity. Likewise, Mrs makes it easy to run jobs without requiring a large amount of configuration. It supports both Python 2 and Python 3 and depends only on the standard library for maximum portability and ease of installation. Furthermore, Mrs is designed to easily run in a varienty of enviornments and filesystems. Mrs is also compatible with PyPy, a high-performance Python interpreter with a JIT compiler that accelerates numerical-intensive programs particularly well.

Mrs does not assume any particular job scheduler and is convenient to run in a variety of different contexts. Starting a job requires merely starting one copy of the program as a master and any number of other copies of the program as slaves. It does not require any running daemons, any configuration files, or any particular network ports. When the master starts, it writes its port to a file (unless a fixed port is specified). A slave needs only the master's address and port to connect. Scripts that automate the startup process are available both for shared clusters such as university supercomputers with many users and for private clusters with a small number of users. The script for shared clusters submits a job to a PBS queue (and is easily adapted for any other batch scheduler). The script for private clusters starts the master and uses pssh (parallel-ssh) to start slaves given a list of hosts. In all cases, configuration consists only of a short list of command-line options.

3.4.1 Programming Model

As a programming framework, Mrs controls the execution flow and is invoked by a call to `mrs.main`. The execution of Mrs depends on the command-line options and the specified program class. In its simplest form, a *program class* has an `__init__` method which takes the arguments `opts` and `args` from command-line parsing and a `run` method that takes a `job` argument. In practice, most program classes inherit from `mrs.MapReduce`, which provides a variety of reasonable but overridable defaults including `__init__` and `run` methods that are sufficient for many simple programs. The simplest MapReduce program need only implement a `map` and a `reduce` method.

Mrs provides several features to make writing, testing, and debugging MapReduce programs easier. First, it can run a program in several different execution contexts to help a programmer track down bugs. Second, it provides a simple mechanism for generating independent streams of pseudorandom numbers to make it easy to ensure that results are deterministic and repeatable. Third, it includes a specialized programming model for high-performance iterative MapReduce algorithms.

Mrs defines several different implementations which define the run-time behavior of a program. The *master/slave implementation* distributes work across a cluster of processors. The *serial implementation* performs all work sequentially on a single processor and makes all work deterministic. The *mock parallel implementation* splits work into the same tasks as would be run in the master/slave implementation but performs all computation on a single processor. Intermediate data between tasks is saved to files which can be helpful for debugging. The *bypass implementation* invokes the program class's optional `bypass` method, which is a simple entry point that avoids almost all of the functionality of Mrs. This implementation makes it easy to share code between a simple serial implementation of a program and the corresponding MapReduce implementation. A program's master/slave, serial, mock parallel, and bypass implementations should all produce identical answers, Differences in behavior between any two implementations, even in stochastic algorithms, indicate a bug in the program or possibly in Mrs.

Mrs provides a mechanism for defining independent streams of pseudorandom numbers. Nondeterministic results fundamentally make debugging difficult and testing impossible. In sequential programs, setting a random seed is a simple way to make stochastic algorithms deterministic. However, in a MapReduce program, setting a fixed random seed at the beginning of each map or reduce task would make all tasks use the same sequence of random numbers. The `mrs.MapReduce` class provides a `random` method that returns a random number generator. The method takes a variable number of integer arguments and ensures that the random number generator is unique for any particular combination of inputs. Because of the large size of the internal state of the Mersenne Twister, the `random` method can accept around 300 arguments that are each 64-bit integers. Mrs makes it easy to generate a unique random number generator in each task or even to create identical random number generators in different tasks that need to duplicate specific calculations.

Mrs is optimized for high-performance iterative algorithms. In most MapReduce systems, there is a significant delay between the end of one iteration and the beginning of the next. Between iterations, a program must retrieve results, check for convergence, and submit a new MapReduce job, which can involve a considerable amount of overhead. Mrs allows a program to queue up map and reduce operations so that each is ready to begin as soon as the previous operation finishes. It can also run operations in parallel if they do not depend on each other. For example, a convergence check can run in parallel with the computation of subsequent iterations. The task scheduler in Mrs also attempts to assign corresponding tasks to the same processor from one iteration to the next, which reduces communication between nodes and latency between iterations. While outside the scope of this work, Mrs includes several other optimizations to improve the performance of computationally intensive iterative algorithms. Support for iterative algorithms allows Mrs to efficiently run iterative algorithms that would otherwise be inappropriate for MapReduce.

3.4.2 Architecture

Mrs owes much of its efficiency to simple design. Many choices are driven by concerns such as simplicity and ease of maintainability. For example, Mrs uses XML-RPC because it is included in the Python standard library even though other protocols are more efficient. Profiling has helped to identify real bottlenecks and to avoid worrying about hypothetical ones. We include a few details about the architecture of Mrs.

Communication between the master and a slave occurs over a simple HTTP-based remote procedure call API using XML-RPC. Intermediate data between slaves uses either direct communication for high performance or storage on a filesystem for increased fault-tolerance. Mrs can read and write to any filesystem supported by the Linux kernel or FUSE, including NFS, Lustre, and the Hadoop Distributed File System (HDFS), and native support for WebHDFS is in progress. For data stored to a filesystem, the writer opens and writes a file and then sends the master the corresponding URL, which is used for any future reads. For data communicated directly, the writer opens and writes a file on a local filesystem, and requests from readers are served by a built-in HTTP server. Though direct communication writes to a local filesystem, small short-lived files are rarely written to disk. Rather, they stay in the kernel's filesystem buffer and are served and removed without ever being flushed.

Python requires a bit more attention to detail than some other languages to properly manage threads. Within each master and slave, Mrs generally uses processes instead of threads because of Python's threading model. The Python language specifies a Global Interpreter Lock (GIL) that prevents multiple threads in a single process from executing at the same time. Because of the GIL, Mrs uses threads sparingly, with their use limited to multiplexed I/O threads. Any threads must be started after all processes have started to avoid any risk of forking while holding a lock. In general, all child threads are configured as *daemon* threads, meaning that they are automatically terminated by Python when the main thread completes. This ensures that a straggling thread does not prevent the program from terminating. In each process, the main thread runs an event loop based on `poll`. Main threads do not wait on locks for extended periods of time because `wait` is not generally interruptible by signals including keyboard interrupts. To avoid such problems and to allow threads to wait on network communication and other threads at the same time, Mrs makes heavy use of pipes. Writing a single byte to a pipe wakes up `poll` in a remote process or thread and causes it to continue through its event loop. Communication between the processes of a master or slave uses Python's `multiprocessing` module which is also based on pipes. Complex Python programs like Mrs are much more robust and easily designed by making greater use of processes and pipes and only sparing use of threads and locks.

3.5 Evaluation

Our objective in creating Mrs was to make MapReduce programming fundamentally more accessible. We have sought to take full advantage of the features and facilities in Python to make Mrs both fast and easy to use. In this section we evaluate our results with the Mrs framework in two ways, first a subjective assessment of programming and running Mrs and second a quantitative assessment of performance and scalability. In both the subjective evaluation and the performance measurements, we will compare Mrs with Hadoop, which is currently the most popular MapReduce framework we know of.

3.5.1 Subjective Assessment

In this subsection we seek to assess how effective Mrs is for program development and for execution. While a software engineering-type comparative study (with multiple groups coding the same application under control circumstances) is outside of the scope of this paper, we present here what we feel is compelling subjective evidence that Mrs is an easier environment the development of MapReduce programs.

The most well known MapReduce example is WordCount, a program which counts the number of occurrences of each word in a document or set of documents. This example problem comes from the original MapReduce paper [23]. For this program, the input and output sets, needed for MapReduce as defined in Section 3.2, 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 generally arbitrarily set to be the line number for the input value. The output key is a word, and the output value is its count.

Program 5 shows the complete Mrs code for the WordCount example. The Mrs implementation follows trivially from the MapReduce approach to the problem described in that paper. The "map" part of the implementation splits the input line into individual words and emits one key-value pair for each word in the input with the word token serving as the key and a constant string representation of the number 1 as the value. In this application, this is the so-called "embarrassingly parallel" part of the program, separate processes can be dispatch to emit these key value pairs for each file or even parts of files without concern that they will conflict with each other.

The MapReduce framework groups all messages with matching keys, via a sort step. The framework passes the key and a list of all of the values with that key to the reduce part of the application. The reduce function in the WordCount example, also shown in Algorithm 5, takes a word (the key) and the 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.

Although not critical to an understanding of MapReduce nor this example, the MapReduce architecture allows for an interesting optimization. If the map tasks emit a large number of records (as in WordCount), the sort step 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 can function as a combiner without any modifications. In our quantitative results included below, we make use of this optimization in both the Mrs version and the java version.

Program 6 shows the code for the same application but for the Hadoop framework (without the needed imports, to conserve space) taken from the examples included with Hadoop.

The same basic structure is discernible. In this case there is a class to hold the needed map and reduce functions. Java forces exception processing to be more visible than it was in the Python version. Likewise the marshalling of data is verbose. Some of the complexity of the main function is driven by the fact that Hadoop makes more of the job structure visible whereas Mrs finds the needed elements through introspection. Likewise typing in java adds to the complexity of Hadoop programming. It is certainly the case that Hadoop requires

Program 5 WordCount in Mrs/Python

```
import mrs

class WordCount(mrs.MapReduce):
    def map(self, key, value):
        for word in value.split():
            yield (word, 1)

    def reduce(self, key, values):
        yield sum(values)

if __name__ == ’__main__’:
    mrs.main(WordCount)
```

users to know much about how the system works. Whereas Mrs really just needs the map and reduce functions which is the whole point of MapReduce programming.

One might assume that running a Mrs job would be more complex than running a Hadoop job because Mrs generally relies on external systems for job management and communications. Fortunately that is not the case. In the shared cluster context running a Mrs job is quite easy. Program 7 shows the basic elements of a PBS script for running a Mrs MapReduce program. This script and the corresponding Hadoop script have been reduced to show just the minimum script elements required to start MapReduce jobs. Full scripts include additional error handling and output specification. Any environment variables not defined within the scripts are assumed to be set externally.

The Mrs script (Program 7) has four basic parts: finding the network address of the master, starting the master, waiting for the master to start, and starting the slaves.

The corresponding Hadoop (program 8) script has more issues to address because Hadoop was designed to run on dedicated hardware. When trying to simply run as mapReduce program, there are many elements that have to be setup and later shut down. There are 6 major part of this script. As with the Mrs scripts, first the network address must me found. Second, the Hadoop configuration must be setup. Note that these files are oriented to the operations of a dedicated infrastructure, thus in some cases (just one in this case, but it could

Program 6 WordCount in Hadoop (imports omitted)

```
public class WordCount {
 public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
     ) throws IOException, InterruptedException {
   StringTokenizer itr =
      new StringTokenizer(value.toString());
   while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
   }
  }
 }
 public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key,
     Iterable<IntWritable> values, Context context
     ) throws IOException, InterruptedException {
   int sum = 0;
   for (IntWritable val : values) {
    sum += val.get();
   }
   result.set(sum);
   context.write(key, result);
  }
 }
 public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  String[] otherArgs = new
     GenericOptionsParser(conf, args).getRemainingArgs();
  if (otherArgs.length != 2) {
   System.err.println(”Usage: wordcount <in> <out>”);
   System.exit(2);
  }
  Job job = new Job(conf, ”word count”);
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  FileInputFormat.addInputPath(job,
     new Path(otherArgs[0]));
```

Program 7 Mrs Startup Script (PBS)

```
# Step 1: Find the network address.
ADDR=$(/sbin/ip −o −4 addr list ”$INTERFACE”
    |sed −e ’s;ˆ.∗inet \(.∗\)/.∗$;\1;’)

# Step 2: Start the master.
PORT_FILE=”$JOBDIR/port”
$PYTHON $MRS_PROGRAM −−mrs=Master \
    −−mrs−runfile=”$PORT_FILE” ${ARGS[@]}

# Step 3: Wait for the master to start.
while [[ ! −e $PORT_FILE ]]; do sleep 1; done
PORT=$(cat $PORT_FILE)

# Step 4: Start the slaves.
pbsdsh bash −i −c ”$PYTHON $MRS_PROGRAM
    −−mrs=Slave −−mrs−master=’$ADDR:$PORT’”
```

be worse) configuration files must be edited (see the “sed” line), not just moved into place. Next the daemon processes must be started on the master node (step 3) and so too must the daemons for the slave nodes (step 4). Now the Master task for the MapReduce can be run (step 5). Lastly, in step 6, the daemons on both the master and slaves can be stopped. Note also that since Hadoop requires that data be stored in the Hadoop file system (HDFS) it must be created and formatted as part of this process, which was included as part of step 3. Furthermore, any data to be processed by the MapReduce program must be copied into the HDFS, and likewise data produced, but be copied back out before the HDFS is deleted. The copying of data in and out of the HDFS is accounted for in step 5. Again, in the context of a dedicated system, many of these steps are not needed but on a shared cluster, they are.

3.5.2 Performance

In this section we will demonstrate how Mrs performance compares to that of Hadoop using three example problems of increasing relevance to scientific computing: WordCount, Pi, and Particle Swarm Function Optimization (PSO). For the first two of these experiments we used

Program 8 Hadoop Startup Script (PBS)

```
# Step 1: Find the network address.
ADDR=$(/sbin/ip −o −4 addr list ”$INTERFACE”
    |sed −e ’s;ˆ.∗inet \(.∗\)/.∗$;\1;’)

# Step 2: Set up the Hadoop configuration.
export HADOOP_LOG_DIR=$JOBDIR/log
mkdir $HADOOP_LOG_DIR
export HADOOP_CONF_DIR=$JOBDIR/conf
cp −R $HADOOP_HOME/conf $HADOOP_CONF_DIR
sed −e ”s/MASTER_IP_ADDRESS/$ADDR/g”
    −e ”s@HADOOP_TMP_DIR@$JOBDIR/tmp@g” \
    −e ”s/MAP_TASKS/$MAP_TASKS/g” \
    −e ”s/REDUCE_TASKS/$REDUCE_TASKS/g” \
    −e ”s/TASKS_PER_NODE/$TASKS_PER_NODE/g” \
    <$HADOOP_HOME/conf/hadoop−site.xml \
    >$HADOOP_CONF_DIR/hadoop−site.xml

# Step 3: Start daemons on the master.
HADOOP=”$HADOOP_HOME/bin/hadoop”
$HADOOP namenode −format # format the hdfs
$HADOOP_HOME/bin/hadoop−daemon.sh start namenode
$HADOOP_HOME/bin/hadoop−daemon.sh start jobtracker
```

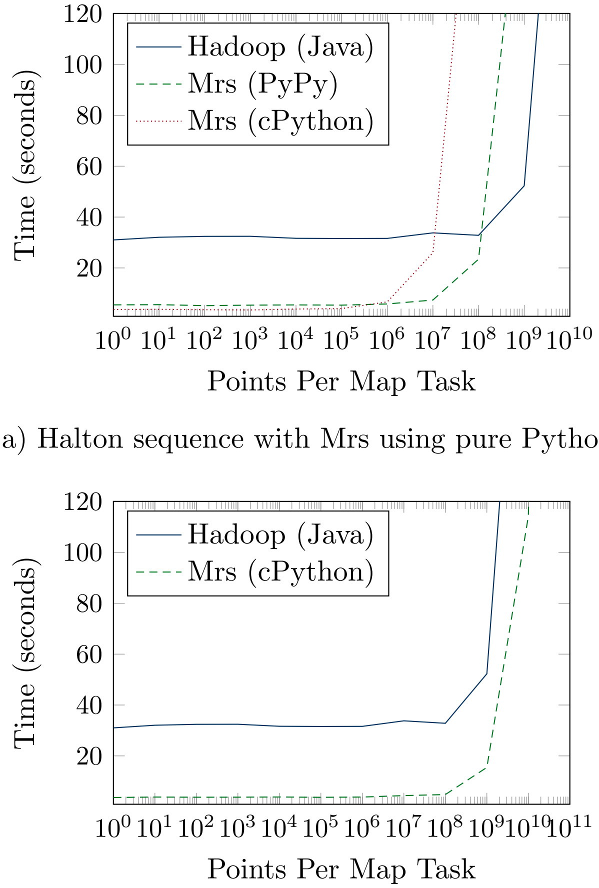
our private cluster of 21 machines, each with 6 cores. For the last set of experiments involving empirical function optimization, we used the Fulton Supercomputing Lab at Brigham Young University. For all experiments using Hadoop we used the Hadoop file system (HDFS) since it is required. For the data intensive WordCount application we also used HDFS, but also tried NFS. The NFS results differ very little (about a second) from those reported. When using HDFS we dedicated one machine as the HDFS name node, Hadoop job tacker and Mrs master. We also assumed that the HDFS was already running for both frameworks. For Hadoop, we ensured that all Hadoop deamons and task trackers were already running. In this way, we measured the performance of the actual MapReduce programs, and not the infrastructure supporting the MapReduce frameworks.

Our first example uses the WordCount problem, as this task is common in MapReduce literature. We use all of the text works from Project Gutenberg, a freely available collection of public domain ebooks (omitting files such as music or Human Genome Project data). Our full dataset of the works available in pure ASCII format includes 31,173 files, for a total of roughly two billion unique word tokens. We utilize HDFS to store this data for both Mrs and Hadoop.

Unfortunately for our comparison, the directory structure from Project Gutenberg is not very amenable to Hadoop. The input file loader for the Hadoop system expects all of the files to be located in a single directory, which is not the case with the Project Gutenberg dataset. With the full dataset, Hadoop struggles to load the data from so many locations, making the start up time alone take nearly nine minutes. In contrast, Mrs is able to perform the entire ReduceMap operation, which included loading the data, counting the words, and aggregating the counts, in under nine minutes. We feel that the directory structure of Project Gutenberg is representative of real world data and that fundamentally Mrs is more flexible in terms of loading data. Note that on a smaller subset of the data with only 8,316 files, Hadoop takes one minute to prepare the data, with a total time of sixteen minutes to finish, while Mrs finishes the entire MapReduce operation in just two minutes.

While WordCount is a canonical MapReduce example, the PiEstimator example in Hadoop is more representative of the numeric, computationally intensive problems encountered in most scientific computation. PiEstimator computes the value of π using a simple Monte Carlo method. While trivial to implement, this task is computational in nature, with no data on disk. This method consists of generating a large number of sample points uniformly distributed on a square with area of 1. An estimation of π / 4 is achieved by multiplying the ratio of points which fall within unit circle centered at a corner of the square to the total number of points. Multipling this value by 4 yields the final approximation of π.

PiEstimator generates random numbers using Halton sequences. While these sequences are entirely deterministic, they are quasi-random. Compared to uniform random numbers, Halton sequences tend to generate numbers which cover the sample space more evenly, which can lead to better results in certain types of Monte Carlo simulations. In all languages, the



(a) Halton sequence with Mrs using pure Python.

(b) Mrs with the inner loop implemented in C.

Figure 3.3: Run times for estimating the value of π. The left hand side of the plots indicates that Mrs has significantly less overhead than Hadoop. The right hand side shows the performance of the numerical code, which is exponential due to the log scale. The algorithm is identical in all cases, so the differences reveal the performance penalty of each programming language.

implementation of the Halton sequence is optimized to minimize the number of function calls and the number of comparison operations.

Figure 3.3 shows the results using Hadoop, Mrs with Python, Mrs with PyPy, and Mrs using `ctypes` to call a C function. We see two interesting trends. On the left-hand side of the graph, we see that Mrs significantly outperforms Hadoop, regardless of the choice of Python interpreter. This can be attributed to the high overhead inherent in using the Hadoop framework. For this problem, in human terms, it may not matter that the task completed in two seconds verses thirty seconds However, as we will discuss shortly, many scientific applications are iterative in nature, and this cost in overhead is multiplied by the number of iterations, making this a strong advantage of the Mrs framework over Hadoop. As we look to the right hand side of Figure 3.3a, we see that the excellent numeric performance of Java begins to win out over pure Python. This can be attributed to the static nature of Java and the high quality of the Java JIT. While not unexpected, this does highlight a weakness of using pure Python for scientific computing.

However, Python makes it easy to rework existing code so that performance critical parts of an application, such as the inner loop of our map tasks, can be rewritten in C. For our second experiment approximating the value of π, we use Python's `ctypes` module to call a C function instead of the the pure Python implementation of the Halton sequence to uniformly generate random points. In this way we were able to very easily replace the inner loop of our map task with optimized C code, while leaving the rest of the loop unchanged. Figure 3.3b shows the results. Once again we see on the left that Mrs has extremely low overhead compared to Hadoop. However, the C function is much faster than the corresponding Java function, so Mrs is much faster than Hadoop, despite the vast majority of Mrs code being in Python.

This experiment does show a key advantage of Python over other languages like Java—Python is designed to easily interface with other languages. We assert that the speed of Python will rarely be the true source of performance problems in the Mrs framework. Instead, thoughtful consideration of algorithms, coupled with profiling and careful optimization will yield the most improvement. In essence, Python allows us to quickly implement scientific application code, and then easily convert any critical paths to C. Java on the other hand, suffers somewhat in this respect.

Our final experiment is an optimization technique used in actual scientific computing. Particle Swarm Optimization (PSO) is an empirical function optimization algorithm inspired by simulations of flocking behaviors in birds and insects [15, 39]. The algorithm simulates the motion of a set of interacting particles within a multidimensional space. At each iteration,

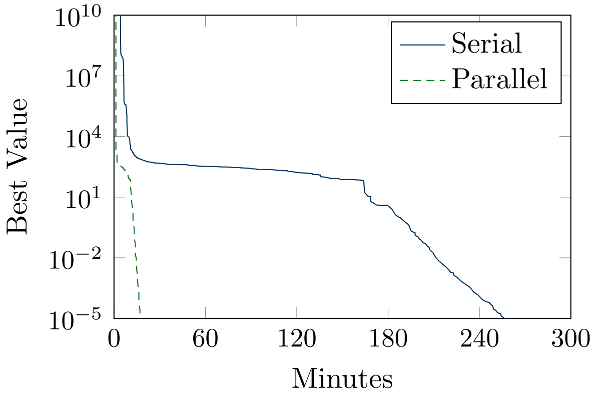


Figure 3.4: Convergence plots of the Apiary topology for the Rosenbrock-250 function with respect to function evaluations and time.

a particle moves and evaluates the objective function at its new position. A particle is drawn toward the best value it has seen and the best value that any of its neighbors has seen. PSO can be naturally expressed as a MapReduce program, with the map function performing motion simulation and evaluation of the objective function and the reduce function calculating the neighborhood best by combining the updated particle with messages from its neighbors [36]. For computationally trivial objective functions, task granularity can be too fine if each map task operates on a single particle. In this case, a swarm can be divided into several subswarms or islands, and each map task operates on several iterations of a subswarm of particles [20, 25, 40].

Using the "Apiary" approach for subswarming [40], Figure 3.4 shows the results for the well-know Rosenbrock benchmark function in 250 dimensions ("Rosenbrock-250") with both serial and parallel computation. Performing 100 iterations on 5 particles requires only 0.2 seconds, and parallel PSO took about 0.5 seconds per iteration. Note that Mrs took advantage of the the features for iterative MapReduce that we previously mentioned. Furthermore, this figure includes only the overhead between iterations, and not the start up time for Mrs (which is about 2 seconds). With any realistically expensive function, the overhead of 0.3 seconds would be negligible.

While we did not actually run PSO using Hadoop, we can estimate its performance. From the execution in Mrs, we know that for the Rosenbrock-250 function PSO took an average of 2471 iterations to reach the target value of 10^{-5}. From our experiments with calculating π, we know that Hadoop takes approximately 30 seconds per iteration. Thus Hadoop would take approximately $2471 * 30$ seconds or a little longer than 20 hours to achieve the same convergence. While we realized that this figure is a rough estimate, our experience with Hadoop suggest that for iterative tasks of this nature, the overhead of Hadoop often makes it slower than running the same task in serial on a single machine.

3.6 Conclusion

The concept of MapReduce has allowed many users to express their scientific computations in an easily parallelizable way. However, the complexity of existing MapReduce frameworks often presents a significant programming burden. Furthermore, most existing MapReduce frameworks such as Hadoop are optimized for performing high volume data analysis rather than solve numerically intense problems. We have presented Mrs as a MapReduce framework which not only eases the programming barrier to entry, but is highly efficient in a scientific context.

Mrs is particularly well suited to an academic or research environment. Many universities and research institutions have supercomputer clusters, or private clusters but these are often generic in nature, not tied to any particular problem or parallel processing technology. Existing frameworks such as Hadoop which require a dedicated cluster and extensive configuration are not always suitable for researchers. Mrs on the other hand has proven exceptionally easy to install and use in a wide variety of environments, scheduling systems, and filesystems.

The choice of Python as our implementation language also aids researchers. Python is a language which naturally lends itself to readable and maintainable code. The syntax is clear and powerful, allowing users to quickly develop, test and deploy scientific applications. Furthermore, Mrs itself takes advantage of Python to make writing MapReduce programs easier. As we have demonstrated, Python lends itself to optimization, without sacrificing code quality by allowing bottleneck portions to be converted to C without affecting any other Python code.

Furthermore, the performance characteristics of Mrs are tailored for scientific computing, where overhead can be more of a significant issue. In particular, the low overhead of Mrs has improved our ability to tackle iterative evaluations such as empirical function optimization. While it was beyond the scope of this paper, future work on Mrs will include additional features which will further improve iterative MapReduce. We have developed a model for iterative MapReduce which allows for efficient implementation. In addition, we have developed novel operations, which lower overhead by significantly reduce the need for communication compared to traditional MapReduce systems.

Chapter 4

High Performance MapReduce for Iterative and Asynchronous Algorithms

This chapter challenges assumptions embodied in most MapReduce systems that limit performance for iterative algorithms. The proposed changes to the MapReduce model and implementations dramatically improve performance for iterative programs and make MapReduce an ideal platform for optimization research.

Abstract

The MapReduce model is designed for large-scale data processing, but its benefits, such as fault tolerance and automatic message routing, are also helpful for computationally-intensive iterative algorithms. Unfortunately, these algorithms perform poorly when implemented in typical MapReduce implementations such as Hadoop.

We propose four modifications to MapReduce to improve performance for iterative programs: First, we combine direct task-to-task communication with strategic use of a distributed filesystems to improve performance while preserving fault tolerance. Second, we combine the reduce and map tasks which span successive iterations to eliminate unnecessary communication and scheduling latency. Third, we propose a generator-callback programming model to allow for greater flexibility in the scheduling of tasks. This allows operations typically found in iterative programs, such as convergence checking and output to be scheduled less frequently and outside of the regular MapReduce cycles. Finally, some iterative algorithms are naturally expressed in terms of asynchronous message passing, and we propose a fully asynchronous variant of MapReduce.

We show that these enhancements yield significant performance improvements in the context of two iterative applications: particle swarm optimization (PSO) and expectation maximization. For example, the combined reduce-map operation alone improves performance from 0.79 to 0.55 seconds (30.7%) per iteration for PSO. Asynchronous MapReduce—even with nearly uniform task times—provides a further improvement of 32% on 768 processors.

4.1 Introduction

Iterative programs exhibit poor performance in most MapReduce [23] frameworks, yet MapReduce remains popular for such applications. It has been used for iterative algorithms such as k-means [16], logistic regression [17], backpropagation [17], independent component analysis [17], expectation maximization (EM) [17], support vector machines [17], genetic algorithms [18], and particle swarm optimization (PSO) [36]. The popularity of MapReduce may be attributed to its simplicity and availability. Unfortunately, such convenience comes with a significant performance penalty [16–18, 36, 38, 41].

Iterative programs are more sensitive to per-iteration overhead than single-pass data processing algorithms are. While a thirty second overhead to start up a MapReduce job may be insignificant in a web indexing program, it adds hours to the execution time of a program that runs for thousands of iterations. If the parallel part of the work requires tenths of seconds and the per-iteration overhead is tens of seconds, then the overhead is two orders of magnitude larger than the work to be done. Several specific problems limit the performance of iterative programs in traditional MapReduce frameworks. First, most MapReduce frameworks write all intermediate data to fault-tolerant distributed filesystems, which is slow, or never write to reliable storage, which sacrifices fault tolerance. Second, separate reduce and map tasks between the end of one iteration and the beginning of the next require unnecessary communication and scheduling latency. Third, submitting operations one at a time adds a delay between iterations and precludes the runtime system from performing operations concurrently. Specifying a sequence of operations including computations like convergence checks is unnatural and inefficient. Finally, some iterative algorithms are expressed in terms of asynchronous message passing, but MapReduce requires synchronous communication. Algorithms such as asynchronous parallel PSO [26, 27] are not expressible in the standard MapReduce programming model.

We contribute techniques for improving the performance of iterative MapReduce programs and an extension to the programming model to support asynchronous algorithms. Three specific improvements for synchronous programs, discussed in Section 4.3, are applicable to any MapReduce implementation. First, we argue for only occasionally checkpointing to reliable storage while using direct communication between nodes for most iterations (Section 4.3.1). Second, we propose that a reduce tasks be agglomerated with the subsequent map task with the same key, which reduces communication and halves the number of tasks that must be assigned each iteration (Section 4.3.2). Third, we present a generator-callback model for submitting operations for concurrent and asynchronous evaluation (Section 4.3.3). This model makes it easy for iterative algorithms to submit intermittent operations, such as convergence checks, to be evaluated concurrently without requiring significant bookkeeping. Finally, we introduce an asynchronous extension of the MapReduce programming model in Section 4.4 which efficiently supports algorithms such as PSO where iteration can proceed at a different rate for each key. This model allows the same straightforward map and reduce functions to work in both synchronous and asynchronous operation.

Although our objective is to motivate, describe, and advocate the widespread use of these approaches, we also demonstrate them in the context of the Mrs [35] MapReduce framework. Section 4.5 evaluates the performance of Mrs with and without these features, using PSO [36] and EM [17] as examples, and shows significant improvements in performance. Compared to standard MapReduce, using a reduce-map operation improved PSO performance by 31%. For EM, iterations without checkpointing to redundant storage show a 91% improvement, making parallelization feasible, and the reduce-map operation gives an extra 11%. Asynchronous MapReduce improves performance of PSO by an additional 24% in the presence of moderate variability in task execution times for a total gain of 53%. Furthermore, it performs iterations faster than synchronous PSO even when task execution times are uniform. With 768 processors and uniform tasks, Asynchronous MapReduce increases the throughput by 47%.

4.2 Related Work

The MapReduce [23] parallel programming model was originally intended for large scale data processing. Hadoop is the most well known and widely used open source implementation of MapReduce. A variety of frameworks based on or related to MapReduce are designed for computationally intensive and iterative programs. Some improvements make the programming model more powerful or convenient, while others improve performance, either through communication or scheduling.

MapReduce is technically defined as a map phase followed by a reduce phase, and this model must be extended, at least trivially, to support iterative programs. In most MapReduce systems, a "user program" or "driver" submits a job consisting of a map phase and a reduce phase, waits for it to complete, reads the results, and then repeats. Several MapReduce-like systems allow the user to specify an arbitrary directed acyclic graph of data dependencies [41–43]. This flexible approach has not yet caught on in MapReduce systems, with the exception of FlumeJava [44], which includes an optimizer to automatically combine arbitrary primitive operations into a smaller number of "MapShuffleCombineReduce" operations. Twister [38] introduces a "combine" phase for collecting the results of a reduce operation at the end of each iteration, but the combine phase, along with the associated delay before submitting the job for the next iteration, adds to the per-iteration overhead. HaLoop [37] introduces a model for specifying a list of map and reduce functions to be executed each iteration but does not allow an arbitrary directed acyclic graph of data dependencies or an arbitrary stopping criterion.

Some MapReduce implementations improve performance by reducing the cost of communication between nodes, which can be significant in iterative programs. Google's MapReduce implementation [23] and Hadoop both use distributed filesystems (GFS [45] and HDFS [46], respectively) for communication between nodes, but this style of communication presents significant per-iteration overhead. Spark [41] and HaLoop [37] pay the cost of writing to a distributed filesystem but try to avoid the cost of reading by caching values in RAM and scheduling tasks accordingly. Twister [38] pushes intermediate data directly from map tasks to reduce tasks and stores data on the master between reduce and map tasks. This enables rollbacks to the previous iteration in the case of failure, but it requires all intermediate data between a reduce task and the subsequent map task to be stored on the master. Furthermore, the approach requires sufficient RAM on each processor to store input to pending reduce tasks.

Some MapReduce implementations make the scheduler more efficient. One simple and effective optimization is to schedule each task to the node which contains its input data. Twister [38] uses static scheduling to preserve locality, but dynamic locality-aware scheduling of each task to a processor which holds its input is more flexible but still simple [37]. Locality-aware dynamic scheduling even scales to multi-user environments [47]. Preserving locality reduces unnecessary communication between processors and can even allow intermediate data to be cached in RAM in unserialized form [41]. Scheduling is extended by iHadoop [48], which allows the convergence check of HaLoop to run concurrently with the next iteration and allows the output of a reduce task to stream to the subsequent map task. In iMapReduce [49], map and reduce tasks are scheduled statically to reduce communication and eliminate the overhead of starting new tasks. In both iHadoop and iMapReduce, a map task may be scheduled "asynchronously," before other reduce tasks have completed.

We build on and extend those observations made in related work. A directed acyclic graph of data dependencies adds great flexibility [41–44]. Direct communication is faster than communicating through a distributed filesystem [38]. Scheduling tasks to the processors which already store the input data reduces communication [37]. The data produced by each reduce task is communicated to a single map task [48, 49]. In many problems, the key in map and reduce tasks refers to a persistent object from iteration to iteration [49]. In addition to these observations from related work, we observe that in the case of persistence, some algorithms may perform more iterations for some keys than for others. We distinguish the novel Asynchronous MapReduce programming model, where iterations do not proceed in lockstep, from asynchronous scheduling [48, 49] which is well known.

4.3 Synchronous MapReduce

Most improvements to frameworks for synchronous MapReduce programs either reduce communication costs or improve scheduling. Iterative programs are sensitive to overhead such as communication costs because such overhead accumulates from iteration to iteration. We propose three improvements to reduce overhead. Section 4.3.1 shows a principled approach for limiting the frequency of checkpoints to distributed storage. Section 4.3.2 describes a reduce-map operation for agglomerating reduce and map tasks. Section 4.3.3 defines a generator-callback model for defining a directed acyclic graph of operations in an iterative program. These three improvements are evaluated later in the paper in Section 4.5.

4.3.1 Infrequent Checkpointing to Distributed Filesystems

Traditional MapReduce implementations communicate all intermediate data through a distributed filesystem. Such filesystems replicate all data to ensure fault tolerance but come with a significant performance penalty. Communication and storage in MapReduce should explicitly address the tradeoff of speed vs. capacity and fault tolerance. An ideal runtime would be able to automatically move data between levels of the memory hierarchy, a well-known strategy for storage devices [50]. While an advanced automatic memory hierarchy may be impractically complex for a MapReduce system, communicating data from some iterations directly between nodes and storing data from other iterations to reliable storage is a simple way to balance speed and fault tolerance.

We advocate storing the output of most map and reduce tasks on the local filesystem, while storing the output from occasional checkpoint iterations to reliable storage. The operating system buffers data on the filesystem in RAM and automatically migrates it to disk if necessary. With fast iterations, short-lived intermediate data is usually deleted before ever being written to disk. This approach provides the speed of RAM when possible and gracefully sacrifices speed for capacity when the size of data is great. In the event that a node fails and makes its local storage unavailable, a MapReduce runtime can roll back to the most recent checkpoint iteration.

In almost any realistic iterative program, checkpointing should occur far less than every iteration, unlike most MapReduce systems, including Hadoop. Some other implementations, like Twister [38], go to the other extreme and do not support distributed storage, sacrificing fault tolerance. The ideal checkpointing frequency depends on the expected cost of failures vs. the cost of redundancy. We estimate and compare these costs using a simple model. While specific circumstances may warrant a customized model to determine the ideal checkpoint frequency, this simple model gives a rule of thumb and demonstrates the cost of checkpointing every iteration.

In this simple model, failures are assumed to be independent. We also assume that the times required to compute an iteration, perform a checkpoint, or initiate a recovery are constant. Let n be the number of iterations between checkpoints, t the time to perform each iteration, c the extra time required for a checkpointed iteration, and r the time to initiate recovery after a failure. Let X be a Bernoulli-distributed random variable indicating whether a failure occurs during an iteration, with probability determined by the product of the mean time between failures in a cluster f and the total time per iteration (including the amortized cost of checkpointing):

$$X \sim Bernoulli\left(\frac{1}{f}\left(t+\frac{c}{n}\right)\right)$$

Let $Y \sim Uniform(n)$ be a random variable indicating the number of iterations since the last checkpoint, which is independent of X. Then the expected value of the number of seconds of extra work in an iteration is:

$$E\left[X\left(r+Yt\right)\right] = \frac{1}{f}\left(t+\frac{c}{n}\right)\left(r+\frac{n}{2}t\right)$$

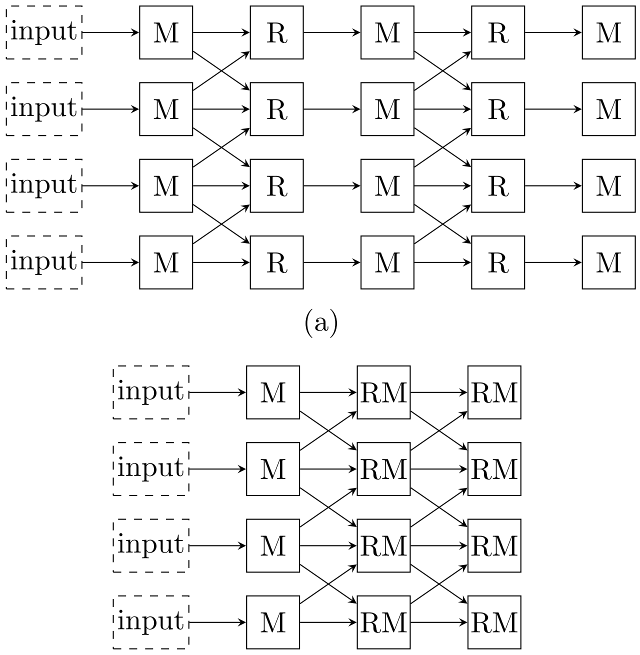
If this is less than the amortized cost of checkpointing per iteration ($\frac{c}{n}$), then redundancy costs more than it helps. The breakeven point is given by solving for n:

$$n = \max\left[1, \frac{1}{t}\left(\sqrt{\left(\frac{c}{2}+r\right)^2 - 2c(r-f)} - \left(\frac{c}{2}+r\right)\right)\right]$$

Most reasonable values cause n to be larger than 1. Suppose that writing to reliable storage adds 10 seconds per iteration ($c = 10$) and that initiating recovery from a checkpoint requires 60 seconds ($r = 60$). Note that the values for c and r are conservative, and increasing c or decreasing r would increase n. For a program with moderately slow one-minute iterations ($t = 60$) and frequent failures on average once every three hours ($f = 10800$), the breakeven point n is 6.7. For a program with fast iterations ($t = 1$) and a moderate failure rate of one failure in a cluster per week ($f = 604800$), the breakeven point n rises to 3413. The ideal frequency of checkpointing depends on individual circumstances, and many short-running programs may not require checkpointing at all.

4.3.2 Reduce-map Operation

Iterative MapReduce programs consist of a string of iterations, each with a map operation and a reduce operation. The new task dependencies between iterations motivate rethinking the decomposition of work into tasks. The output from each reduce task is the sole input to a single map task in the next iteration. Some systems take advantage of this relationship between tasks by scheduling them to the same processor or starting a map task before all preceding reduce tasks are complete [48, 49]. We instead agglomerate each reduce task with



(a)

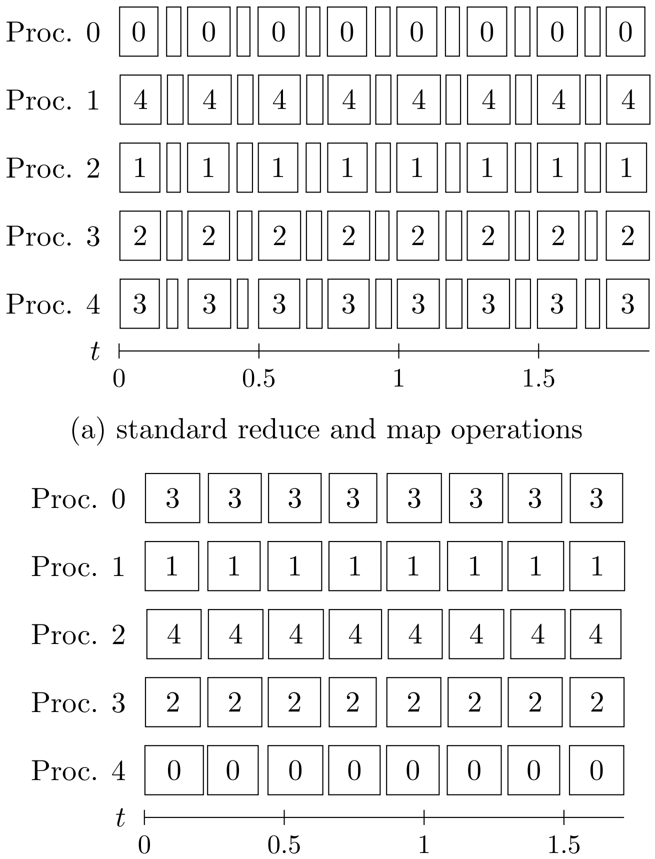
(b)

Figure 4.1: Task dependencies of a typical iterative MapReduce program with (a) standard map (M) and reduce (R) operations, contrasted with (b) combined reduce-map (RM) operations.

the map task that uses its output, which removes this communication and halves the number of tasks that the master must assign each iteration. Figure 4.1 shows the dependencies between tasks with separate reduce and map tasks (Figure 4.1a) and with combined reduce-map tasks (Figure 4.1b).

In principle, the master might be able to autodetect these fine-grained data dependencies, but we allow the user to either specify a reduce-map dataset or separate reduce and map datasets. The user still provides a map function and a reduce function, but specifying a reduce-map operation allows the runtime to combine tasks and eliminate communication.

Figure 4.2 demonstrates the difference between MapReduce using separate reduce and map operations and MapReduce using combined reduce-map operations. Combining the reduce and map eliminates the time spent in assigning each reduce task and waiting for it to complete.



(a) standard reduce and map operations

(b) combined reduce-map operations

Figure 4.2: Actual task execution traces generated from a sample application (particle swarm optimization, see Section 4.5.1.1 for details) without and with combined reduce-map tasks. The run with reduce-map operations avoids the overhead of an independent reduce task and completes sooner. The horizontal axis is measured in seconds, with the left and right sides of each box aligning with the task's start and stop times. The number in each box is the key of the map task.

4.3.3 Iterative Programming Model

The standard MapReduce model defines a single map phase followed by a single reduce phase [23], but iterative programs execute an arbitrary number of operations and often need to compute a loop termination condition that depends on the results. Computing convergence checks infrequently and concurrently with subsequent iterations improves performance, but most MapReduce implementations do not provide any mechanism to specify this behavior. We propose an alternative model for defining operations that allows programs to specify complex behavior without becoming inherently complicated. This model is available for iterative programs that require such behavior but is not required for traditional single-iteration MapReduce programs.

Varying the operations that are performed each iteration—for example, only performing convergence checks or printing intermediate output occasionally—can significantly improve performance. Suppose a program runs in one second per iteration and that evaluating the loop termination condition requires a tenth of a second. If this loop condition computation is performed every iteration, it adds about 6 minutes over the course of an hour. Reducing the check to once per minute extends execution by an average of 30 iterations but still saves about 5 minutes total.

We represent parallel computation with a directed acyclic graph of datasets. A *dataset* represents data to be produced along with the associated operations required to produce it. In the representation of computation as directed acyclic graph, the edges are the work, and the vertices are the data. Such datasets are similar in spirit to resilient distributed datasets [41]. When a user program submits datasets for asynchronous evaluation, the runtime performs computations in any order consistent with the dependency graph. Unlike the lazily evaluated tasks in Ciel [43], these datasets are evaluated eagerly. Because the next iteration can begin before evaluation of the loop condition completes, both operations can be performed concurrently. Likewise, the runtime can begin work on subsequent iterations while a user program is collecting and printing intermediate results. Unfortunately, manually managing a backlog of submitted datasets is tedious and error-prone, particularly if the work varies between iterations.

We propose a generator-callback model for submitting an arbitrary directed acyclic graph of asynchronously evaluated datasets and for handling their completion. The generator-callback model requires the program to provide a `generator` method. The `generator` method serves as an iterator or coroutine that produces work to be done. It submits each dataset for computation, along with an optional callback function to be called when computation completes. The master keeps a backlog of pending datasets, and if the backlog gets full, the generator blocks when it submits a dataset, later resuming when the backlog shrinks. Implementation is especially straightforward in languages which natively support coroutines, such as Python, C#, Go, and Javascript. As each dataset completes, the master calls the associated callback method, which can optionally read and process the results in parallel with subsequent MapReduce iterations. Termination is triggered either by the backlog exhausting after the generator completes or by a callback function returning `False` to indicate that the loop termination condition has been met. This model allows the MapReduce system itself to manage the backlog of datasets rather than exposing the details to the user. Manually maintaining a backlog requires bookkeeping that runs contrary to the simplicity of MapReduce.

Programs using the generator-callback model have greater flexibility and performance. This model is optional but may provide significant benefits for iterative programs that use it. Program 9 is a program which submits one MapReduce step at a time. Unfortunately, the structure of this program forces computation to wait while the master blocks on pending operations, performs the convergence check, and outputs intermediate results. Program 10 uses a generator-callback API to gain flexibility and performance. Note that in this example, an operation is submitted in the form of a declaration of the dataset it is to produce, not the operation itself. The generator function submits several iterations in advance, pausing only when the submit call (or the yield statement if implemented in a language with native support for iterators or coroutines) blocks. This allows tasks to be assigned with lower latency. The generator function also runs convergence checks with limited frequency to reduce overhead. These convergence checks are performed concurrently with subsequent iterations and could be submitted as datasets if they represent significant computation. The simple generator-callback structure makes it easy to specify computation that varies from iteration to iteration and to read data asynchronously as computation completes. Both the blocking program and the generator-callback program use the same simple map and reduce functions.

Program 9 The structure of a generic iterative program using a standard iterative-unaware MapReduce API.

```
run_batches():
    # Intialize key value pairs with empty data.
    init_file = makeTempPath()
    for element_id = 1 to NUM_ELEMENTS
        init_file.writePair(element_id, ””)

    # Perform mapreduce to obtain initial data.
    job = new_job()
    job.setInput(init_file)
    job.setMapper(init_map_func)
    job.setReducer(identity_reduce_func)
    data_path = makeTempPath()
    job.setOutput(data_path)
    job.waitForCompletion()
    last_data = data_path

    # Perform mapreduce iteratively.
    for iteration = 1 to MAX_ITERATIONS
        # Run a mapreduce iteration and wait for a dataset.
        job = new_job()
        job.setInput(last_data)
        job.setMapper(map_func)
        job.setReducer(reduce_func)
        data_path = makeTempPath()
        job.setOutput(data_path)
        job.waitForCompletion()
        last_data = data_path

        # Occasionally output and run convergence check.
        if iteration % CHECK_FREQUENCY = 0
            # Iteration stalls until this completes in serial.
            data = readAllFiles(data_path)
            perform_output(data)
            if converged(data)
                break
```

Program 10 The structure of a generic iterative program using a generator-callback Map-Reduce API for performance and flexibility.

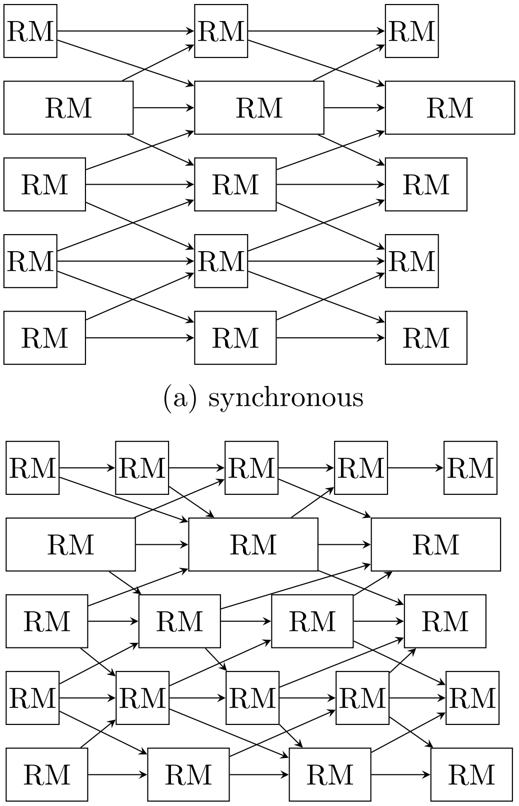
```
generator(queue):
    # Intialize key value pairs with empty data.
    kv_pairs = empty list
    for element_id = 1 to NUM_ELEMENTS
        kv_pairs.append(element_id, ””)

    # Submit request to initialize curr_data.
    curr_data = MapDataset(kv_pairs, init_map_func)
    queue.submit(curr_data, NULL)

    for iteration = 1 to MAX_ITERATIONS
        # Submit asynchronous request to map iterm_data.
        interm_data = MapDataset(curr_data, map_func)
        queue.submit(interm_data, NULL)

        # Submit asynchronous request to reduce curr_data.
        curr_data = ReduceDataset(interm_data,
                                  reduce_func)

        # Occasionally submit output or convergence check.
        if iteration % CHECK_FREQUENCY = 0
            # Iterations continue in parallel with callback.
            queue.submit(curr_data, output_callback)
        else
            queue.submit(curr_data, NULL)

output_callback(data):
    data.readAllFiles()
    perform_output(data)

    # Continue processing if not converged.
    return !converged(data)
```

4.4 Asynchronous MapReduce Programming Model

Iterative MapReduce can serve as a simple message passing framework. A map task serves to update an object, emit it, and emit messages to other objects. Between map tasks and reduce tasks is an implicit barrier for communication to complete, and a reduce task aggregates messages and emits the object, updated with information from the messages. With a reduce-map operation, the second implicit barrier, between the reduce and the following map, is removed. In the context of message passing algorithms, the MapReduce framework conceptually manages all communication, leaving map and reduce functions focused on the essence of the algorithm.

Not all iterative message passing algorithms require a barrier between each map operation and the following reduce. Such algorithms take advantage of all of the messages that have been received so far, and consideration of late-arriving messages is delayed to the next iteration. This class of algorithms is not expressible in the standard MapReduce programming model. Figure 4.3 illustrates task dependencies in an iterative program with heterogeneous task execution times. In synchronous MapReduce (Figure 4.3a), the barrier betweeen iterations leaves the faster processors idle, but in asynchronous MapReduce (Figure 4.3b), the faster processors evaluate more iterations. The benefit can be similar on homogeneous processors if the map and reduce execution times vary or if there are a large number of processors.

We extend the MapReduce programming model to allow asynchronous message passing algorithms. In Asynchronous MapReduce, the programmer may specify that computation of a dataset may begin before all of the tasks in its parent have completed. Unfinished tasks continue execution, and upon completion, their results are added to a subsequent dataset specified by the programmer. The runtime framework keeps track of messages sent to keys with uncompleted tasks and ensures that they do not get lost. These pending messages are included in the same dataset as the results of the task when it eventually finishes. This simple model assumes only that a key refers to a specific object that remains fixed in each iteration.



(a) synchronous

(b) asynchronous

Figure 4.3: Task dependencies for reduce-map tasks in synchronous and asynchronous iterative MapReduce. Asynchronous MapReduce makes much more efficient use of processors.

It works for programs that require multiple map and reduce phases in each iteration, and it is compatible with optimizations like the reduce-map operation.

Adapting a message passing MapReduce program to the asynchronous model requires the programmer to be aware of three new parameters to datasets:

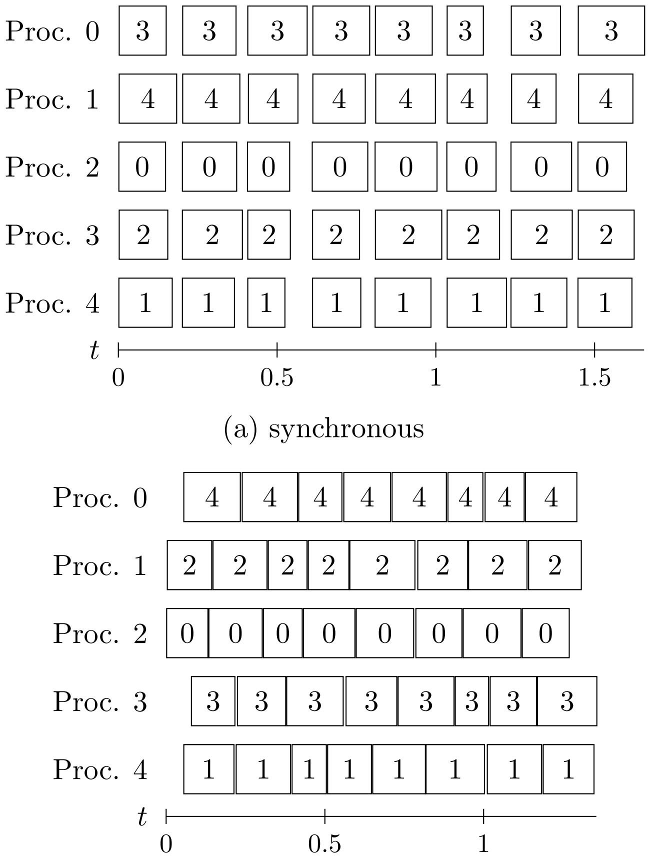
- `async_start`
- `blocking_ratio`,
- `backlink`.

The `async_start` parameter is a boolean indicating whether a dataset can start asynchronously while some tasks in its input are still running. The `blocking_ratio` parameter determines the minimum fraction of tasks that must be completed before any child dataset can start asynchronously and defaults to 1 (fully synchronous). The `backlink` parameter specifies an earlier dataset from which uncompleted tasks are inherited. New tasks are only started for those keys whose corresponding tasks in the `backlink` dataset were completed before any asynchronous execution of its children began.

Although implementation of this model in the runtime framework is not quite trivial, its effect on the map and reduce functions is minimal. The semantics of the map function is unchanged. It still updates an object, emits it, and emits messages. The reduce function, however, is no longer guaranteed to be given the object at every iteration. It might receive only messages intended for the object. In iterations where the reduce function does not receive the object, it can combine messages together, but these messages cannot be incorporated into the object yet. Note that a program that works with Asynchronous MapReduce can also run in traditional synchronous mode.

Particle swarm optimization (PSO), described in more detail in Section 4.5.1.1, is an example of a simple iterative message passing algorithm that is naturally expressed in MapReduce [36]. The map function updates the position of a particle, emits the updated particle, and emits messages to neighboring particles. The reduce function aggregates the messages from neighboring particles, and emits the particle with updated information about its neighbors. Asynchronous parallel PSO is a variant of PSO which allows the evaluation of a particle to proceed even if messages have not been received from all of its neighbors [26, 27]. The fully distributed variant of asynchronous parallel PSO makes its message passing nature particularly clear [51].

Adapting a MapReduce implementation of parallel PSO to the asynchronous model requires very few changes. The reduce function must be tolerant of input that includes several messages but no complete particle; in this case it simply emits the best message. Assuming that this case is correctly handled, the map and reduce functions are identical to those in the synchronous MapReduce PSO implementation. The driver must be updated only to include the asynchronous MapReduce parameters. The map dataset at each iteration must be specified with a `blocking_ratio` below 1 and with a `backlink` pointing at the map dataset



(a) synchronous

(b) synchronous

Figure 4.4: Actual task execution traces for PSO with synchronous and Asynchronous MapReduce. The horizontal axis is measured in seconds.

from the previous iteration. The reduce dataset at each iteration must be specified with the `async_start` parameter set to true. In the case that a single reduce-map dataset is used, it must be given all of these options.

Figure 4.4 shows the improved efficiency of Asynchronous MapReduce compared to synchronous MapReduce for tasks with variable execution times. In synchronous MapReduce, all tasks in an iteration start at the same time, which is limited by the end time of the slowest task in the previous iteration. In Asynchronous MapReduce, each task can start as soon as the corresponding task from the previous iteration completes. Also note that the time between tasks is slightly less in asynchronous MapReduce, presumably due to the load on the master and the traffic on the network being less bursty.

4.5 Experimental Results

Although the approaches described in this paper are applicable to any MapReduce implementation, we evaluate their effects using the Mrs [35] framework. Experiments are performed on two clusters: a 2560-core cluster of 320 nodes, each with two quad-core 2.8 GHz Intel Nehalem processors and 24 GB of memory, and a 150-core cluster of 25 nodes, each with a 6-core 3.2 GHz AMD Phenom II X6 1090T processor with 16 GB of RAM, We run Mrs with and without various techniques enabled, compare the average time per iteration, and measure the average parallel efficiency per iteration. Parallel efficiency is the speedup per processor, relative to the fastest serial algorithm [32], for which we use typical serial implementations.

4.5.1 Synchronous MapReduce

In addition to the serial baseline, we compare with a baseline parallel configuration. This configuration uses redundant storage and convergence checks in serial every iteration, as is common in most MapReduce frameworks, but it also performs some optimizations, such as locality-aware scheduling, which are unavailable in some frameworks.

Although most users will wish to use redundant storage and perform convergence checks, these do not need to be run every iteration. Even if the occasional iteration cannot take advantage of the improved performance, the majority of iterations are accelerated. Section 4.5.1.1 describes particle swarm optimization (PSO) and shows the parallel efficiency of parallel PSO in MapReduce with the cumulative effects of direct communication, concurrent convergence checks, disabled convergence checks, and combined reduce-map tasks. Section 4.5.1.2 describes the EM algorithm and shows similar cumulative improvements.

4.5.1.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an empirical function optimization algorithm inspired by simulations of flocking behaviors in birds and insects [15, 39]. The algorithm simulates the motion of a set of interacting particles within a multidimensional space. At each iteration, a particle moves and evaluates the objective function at its new position. A particle is drawn toward the best value it has seen and the best value that any of its neighbors has seen. PSO can be naturally expressed as a MapReduce program, with the map function performing motion simulation and evaluation of the objective function and the reduce function calculating the neighborhood best by combining the updated particle with messages from its neighbors [36]. For computationally inexpensive objective functions, task granularity is too fine if each map task operates on a single particle. In this case, a swarm can be divided into several subswarms or islands, and each map task operates on several iterations of a subswarm of particles [20, 25].

Program 11 is an implementation of PSO using a generator-callback API as in Program 10 from Section 4.3.3. This is a fully-functional program in Mrs, but it would be very similar in any MapReduce runtime system were it adapted to use a generator-callback API.

We find significant performance improvements for PSO in MapReduce. We use PSO with subswarms of 5 particles applied to the 250 dimensional Rosenbrock function [52]. Each subswarm runs for 50 “subiterations” in each map task. A baseline serial implementation of PSO takes an average of 0.26 seconds to simulate 5 particles for 50 iterations. Note that unlike the parallel implementation, this serial baseline does not serialize the state of particles between iterations. Combining reduce and map operations into a single reduce-map operation significantly reduces the overhead of assigning tasks. With separate reduce and map operations, the average time per iteration is 0.79 seconds. With a combined reduce-map operation, the average time per iteration drops to 0.55 seconds. This represents a reduction of 30.7% in each iteration.

Even a most inefficient MapReduce implementation would be able to provide reasonable parallel efficiency for a large enough problem size, but features that take into account the nature of iterative algorithms are able to extend the range of reasonable performance to more modestly sized problems. Figure 4.5 demonstrates the benefits of several techniques with

Program 11 PSO program using a generator-callback MapReduce API.

```
def run(self, job):
    job.default_reduce_tasks = NUM_PARTICLES
    job.default_reduce_splits = NUM_PARTICLES
    self.check_datasets = set()
    IterativeMR.run(self, job)

def producer(self, job, iteration):
    if iteration == 0:
        kvpairs = []
        for i in range(NUM_PARTICLES):
            kvpairs.append(i, '')
        start_data = job.local_data(kvpairs)
        self.swarm_data = job.map_data(start_data,
                self.init_map)
        start_data.close()
    elif iteration <= MAX_ITERS:
        tmp_data = job.map_data(self.swarm_data,
                self.pso_map)
        self.swarm_data.close()
        self.swarm_data = job.reduce_data(tmp_data,
                self.pso_reduce)
        tmp_data.close()
        if iteration % CHECK_FREQ == 0:
            tmp_data = job.map_data(self.swarm_data,
                    self.collapse_map, splits=1)
            check_data = job.reduce_data(tmp_data,
                    self.findbest_reduce, splits=1)
            self.check_datasets.add(check_data)
    else:
        return []

def consumer(self, dataset):
    if dataset in self.check_datasets:
        self.check_datasets.remove(dataset)
        dataset.fetchall()
        self.output(dataset.data())
        if self.converged(dataset.data()):
            return False
    return True
```

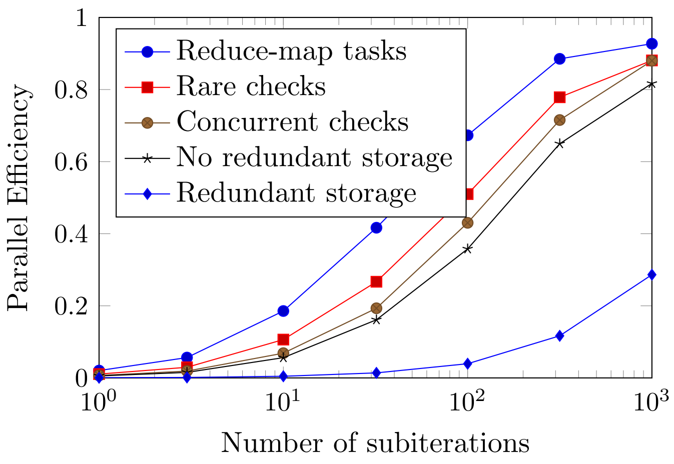


Figure 4.5: Parallel Efficiency (per iteration) of PSO in MapReduce with a sequence of cumulative optimizations. The x-axis represents the problem size (the number of subiterations in each map task). "Redundant storage" represents the baseline performance, with all data stored to a redundant filesystem and with convergence checks occurring after each iteration. "No redundant storage" shows performance for iterations with data communicated directly between processors. "Concurrent checks" shows further improvements when the convergence check is performed alongside the following iteration's work. "Rare checks" avoids unnecessarily frequent convergence checks. Finally, "reduce-map tasks" agglomerates each pair of reduce and map tasks into a single reduce-map task.

respect to the problem size, which in the case of PSO is the number of subiterations performed by each subswarm within each map task. Note that the improvements are cumulative and optional. Though the figure only shows the performance of a reduce-map task in conjunction with direct communication, a configuration using redundant storage would still benefit from using combined reduce-map tasks. Furthermore, a program need not be equally efficient in each iteration. For example, even if redundant storage and convergence checks are performed occasionally, the majority of iterations can benefit from these optimizations. In MapReduce implementations that make redundant storage optional, a program only pays for the level of redundancy it needs.

4.5.1.2 Expectation Maximization

Expectation Maximization (EM) is an iterative algorithm commonly used to optimize parameters of finite mixture models in order to maximize the likelihood of the observed data [53].

Specifically, we apply the algorithm to a mixture of multinomials model in the context of clustering text documents [54, 55]. For each multinomial component in the model, we must maintain vectors with the same dimensionality as the number of features, which can be large. This greatly increases the communication cost when running in parallel, making efficiency difficult to obtain. Other mixture models, such as mixture of Gaussians, have much smaller parameter sizes, and have been parallelized successfully with the EM algorithm [56, 57]. We choose this particular model because it is inherently difficult to parallelize. With redundant storage and convergence checks at every iteration, performance was abysmal. However, with the suggested improvements give much better parallel efficiency.

A single iteration of the EM algorithm consists of two steps. For our model, the expectation step (E-step) uses the current state of the parameters to estimate partial label assignments for the data. This is followed by the maximization step (M-step), which re-estimates the parameters using those partial label assignments. This algorithm is guaranteed to never decrease the log-likelihood of the data and will always converge to a local maximum.

EM for mixture of multinomials can be expressed as a two-stage iterative MapReduce program. The first stage of the program performs the E-step. Each map processes a shard of the documents and computes a posterior distribution given the current state parameters. The reduce then combines the posterior into partial counts for each of the labels. The second stage of the program re-estimates the parameters of the model. The map task performs normalization for each of the labels, and then the reduce tasks combine the normalized counts to produce the updated model parameters.

We tested the MapReduce implementation of EM with the 20 newsgroups dataset, a common benchmark for document clustering [58]. After preprocessing, the dataset had a vocabulary size of approximately 80,000 unique words. As a final step, we applied random feature hashing, which maps each unique word to a predefined number of bins. Although simple, this type of feature selection has been show to perform surprisingly well [59, 60], but

Table 4.1: Parallel efficiency per iteration of EM for various feature set sizes. As expected, higher feature set sizes lead to lower parallel efficiency, but removing redundant storage significantly helps. Further gains are realized by reducing convergence checks and using the reduce-map operation.

Optimization	80	252	8000	25298
Reduce-map tasks	0.411	0.357	0.277	0.193
Rare checks	0.362	0.314	0.253	0.18
Redundant storage	0.013	0.013	0.013	0.012

other more principled dimensionality reductions such as latent Dirichlet allocation [61] could also be used to reduce the feature set size.

Table 4.1 shows the efficiency of parallel EM for various reasonable feature set sizes. Note that as the feature set increases in size, the amount of communication increases at a faster rate than the amount of computation which must be performed for each task, which decreases parallel efficiency. In fact, if one were to do no feature engineering whatsoever and use all 80,000 words as features, the cost of writing this large number of features is so high, that when using a distributed filesystem, the serial implementation of EM runs nearly twice as fast as the parallel version. However, that is not the point here, rather we show that in this application for any reasonable number of features, eliminating the use of redundant storage significantly improves performance. In addition, rare convergence checks in combination with our reduce-map operation brought runtime down from average 83.93 seconds per iteration to only 3.41 seconds, a 95.9% improvement.

4.5.2 Asynchronous MapReduce

The asynchronous programming model of Section 4.4 allows asynchronous parallel PSO [26, 27] to be expressed in MapReduce. This variant of PSO is particularly well-suited for functions whose execution time has high variance, with heterogeneous processors, and in distributed environments [51]. To evaluate the behavior of asynchronous parallel PSO in MapReduce, we vary the number of subiterations performed in each map task.

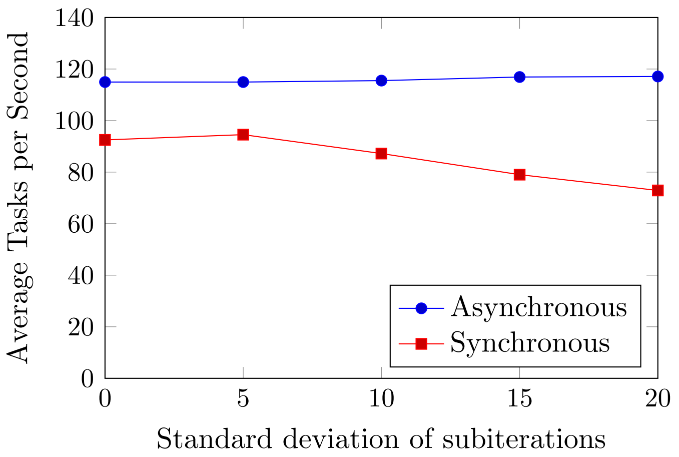


Figure 4.6: The average throughput (in tasks per second) for synchronous and asynchronous PSO. The number of subiterations per map task vary, with an average of 50 and a standard deviation ranging from 0 to 20. Throughput of the asynchronous implementation is unaffected by task variance and is better even when there is no variance.

With a varying number of subiterations, asynchronous parallel PSO is distinctly faster than standard parallel PSO. We draw the number of subiterations from a normal distribution with a mean of 50 and a standard deviation ranging from 0 (no variability) to 20. Figure 4.6 shows the difference in throughput between synchronous and asynchronous PSO in MapReduce as the standard deviation varies. The throughput of asynchronous PSO is fairly constant at around 115 tasks per second. Synchronous PSO, on the other hand, slows as the standard deviation increases, with a throughput of 73 tasks per second when the standard deviation is 20.

Even with small or no standard deviation, asynchronous parallel PSO outperforms the synchronous variant. With a standard deviation of 5, synchronous PSO with combined reduce-map tasks requires an average of 0.58 seconds per iteration, while asynchronous PSO requires only 0.44 seconds. Note that reduce-map operations provide a similar benefit with variance as it does without variance: with separated reduce and map tasks, the time per iteration for synchronous PSO rises to 0.82 seconds.

We speculate that the advantage of Asynchronous MapReduce in the case where task times are uniform is due to a more even load on the master. With synchronous MapReduce, as

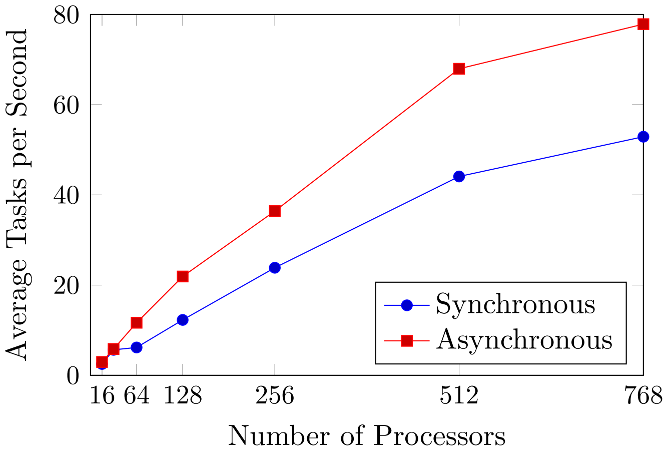


Figure 4.7: The average throughput (in tasks per second) for synchronous and asynchronous PSO with respect to the number of processors. The number of subswarms is equal to the number of processors, and the number of subiterations is 1000.

soon as the last task in a dataset completes, the master is suddenly able to make assignments to each of the slaves. This creates a bottleneck, not only in the master as it makes assignments, but also in the slaves as they all start communicating at the same time. In Asynchronous MapReduce, the master has no such bottleneck because it can make an assignment as soon as a single task completes, without waiting for all other tasks in the dataset to finish. Figure 4.7 explores this phenomenon and shows that the effect increases with the number of processors.

4.6 Conclusion

This paper contributes the following approaches for making MapReduce more appropriate for computationally intensive iterative algorithms:

- Checkpointing: we combine direct task-to-task communication with strategic use of a distributed filesystems to improve performance while preserving fault tolerance.
- The reduce-map operation: this operation is a combination of the reduce and map tasks which span successive iterations. It eliminates unnecessary communication and scheduling latency.

- A generator-callback model for task management: This model provides for both greater flexibility in the scheduling of tasks and better supports operations typically found in iterative programs, such as convergence checking to be scheduled less frequently and outside of the regular MapReduce iterations.

- Fully asynchronous operation: iterative algorithms which are naturally expressed in terms of asynchronous message passing can now be easily expressed and efficiently run.

These approaches improve the efficiency of MapReduce for all iterative algorithms but also make MapReduce feasible for a wide range of applications where its overhead was previously too high to be practical.

Part II

Reconsidering Particle Swarm Optimization in a Parallel Context

The improvements to the MapReduce model described in Part I, along with the associated implementation, provide a flexible and efficient framework for experimenting with a variety of approaches to parallel optimization. The basic MapReduce PSO algorithm, shown in Part I, performs one iteration of PSO per MapReduce stage. Chapter 5 introduces a novel decomposition of the operations of PSO to perform two iterations concurrently in the same MapReduce stage. Chapter 6 presents a PSO topology that organizes particles into subswarms to perform multiple iterations of each subswarm per MapReduce stage, without requiring any additional communication or centralized coordination. Chapter 7 reviews these and other techniques to show how parallel PSO can be applied in a variety of situations and to emphasize the need for considering parallel computation throughout the development of variants of PSO. Together, these chapters demonstrate that effective parallel optimization algorithms can use communication sparingly without requiring centralized coordination.

Chapter 5

Speculative Evaluation in Particle Swarm Optimization

Published in Proceedings of PPSN 2010 in conjunction with Matthew Gardner [62]

This chapter considers a unique approach to parallelizing PSO. While some functions benefit from increased swarm sizes, others show diminishing returns. As an alternative to using additional processors to increase the swarm size, it is possible to perform two or more iterations at the same time with a small swarm. The implementation is built on the adaptation of PSO to Mapreduce described in Chapter 2 and is built on the MapReduce system described in Chapters 3 and 4. This platform made it straightfoward to debug this stochastic algorithm and ensure that it gives numerically identical results to standard parallel PSO.

Abstract

Particle swarm optimization (PSO) has previously been parallelized only by adding more particles to the swarm or by parallelizing the evaluation of the objective function. However, some functions are more efficiently optimized with more iterations and fewer particles. Accordingly, we take inspiration from speculative execution performed in modern processors and propose speculative evaluation in PSO (SEPSO). Future positions of the particles are speculated and evaluated in parallel with current positions, performing two iterations of PSO at once.

We also propose another way of making use of these speculative particles, keeping the best position found instead of the position that PSO actually would have taken. We

show that for a number of functions, speculative evaluation gives dramatic improvements over adding additional particles to the swarm.

5.1 Introduction

Particle swarm optimization (PSO) has been found to be a highly robust and effective algorithm for solving many types of optimization problems. For much of the algorithm's history, PSO was run serially on a single machine. However, the world's computing power is increasingly coming from large clusters of processors. In order to efficiently utilize these resources for computationally intensive problems, PSO needs to run in parallel.

Within the last few years, researchers have begun to recognize the need to develop parallel implementations of PSO, publishing many papers on the subject. The methods they have used include various synchronous algorithms [25, 28] and asynchronous algorithms [26, 27, 29]. Parallelizing the evaluation of the objective function can also be done in some cases, though that is not an adaption of the PSO algorithm itself and thus is not the focus of this paper.

These previous parallel techniques distribute the computation needed by the particles in the swarm over the available processors. If more processors are available, these techniques increase the number of particles in the swarm. The number of iterations of PSO that the algorithms can perform is thus inherently limited by the time it takes to evaluate the objective function—additional processors add more particles, but do not make the iterations go any faster.

For many functions there comes a point of diminishing returns with respect to adding particles. Very small swarms do not produce enough exploration to consistently find good values, while large swarms result in more exploration than is necessary and waste computation. For this reason, previous work has recommended the use of a swarm size of 50 for PSO [39]. Thus, in at least some cases, adding particles indefinitely will not yield an efficient implementation.

In this paper we consider PSO parallelization strategies for clusters of hundreds of processors and functions for which a single evaluation will take at least several seconds. Our purpose is to explore the question of what to do with hundreds of processors when 50 or 100 particles is the ideal swarm size, and simply adding particles yields diminishing returns.

To solve this problem, we propose a method for performing two iterations of PSO at the same time in parallel that we call speculative evaluation. The name comes from an analogy to speculative execution (also known as branch prediction), a technique commonly used in processors. Modern processors, when faced with a branch on which they must wait (e.g., a memory cache miss), guess which way the branch will go and start executing, ensuring that any changes can be undone. If the processor guesses right, execution is much farther ahead than if it had idly waited on the memory reference. If it guesses wrong, execution restarts where it would have been anyway.

We show that the results of standard PSO can be reproduced *exactly*, two iterations at a time, using a speculative approach similar to speculative execution. We prove that the standard PSO equations can be factored such that a set of speculative positions can be found which will *always* include the position computed in the next iteration. By computing the value of the objective function for each of the speculative positions at the same time the algorithm evaluates the objective function for the current position, it is possible to know the objective function values for both the current and the next iteration at the same time. The resulting implementation runs efficiently on large clusters where the number of processors is much larger than a typical or reasonable number of particles, producing better results in less "wall-clock" time.

The balance of this paper is organized as follows. Section 5.2 describes the particle swarm optimization algorithm. Section 5.3 describes how speculative evaluation can be done in parallel PSO to perform two iterations at once. In Section 5.4 and Section 5.5 we present our results and conclude.

5.2 Particle Swarm Optimization

Particle swarm optimization was proposed in 1995 by James Kennedy and Russell Eberhart [15]. This social algorithm, inspired by the flocking behavior of birds, is used to quickly and consistently find the global optimum of a given objective function in a multi-dimensional space.

The motion of particles through the search space has three components: an inertial component that gives particles momentum as they move, a cognitive component where particles remember the best solution they have found and are attracted back to that place, and a social component by which particles are attracted to the best solution that any of their neighbors have found.

At each iteration of the algorithm, the position $\mathbf{x}_t$ and velocity $\mathbf{v}_t$ of each particle are updated as follows:

$$\mathbf{v}_{t+1} = \chi\big[\mathbf{v}_t + \phi^P U_t^P \otimes (\mathbf{x}_t^P - \mathbf{x}_t) + \phi^N U_t^N \otimes (\mathbf{x}_t^N - \mathbf{x}_t)\big] \tag{5.1}$$

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}_{t+1} \tag{5.2}$$

where U_t^P and U_t^N are vectors of independent random numbers drawn from a standard uniform distribution, the $\otimes$ operator is an element-wise vector multiplication, $\mathbf{x}^P$ (called personal best) is the best position the current particle has seen, and $\mathbf{x}^N$ (called neighborhood best) is the best position the neighbors of the current particle have seen. The parameters ϕ^N, ϕ^P, and χ are given prescribed values required to ensure convergence (2.05, 2.05, and .73, respectively) [24].

Changing the way neighbors are defined, usually called the "topology," has a significant effect on the performance of the algorithm. In the Ring topology, each particle has one neighbor to either side of it; in the Complete topology, every particle is a neighbor to every other particle [39]. In all topologies a particle is also a neighbor to itself in that its own position and value are considered when updating the particle's neighborhood best, $\mathbf{x}^N$. Thus with p particles, using the Ring topology each particle with index i has three neighbors: $i-1$, i (itself), and $i+1$. With the Complete topology, each particle has p neighbors.

In this paper we use these topologies as well as a parallel adaptation of the Complete topology, called Random, that has been shown to approximate the behavior of Complete with far less communication [63]. In the Random topology, each particle randomly picks two other particles to share information with at each iteration, along with itself. Thus in both the Ring and the Random topologies, all particles have three neighbors.

5.3 Speculative Evaluation in PSO

The PSO algorithm can be trivially parallelized by distributing the computation needed for each particle across processors. But for some functions adding particles yields diminishing returns. That is, adding processors does not help reach any given level of fitness appreciably faster. Instead of adding particles, speculative evaluation performs iterations two at a time.

Speculative evaluation is made possible by refactoring PSO such that evaluating the objective function is separate from the rest of the computation. For simplicity, this discussion will describe the case where PSO is performing function minimization using the Ring topology. In this example, each particle has two neighbors, the "right neighbor" and "left neighbor," whose positions are represented as $\mathbf{x}^R$ and $\mathbf{x}^L$ respectively. Though we will only describe the case of the Ring topology, this method is easily extended to arbitrary topologies.

The refactoring hinges on the idea that once the random coefficients U_t^P and U_t^P are determined, there are only a few possible updates to $\mathbf{x}^N$ and $\mathbf{x}^P$. For the Ring topology there are 7 possible update cases, identified in Table 5.1. We label each case with an identifier referring to the source of the update: a minus sign $(-)$ represents no update, L represents an update to $\mathbf{x}^N$ coming from the left neighbor, R represents an update to $\mathbf{x}^N$ coming from the right neighbor, and S represents an update to either $\mathbf{x}^P$ or $\mathbf{x}^N$ coming from the particle itself. As an example, $(S,-)$ refers to the case that the particle finds a new personal best, but neither it nor its neighbors found a position that updated its neighborhood best. In the

Table 5.1: All possible updates for a particle with two neighbors

Identifier	Source of $\mathbf{x}^P$ update	Source of $\mathbf{x}^N$ update
$(-,-)$	No update	No update
$(-,L)$	No update	Left Neighbor
$(-,R)$	No update	Right Neighbor
$(S,-)$	Self	No update
(S,L)	Self	Left Neighbor
(S,R)	Self	Right Neighbor
(S,S)	Self	Self

equations that follow, we refer to an unspecified update case as c, and to the set of cases collectively as $\mathcal{C}$.

In order to incorporate the determination of which case occurs into the position and velocity update equations, we introduce an indicator function I_{t+1}^{c} for each case $c \in \mathcal{C}$. When c corresponds to the case taken by PSO, I_{t+1}^{c} evaluates to 1; otherwise it evaluates to 0. We can then sum over all of the cases, and the indicator function will make all of the terms drop to zero except for the case that actually occurs. For example, the indicator function for the specific case $(S,-)$ can be written as follows:

$$\begin{aligned} &I_{t+1}^{(S,-)}\left(f(\mathbf{x}_t), f\left(\mathbf{x}_t^L\right), f\left(\mathbf{x}_t^R\right), f\left(\mathbf{x}_{t-1}^P\right), f\left(\mathbf{x}_{t-1}^N\right)\right) \\ &\quad = \begin{cases} 1 & \text{if } f(\mathbf{x}_t) < f\left(\mathbf{x}_{t-1}^P\right) \text{ and } f\left(\mathbf{x}_{t-1}^N\right) < \min\left(f(\mathbf{x}_t), f\left(\mathbf{x}_t^L\right), f\left(\mathbf{x}_t^R\right)\right) \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{5.3}$$

For each case $c \in \mathcal{C}$, there is also a corresponding velocity update function $\mathbf{V}_{t+1}^{c}$. When the case is known, the specific values of $\mathbf{x}_t^P$ and $\mathbf{x}_t^N$ may be substituted directly into (5.1). For example, in case $(S,-)$, $\mathbf{x}_t^P = \mathbf{x}_t$, as $\mathbf{x}^P$was updated by the particle's current position, and $\mathbf{x}_t^N = \mathbf{x}_{t-1}^N$, as $\mathbf{x}^N$ was not updated at iteration t:

$$\begin{aligned} &\mathbf{V}_{t+1}^{(S,-)}\left(\mathbf{v}_t, \mathbf{x}_t, \mathbf{x}_t^L, \mathbf{x}_t^R, \mathbf{x}_{t-1}^P, \mathbf{x}_{t-1}^N, U_t^P, U_t^N\right) \\ &\qquad = \chi\left[\mathbf{v}_t + \phi^P U_t^P \otimes (\mathbf{x}_t - \mathbf{x}_t) + \phi^N U_t^N \otimes \left(\mathbf{x}_{t-1}^N - \mathbf{x}_t\right)\right] \end{aligned} \tag{5.4}$$

In the same way we can create notation for the position update function by substituting into (5.2):

$$\begin{aligned}&\mathbf{X}^c_{t+1}\big(\mathbf{x}_t, \mathbf{v}_t, \mathbf{x}^L_t, \mathbf{x}^R_t, \mathbf{x}^P_{t-1}, \mathbf{x}^N_{t-1}, U^P_t, U^N_t\big)\\ &\quad = \mathbf{x}_t + \mathbf{V}^c_{t+1}\big(\mathbf{v}_t, \mathbf{x}_t, \mathbf{x}^L_t, \mathbf{x}^R_t, \mathbf{x}^P_{t-1}, \mathbf{x}^N_{t-1}, U^P_t, U^N_t\big)\end{aligned} \tag{5.5}$$

With this notation we can re-write the original PSO velocity equation (5.1), introducing our sum over cases with the indicator functions. The velocity (5.1) and position (5.2) equations become:

$$\begin{aligned}\mathbf{v}_{t+1} = \sum_{c\in\mathcal{C}} \Big[& I^c_{t+1}\big(f(\mathbf{x}_t), f\big(\mathbf{x}^L_t\big), f\big(\mathbf{x}^R_t\big), f\big(\mathbf{x}^P_{t-1}\big), f\big(\mathbf{x}^N_{t-1}\big)\big)\\ & \mathbf{V}^c_{t+1}\big(\mathbf{x}_t, \mathbf{v}_t, \mathbf{x}^L_t, \mathbf{x}^R_t, \mathbf{x}^P_{t-1}, \mathbf{x}^N_{t-1}, U^P_t, U^N_t\big)\Big] \qquad (5.6)\\ \mathbf{x}_{t+1} = \sum_{c\in\mathcal{C}} \Big[& I^c_{t+1}\big(f(\mathbf{x}_t), f\big(\mathbf{x}^L_t\big), f\big(\mathbf{x}^R_t\big), f\big(\mathbf{x}^P_{t-1}\big), f\big(\mathbf{x}^N_{t-1}\big)\big)\\ & \mathbf{X}^c_{t+1}\big(\mathbf{x}_t, \mathbf{v}_t, \mathbf{x}^L_t, \mathbf{x}^R_t, \mathbf{x}^P_{t-1}, \mathbf{x}^N_{t-1}, U^P_t, U^N_t\big)\Big] \qquad (5.7)\end{aligned}$$

In this form the important point to notice is that there are only 7 values (for this Ring topology) in the set $\{\mathbf{X}^c_{t+1} : c \in \mathcal{C}\}$ and that none of them depend upon $f(\mathbf{x}_t)$ or any other objective function evaluation at iteration t. Note also that while there are random numbers in the equation, they are assumed fixed once drawn for any particular particle at a specific iteration. Thus PSO has been refactored such that the algorithm can begin computing all 7 of the objective function evaluations potentially needed in iteration $t+1$ *before* $f(\mathbf{x}_t)$ is computed. Once the evaluation of $f(\mathbf{x}_t)$ is completed for all particles only one of the indicator functions I^c_{t+1} will be set to 1; hence only one of the positions $\mathbf{X}^c_{t+1}$ will be kept.

Although this speculative approach computes $f(\mathbf{X}^c_{t+1})$ for all $c \in \mathcal{C}$, even those for which $I^c_{t+1} = 0$, these extra computations will be ignored, and might just as well never have been computed. We call the set $\{f(\mathbf{X}^c_{t+1}) : c \in \mathcal{C}\}$ "speculative children" because only one of them will be needed.

To see the value of this refactoring, suppose that 800 processors are available, and that the evaluation of the objective function takes one hour. If we only want a swarm of 100 particles, 700 of the processors would be sitting idle for an hour at every iteration, and it would take two hours to run two iterations. If instead we perform speculative evaluation, sending each of the $f(\mathbf{X}^c_{t+1})$ to be computed at the same time as $f(\mathbf{x}_t)$, we could create a swarm of size 100, each particle with 7 speculative evaluations (700 processors dedicated to speculative evaluation), thus using all 800 processors and performing two iterations in one hour.

In order to do two iterations at once, we use 8 times as many processors as there are particles in the swarm. If these processors were not performing speculative evaluation, they might instead be used for function evaluation needed to support a large swarm. This raises the question of whether a swarm of 100 particles doing twice as many iterations outperforms a swarm of 800 particles. We show in the rest of this paper that in many instances a smaller swarm performing more iterations does in fact outperform a larger swarm. We acknowledge that both intuition and prior research [63] indicate that the optimization of deceptive functions benefits greatly from large and even very large swarm sizes. Thus this work will focus on less deceptive functions.

5.3.1 Implementation

The number of speculative evaluations needed per particle depends on the number of neighbors each particle has. In a swarm with p particles and n neighbors per particle, $(2n + 1)p$ speculative evaluations are necessary (each additional neighbor adds two rows to Table 5.1). This dependence on the number of neighbors necessitates a wise choice of topology. The use of the Complete topology, where every particle is a neighbor to every other particle, would require $O(p^2)$ speculative evaluations per iteration. It is much more desirable to have a sparse topology, where $O(np)$ is much smaller than $O(p^2)$. However, some functions are better optimized with the Complete topology and the quick spread of information it entails than with sparse topologies. In such cases, we use the Random topology described in Section 5.2.

To aid in describing our implementation, we introduce a few terms. We use p_t to denote a particle at iteration t and s_{t+1} to denote one of p_t's speculative children, corresponding to one of the rows in Table 5.1. n_t is a neighbor of particle p_t. Sets of particles are given by $\mathbf{p}$, $\mathbf{s}$, or $\mathbf{n}$, whereas single particles are simply p, s, or n.

A particle at iteration $t-1$ that has been moved to iteration t using (5.1) and (5.2), but whose position has not yet been evaluated, is denoted as p_t^{-e}. Once its position has been evaluated, but it has still not yet received information from its neighbors, it is denoted as p_t^{-n}. Only when the particle has updated its neighborhood best is it a complete particle at iteration t. It is then simply denoted as p_t.

The outline of the speculative evaluation in PSO (SEPSO) algorithm is given in Algorithm 12.

Program 12 Speculative Evaluation PSO (SEPSO)

```
 1: Move all $p_{t-1}$ to $p_t^{-e}$ using (5.1) and (5.2)
 2: For each $p_t^{-e}$, get its neighbors $\mathbf{n}_t^{-e}$ and generate $\mathbf{s}_{t+1}^{-e}$ according to (5.5).
 3: Evaluate all $p_t^{-e}$ and $\mathbf{s}_{t+1}^{-e}$ in parallel
 4: Update personal best for each $p_t^{-e}$ and $s_{t+1}^{-e}$, creating $p_t^{-n}$ and $s_{t+1}^{-n}$
 5: Update neighborhood best for each $p_t^{-n}$, creating $\mathbf{p}_t$
 6: for each $p_t$ do
 7:     Pick $s_{t+1}^{-n}$ from $\mathbf{s}_{t+1}^{-n}$ that matches the branch taken by $p_t$ according to (5.7).
 8:     Pass along personal and neighborhood best values obtained by $p_t$, making $p_{t+1}^{-n}$
 9: end for
10: Update neighborhood best for each $p_{t+1}^{-n}$, creating $\mathbf{p}_{t+1}$
11: Repeat from Step 1 until finished
```

5.3.2 Using All Speculative Evaluations

In performing speculative evaluation as we have described it, $2n+1$ speculative evaluations are done per particle, while all but one of them are completely ignored. It seems reasonable to try to make use of the information obtained through those evaluations instead of ignoring it. Making use of this information changes the behavior of PSO, instead of reproducing it exactly as the above method explains, but the change turns out to be an improvement in our context.

To make better use of the speculative evaluations, instead of choosing the speculative child that matches that branch that the original PSO would have taken, we take the child that has the best value. The methodology is exactly the same as above except for the process of choosing which speculative child to accept. The only change needed in Algorithm 12 is in step 7, where the s_{t+1}^{-e} with the best value is chosen from $\mathbf{s}_{t+1}^{-e}$ instead of with the matching branch. We call this variant “Pick Best”.

5.4 Results

In our experiments we compared our speculative PSO algorithm to the standard PSO algorithm. At each iteration of the algorithms, we use one processor to perform one function evaluation for one particle, be it speculative or non-speculative. The speculative algorithm actually performs two iterations of PSO at each “iteration,” so we instead call each “iteration” a “round of evaluations.” For benchmark functions with very fast evaluations this may not be the best use of parallelism in PSO. But for many real-world applications, the objective function takes on the order of at least seconds (or more) to evaluate; in such cases our framework is reasonable.

In each set of experiments we keep the number of processors for each algorithm constant. In all of our experiments SEPSO uses topologies in which each particle has two neighbors in addition to itself. As shown in Table 5.1, this results in 7 speculative evaluations per particle. With one evaluation needed for the original, non-speculative particle, we have a total of $8p$ evaluations for every two iterations, where p is the number of particles in the speculative swarm. In order to use the same resources for each algorithm, we compare swarms of size p in the speculative algorithms with swarms of size $8p$ in standard PSO.

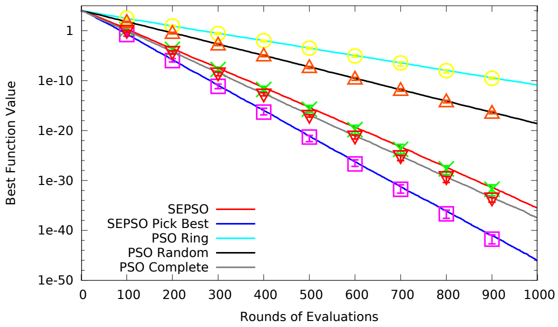


Figure 5.1: Function Sphere with a swarm that uses 240 processors per round of evaluations. We show 10th and 90th percentiles every 100 iterations. Note that PSO Complete requires $O(p^2)$ messaging and may not be practical in many cases.

As discussed in Section 5.3.1, where the Complete topology would normally be used, we use a Random topology in our speculative algorithm, as Complete leads to an explosion of speculative evaluations. For the standard PSO baseline we have included experiments with the Ring and the Random topologies, both with two neighbors, as well as for the Complete topology. It is important to note however, that in many cases the Complete topology is not a practical alternative in the context of large swarms on large clusters where the messaging complexity is $O(p^2)$ and can overwhelm the system.

We follow the experimental setup used in [39]. All functions have 20 dimensions, and all results shown are averages over 20 runs. For ease of labeling, we call our speculative algorithm SEPSO, the variant SEPSO Pick Best, and standard PSO just PSO and identify the topology in a suffix, "PSO Ring" for example.

The benchmark function Sphere ($f(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$) has no local optima and is most efficiently optimized using a small swarm but many iterations. In our experiments with Sphere, we use 240 processors; thus 30 particles for SEPSO and 240 for PSO. In Figure 5.1, we can see that SEPSO clearly beats PSO with a Random topology or a Ring topology. SEPSO approaches the performance of PSO with the Complete topology, even though PSO with the Complete topology requires $O(p^2)$ communication. SEPSO Pick Best handily outperforms all other algorithms in this problem.

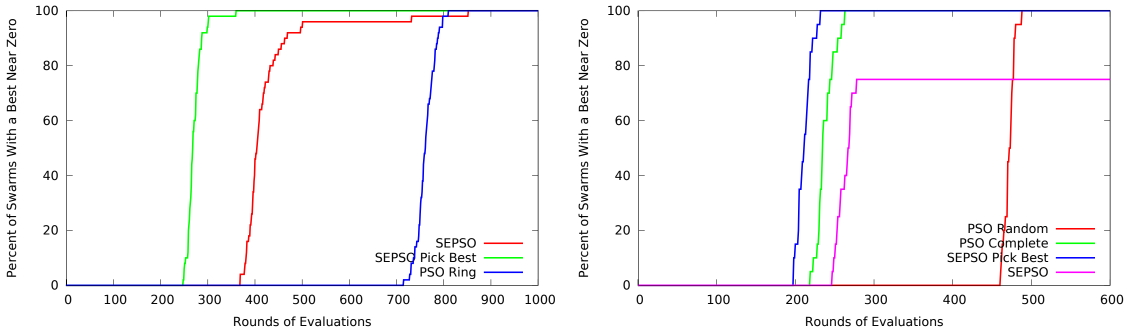


Figure 5.2: Function Griewank with a swarm that uses 800 processors per round of evaluations, and function Bohachevsky with a swarm that uses 480 processors per round of evaluations.

The benchmark function Griewank is defined by the equation $f(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{D} x_i^2 - \Pi_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$. It is best solved in PSO using the Ring topology, as Complete is prone to premature convergence on local optima. Griewank has a global optimum with a value of 0, and sometimes the swarm finds the optimum and sometimes it does not. Instead of showing average function value at each iteration, a more enlightening plot for Griewank shows the percent of runs that have found the global optimum by each iteration.

PSO and SEPSO get caught in local minima with small swarm sizes so we show results in Figure 5.2 for swarms of size 100 (SEPSO) and 800 (standard) using the Ring topology. Figure 5.2 shows that SEPSO quickly finds the global optimum, between two and three times faster than running standard PSO.

In Figure 5.2 we also show results for the Bohachevsky function, defined as $f(\mathbf{x}) = \sum_{i=1}^{D}(x_i^2 + 2x_{i+1}^2 - .3\cos(3\pi x_i) - .4\cos(4\pi x_{i+1}) + .7)$. Bohachevsky is a unimodal function best optimized with a Complete swarm. It is similar to Griewank in that there is a global optimum with a value of 0, and swarms either find the optimum or get stuck. Both SEPSO algorithms find the optimum much faster than PSO Random, though only SEPSO Pick Best beats PSO Complete. Also, while the smaller swarm size of SEPSO gets stuck 75% of the time, when using SEPSO Pick Best with the same swarm size, the algorithm finds the optimum every time.

Previous work has shown that the optimization of highly deceptive functions like Rastrigin $(f(\mathbf{x}) = \sum_{i=1}^{D} (x_i^2 - 10\cos(2\pi x_i) + 10))$ benefit greatly from the addition of particles. Smaller swarms get caught in local optima, up to swarms of at least 4000 particles [63]. Because our speculative algorithms have a significantly smaller swarm size, they get stuck at higher values while the larger swarms performing regular PSO continue to improve the best value found. Our experiments with SEPSO on Rastrigin were predicably lack luster, yielding an average value of 31 after 1000 evaluations, as compared to 10 for standard PSO.

5.5 Conclusions

We have described how parallel implementations of particle swarm optimization can be modified to allow additional processors to increase the number of iterations of the algorithm performed, instead of merely adding more particles to the swarm. Using our modifications, the original PSO algorithm is exactly reproduced two iterations at a time. This technique requires more function evaluations per iteration than regular PSO, but for some functions still performs better when run in a parallel environment. We have also described a method for making use of extra speculative evaluations that performs very well on some functions.

There are some functions for which very little exploration needs to be done; Sphere is an example of such a function. For such functions the best use of processors is to have a small swarm performing speculative evaluation with our Pick Best method, where all speculative evaluations are used.

There are other functions for which it seems there is never enough exploration, such as the Rastrigin function. It has been shown that up to 4000 particles there is no point at which "enough" exploration has been done [63]. With such functions, the smaller swarm size required by speculative evaluation is not able to produce enough exploration to perform better than standard PSO.

Griewank and Bohachevsky are functions between Sphere and Rastrigin. They are deceptive and prone to premature convergence, but by adding particles to the swarm a point is reached where “enough” exploration is done, and the algorithm finds the optimum essentially all of the time. For such functions, the best approach seems to be to increase the swarm size until “enough” exploration is reached, then use extra processors to perform speculative evaluation and increase the number of iterations performed. Sphere and Rastrigin can be thought of as special cases of these types of functions; Sphere simply needs a very small swarm size to produce “enough” exploration, and Rastrigin requires a very large swarm. We expect that for all functions there is a swarm size for which additional particles are less useful than additional iterations.

Large parallel clusters are often required to successfully optimize practical modern problems. To properly use PSO with such clusters, a balance needs to be made between using processors to increase the swarm size and using them to increase the speed of the algorithm. This work is a first step in that direction.

Chapter 6

The Apiary Topology: Emergent Behavior in Communities of Particle Swarms

Published in Proceedings of PPSN 2012 [40]

This paper presents a communication topology for subswarms in PSO which is well suited to parallel computation. For objective functions with fast evaluation times, assigning a single particle per processor requires impractically large amounts of communication, so it is much more efficient to assign a group of particles to each processor. In contrast with other subswarm-style variants of PSO that require centralized coordination of migration, this approach uses topologies, which are part of the standard definition of PSO.

Abstract

In the natural world there are many swarms in any geographical region. In contrast, Particle Swarm Optimization (PSO) is usually used with a single swarm of particles. We define a simple new topology called Apiary and show that parallel communities of swarms give rise to emergent behavior that is fundamentally different from the behavior of a single swarm of identical total size. Furthermore, we show that subswarms are essential for scaling parallel PSO to more processors with computationally inexpensive objective functions. Surprisingly, subswarms are also beneficial for scaling PSO to high dimensional problems, even in single processor environments.

6.1 Introduction

Particle Swarm Optimization (PSO) is a continuous function optimization algorithm inspired by the flocking behaviors of birds and insects. It is typically used with small swarms of 20 to 50 particles organized in simple topologies that do not fully reflect the complex social interactions of insects. In agriculture, for example, bees are managed in sets of hives called apiaries. The number of hives in an apiary usually ranges from 10 to 150.

Using conventional topologies, a single swarm of particles often fails to scale both to large numbers of processors and to high-dimensional problems. First, with a large number of processors and an inexpensive objective function, communication costs make parallel PSO with a single swarm impractical. Parallel PSO naturally works well for problems with computationally expensive function evaluations, but for inexpensive objective functions, the time to communicate a single position can exceed the time to perform a function evaluation. Second, for high-dimensional problems, particles are prone to premature convergence. Even for Sphere, the simplest of benchmark functions, standard PSO struggles to find the global optimum when the number of dimensions is 400 or greater.

Multiple swarms have been used to scale parallel PSO for inexpensive objective functions but have not been considered for scaling to high-dimensional problems. Semi-independent swarms of particles provide a natural way to parallelize the computation of PSO across a set of processors without requiring instantaneous communication [25]. However, the behavior of subswarms has not been explored, particularly with respect to high-dimensional problems.

The Apiary topology, proposed in Section 6.3, spreads the population of particles among a set of small subswarms. In this topology, a subswarm is a social entity which lies between the individual particle and the full population and which serves as another source of emergent behavior. Each subswarm consists of a fixed set of particles and is mostly independent of other subswarms. Periodically, a single particle in each subswarm communicates with a few particles in other subswarms. This communication between subswarms is rare and limited, so computation is particularly well suited to parallel computation.

The Apiary topology helps PSO scale, both to large numbers of processors and to high-dimensional objective functions. Unlike some other proposed PSO techniques using subswarms, this topology is simple, clearly defined, and appropriate for parallel PSO. Experiments, described in Section 6.4, show significant improvements over standard PSO. Even in single processor environments, apiaries produce better results in the same time and are less prone to premature convergence for every benchmark function we tested. These results are presented and discussed in Section 6.4.1. The standard parameters are justified in Section 6.4.2, along with indications of when these parameters might be changed. Parallel PSO with Apiary is compared in Section 6.4.3. Despite inexpensive functions being particularly challenging for parallelization, the run time is reduced from 256 minutes with a single processor to 17 minutes with 40 processors.

6.2 Background Material: Particle Swarm Optimization

Particle Swarm Optimization, proposed by Kennedy and Eberhart [15], simulates the motion of particles in the domain of an objective function. These particles search for the global optimum by evaluating the function as they move. During each iteration, each particle is pulled toward the best position it has sampled, known as the *personal best*, and the best position of any particle in its neighborhood, known as the *neighborhood best*.

Constricted PSO is generally considered the standard variant [39]. Each particle's position $\mathbf{x}_0$ and velocity $\mathbf{v}_0$ 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 $\mathbf{x}_t$ and velocity $\mathbf{v}_t$ with respect to the personal best $\mathbf{x}^P_{t-1}$ and neighborhood best $\mathbf{x}^N_{t-1}$ from the preceding iteration:

$$v_{t,i} = \chi \left[v_{t-1,i} + \phi^P u^P_{t-1,i}(x^P_{t-1,i} - x_{t-1,i}) + \phi^N u^N_{t-1,i}(x^N_{t-1,i} - x_{t-1,i}) \right] \tag{6.1}$$

$$x_{t,i} = x_{t-1,i} + v_{t,i} \tag{6.2}$$

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 $\chi = 2/\left|2 - \phi - \sqrt{\phi^2 - 4\phi}\right|$ where $\phi = \phi^P + \phi^N$ [24].

The neighborhoods within a swarm are defined by the *topology* graph. The choice of topology can have a significant effect on performance [64]. Additionally, the topology determines task dependencies and overhead in parallel PSO [63]. The $Ring_{50}$ topology, a swarm of 50 particles where each particle has a single neighbor on either side, is a standard starting point [39].

6.3 The Apiary Topology

The Apiary topology is a dynamic topology of independent subswarms which occasionally communicate with each other. Each subswarm has an *inner topology*, and the subswarms are connected in an *outer topology*. In most iterations, the neighbors of each particle are defined purely by the inner topology of its subswarm. After a fixed number of independent *subiterations*, each subswarm communicates with its neighboring subswarms, as defined by the outer topology. Each subswarm sends its neighbors the best value from any of its particles. It updates the neighborhood best of a fixed set of particles (the neighborhood of the first particle in the swarm) with the values from neighboring subswarms.

In the Apiary topology, subswarms share important characteristics with communities in nature. Just as each bee colony has its own social structure, each subswarm has its own particles and its own topology. Like bee colonies, the subswarms are independent and rarely interact. Curiously, bees occasionally allow foreign forage bees to enter a hive if they are fully loaded [65], and the native bees will be able to learn from those foreign bees if they are from another colony or even another species [66]. Likewise, a single particle in each subswarm occasionally engages in light communication with neighboring swarms. In this simple structure, subswarms are simple entities with a balance of independence and interaction that favors emergent behavior.

This approach contrasts with previous attempts to define subpopulations in PSO. Dynamic Multi-Swarm PSO [19] periodically shuffles by reassigning all particles to random subswarms. This global reshuffling increases the amount of communication required in parallel PSO and is incompatible with asynchronous parallel PSO [27]. In contrast, neighborhoods in the Apiary topology are deterministic and require very little communication. Section 6.4.1 compares the performance of Dynamic Multi-Swarm PSO with that of the Apiary topology. Most subswarm approaches have introduced strategies—some of them quite complex—to manage the migration of particles between subswarms [20, 67–69]. Other works have used subswarm-style topologies within a limited context [63], including completely independent subswarms [25]. Romero and Cotta's island-structured swarms [20], is limited to small numbers of large subswarms and low-dimensional problems, and its conclusions do not seem to apply to high-dimensional problems. In contrast to other approaches, the Apiary topology is static and thus well suited to any implementation of parallel PSO, and it requires very little communication between subswarms.

The inner and outer topologies, as well as the number of subiterations, are changeable parameters. We recommend *Ring* for both the outer and inner topologies, with a starting point of 5 particles per subswarm, 40 total subswarms, and 100 subiterations. These recommendations are justified in Section 6.4.2.

6.4 Experimental Results

The Apiary topology provides significant improvements for both serial and parallel PSO with respect to a variety of benchmark functions. Benchmark functions are computationally inexpensive enough for large-scale experimentation but share interesting properties with challenging real-life problems. We use the Ackley, Rastrigin, Rosenbrock, Schwefel 1.2, and Sphere benchmark functions [52] with both 250 and 500 dimensions. Experiements were run on a Linux cluster consisting of 320 nodes (Dell PowerEdge M610). Each node is equipped with two quad-core Intel Nehalem processors (2.8 GHz) and 24 GB of memory.

Each experiment was repeated at least 40 times. We report the median instead of the mean because these distributions are skewed. The 10^{th} and 90^{th} percentiles illuminate both the variability and skewness. We determine statistical significance using a one-sided Monte Carlo permutation test [70]. A t-test would be inappropriate because it uses the mean statistic and assumes a normal distribution, which we can not assume in part because of skew. Each table cell is bolded if it is better than every other entry in its row with a p-value of 0.05.

Each table and plot presents either the median number of evaluations required to reach a threshold or the median best value at a fixed number of evaluations or iterations. The notation $Ring_n$ denotes a ring topology where each particle has one neighbor on each side, and $Ring_m$–$Ring_n$ denotes an Apiary topology with a $Ring_m$ outer topology and a $Ring_n$ inner topology. Each benchmark function is accompanied by its dimensionality, for example, "Sphere-500."

The balance of this section seeks to identify some of the most interesting observation and give greater clarity and meaning to these results. Section 6.4.1 compares the $Ring_{40}$–$Ring_5$ apiary with the standard recommendation of $Ring$. Section 6.4.2 justifies the particular choice of $Ring_{40}$–$Ring_5$ as a standard starting point. Finally, Section 6.4.3 demonstrates the suitability of the Apiary topology to parallel PSO by demonstrating its efficiency in a typical parallel environment.

6.4.1 Apiaries in Serial PSO

Limiting the interaction between subswarms to once every 100 iterations might be expected to compromise the performance of serial PSO in exchange for improved parallel efficiency, but this social organization in fact improves performance even in serial PSO. Figures 6.1 and 6.2 show the progress toward convergence for 500 dimensional Rastrigin and Sphere

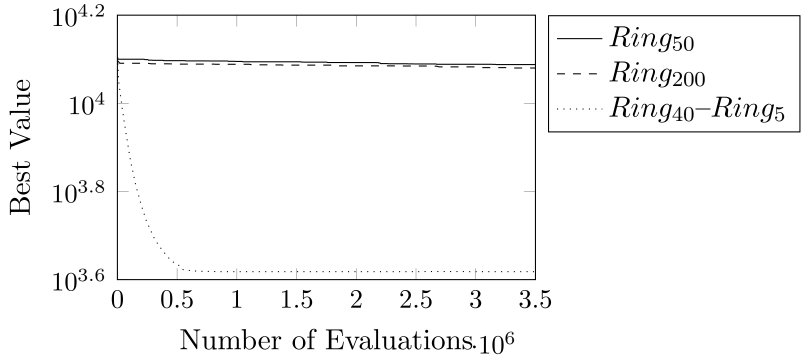


Figure 6.1: Convergence plot for Rastrigin in serial PSO, comparing an apiary (using 100 subiterations) with a swarm of the same total number of total particles (200) and a swarm of 50 particles.

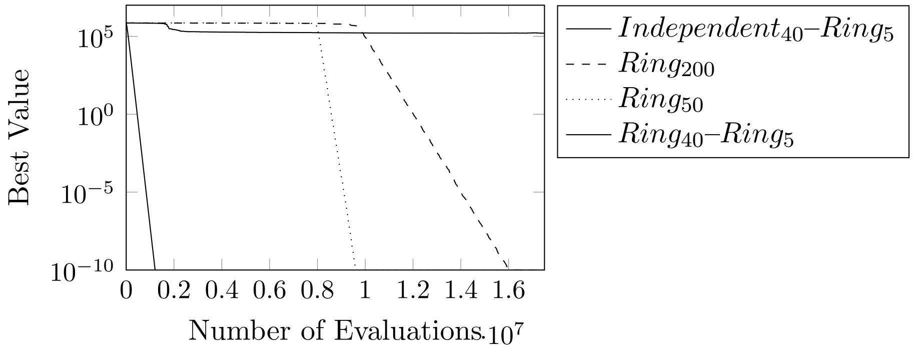


Figure 6.2: Convergence plot for Sphere in serial PSO, comparing an apiary (using 100 subiterations) with a swarm of the same number of total particles (200) and a swarm of 50 particles.

respectively. The $Ring_{40}$–$Ring_{5}$ apiaries require the same number of evaluations per iteration as the $Ring_{200}$ swarms, but they perform far better than the individual *Ring* swarms. Note that the $Ring_{200}$ swarm in Figure 6.2 converges more slowly than the $Ring_{50}$ swarm because it requires more evaluations per iteration.

One might wonder whether the performance of the Apiary topology are dependent on the social interactions or whether they are merely due to the repetition of a high-variance experiment. After all, running 40 independent swarms of 5 particles would be expected to perform better than a single swarm of 5 particles. Figure 6.2 includes the abysmal results of such an $Independent_{40}$–$Ring_{5}$ topology, thus dispelling this possibility. The difference between the independent swarms and the apiary demonstrates emergent behavior.

Table 6.1: Median number of function evaluations to reach a value of 10^{-10}. The best cell in each row is bolded if statistically significant.

Function	$Ring_{50}$	$Ring_{200}$	$Ring_{40}$–$Ring_5$
Sphere-250	7.8×10^5	3×10^6	**5.9×10^5**
$(10^{th}, 90^{th})$	$(7.6\times10^5, 8.1\times10^5)$	$(3\times10^6, 3.1\times10^6)$	**$(5.9\times10^5, 6\times10^5)$**
Sphere-500	9.5×10^6	1.6×10^7	**1.2×10^6**
$(10^{th}, 90^{th})$	$(3.5\times10^6, 2.6\times10^7)$	$(1\times10^7, 4\times10^7)$	**$(1.2\times10^6, 1.2\times10^6)$**

Table 6.2: Median best value at a fixed number of function evaluations.

Function	$Ring_{50}$	$Ring_{200}$	$Ring_{40}$–$Ring_5$
Ackley-250	20	20	**20**
$(10^{th}, 90^{th})$	(20, 20)	(20, 20)	**(20, 20)**
Ackley-500	20	20	**20**
$(10^{th}, 90^{th})$	(20, 21)	(20, 21)	**(20, 20)**
Rastrigin-250	2.6×10^3	2.3×10^3	**1.9×10^3**
$(10^{th}, 90^{th})$	$(2.1\times10^3, 2.9\times10^3)$	$(2\times10^3, 2.5\times10^3)$	**$(1.7\times10^3, 2.1\times10^3)$**
Rastrigin-500	1.2×10^4	1.2×10^4	**4.1×10^3**
$(10^{th}, 90^{th})$	$(1.1\times10^4, 1.3\times10^4)$	$(8.5\times10^3, 1.2\times10^4)$	**$(3.9\times10^3, 4.5\times10^3)$**
Rosenbrock-250	70	3.7×10^2	**0.0012**
$(10^{th}, 90^{th})$	$(0.029, 3.4\times10^2)$	$(2.8\times10^2, 4.5\times10^2)$	**$(6.1\times10^{-9}, 4)$**
Rosenbrock-500	4.3×10^{12}	4.1×10^{12}	**8.8×10^2**
$(10^{th}, 90^{th})$	$(4\times10^{12}, 4.6\times10^{12})$	$(3.8\times10^{12}, 4.3\times10^{12})$	**$(7\times10^2, 1.1\times10^3)$**
Schwefel1.2-250	7.4×10^4	2.3×10^5	**1.6×10^4**
$(10^{th}, 90^{th})$	$(4.6\times10^4, 1.5\times10^5)$	$(2.1\times10^5, 2.7\times10^5)$	**$(1.2\times10^4, 2.1\times10^4)$**
Schwefel1.2-500	1.6×10^6	2.2×10^6	**8.1×10^5**
$(10^{th}, 90^{th})$	$(1.1\times10^6, 2.3\times10^6)$	$(1.8\times10^6, 2.8\times10^6)$	**$(7.1\times10^5, 9.4\times10^5)$**

We include results for the full range of benchmark functions in tabular form. In the case of Sphere, all runs of all PSO variants eventually converge to the global minimum. Table 6.1 reports the number of function evaluations to convergence. Table 6.2 reports the best value obtained at a fixed number of function evaluations for the other benchmark functions. The fixed number of evaluations for each function are equivalent to about 6 hours of computation, specifically: 6×10^6 for Ackley-250 and Ackley-500, 1×10^7 for Rastrigin-250, 3.5×10^6 for Rastrigin-500, 1×10^7 for Rosenbrock-250, 5×10^6 for Rosenbrock-500, 6×10^6 for Schwefel1.2-250, and 2×10^6 for Schwefel1.2-500. In all cases the apiary is best with statistical significance. Though the results for the Ackley function are statistically significant, the difference is small.

Table 6.3: Median best value at a fixed number of function evaluations. The topology is $Ring_{40}$–$Ring_{25}$ for Rastrigin and $Ring_{40}$–$Ring_{5}$ for Rosenbrock and Schwefel 1.2.

Function	Apiary	DMS-PSO
Rastrigin-250 (10^{th}, 90^{th})	1.5×10^{3} (1.4×10^{3}, 1.6×10^{3})	$\mathbf{4.8\times10^{2}}$ $\mathbf{(4\times10^{2},\ 5.9\times10^{2})}$
Rastrigin-500 (10^{th}, 90^{th})	3.3×10^{3} (3×10^{3}, 3.6×10^{3})	$\mathbf{1.3\times10^{3}}$ $\mathbf{(1.2\times10^{3},\ 1.6\times10^{3})}$
Rosenbrock-250 (10^{th}, 90^{th})	$\mathbf{0.0012}$ $\mathbf{(6.1\times10^{-9},\ 4)}$	1.3×10^{2} (1.4, 2.3×10^{2})
Rosenbrock-500 (10^{th}, 90^{th})	8.8×10^{2} (7×10^{2}, 1.1×10^{3})	8.3×10^{2} (6.4×10^{2}, 9.8×10^{2})
Schwefel1.2-250 (10^{th}, 90^{th})	$\mathbf{1.6\times10^{4}}$ $\mathbf{(1.2\times10^{4},\ 2.1\times10^{4})}$	7×10^{4} (5.1×10^{4}, 8.9×10^{4})
Schwefel1.2-500 (10^{th}, 90^{th})	$\mathbf{8.1\times10^{5}}$ $\mathbf{(7.1\times10^{5},\ 9.4\times10^{5})}$	1.4×10^{6} (1.2×10^{6}, 1.7×10^{6})

For some functions, the Apiary topology outperforms Dynamic Multi-Swarm PSO [19] (DMS-PSO) in serial, while for other functions, DMS-PSO outperforms the Apiary topology. For Sphere-500, the Apiary topology finds the minimum faster with a small but statistically significant advantage, while for Sphere-250, the situation is reversed (the full table is omitted due to space constraints). Table 6.3 shows similarly mixed results for the other benchmark functions.

6.4.2 Apiary Parameters

We now justify the basic apiary parameters of a $Ring_{40}$ outer topology, a $Ring_{5}$ inner topology, and 100 subiterations. General recommendations set $Ring_{50}$ as a standard swarm topology [39] or even higher for difficult problems [63]. Previous subswarm topologies have suggested that 50–100 particles per subswarm [20] or 32 particles per subswarm [25] give ideal performance. In contrast, we recommend starting with small subswarms of about 5 particles.

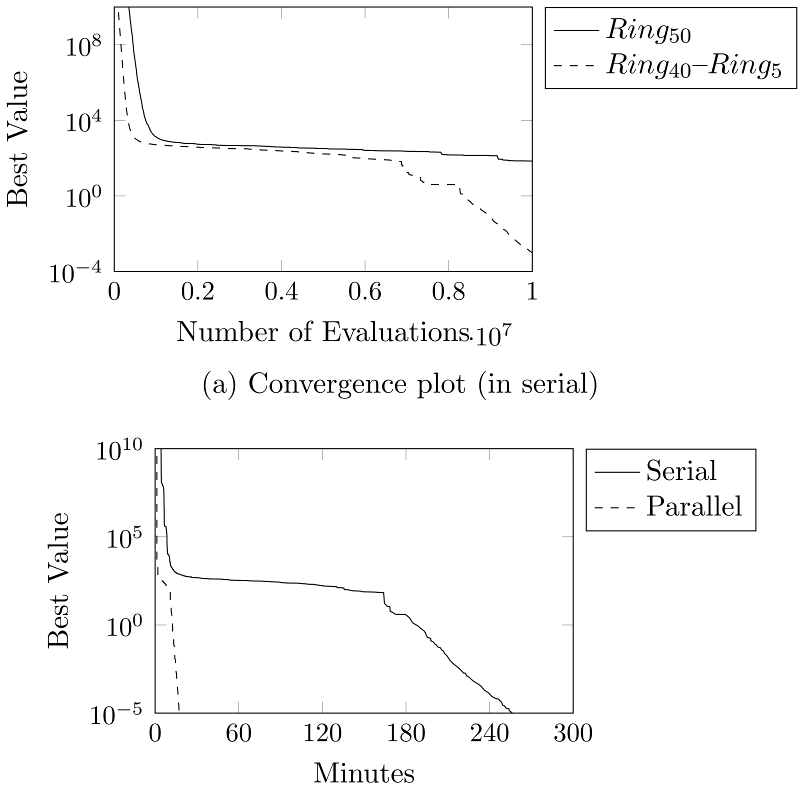
Increasing the number of particles per subswarm or the total number of subswarms provide improvements only in some circumstances. Table 6.4 compares $Ring_{40}$–$Ring_{5}$ to $Ring_{200}$–$Ring_{5}$, an apiary with 5 times as many subswarms, and to $Ring_{40}$–$Ring_{25}$, an

Table 6.4: Median best value at n function evaluations.

Function	$Ring_{40}$–$Ring_{5}$	$Ring_{200}$–$Ring_{5}$	$Ring_{40}$–$Ring_{25}$
Rastrigin-250	1.9×10^{3}	1.7×10^{3}	**1.5×10^{3}**
(10^{th}, 90^{th})	(1.7×10^{3}, 2.1×10^{3})	(1.6×10^{3}, 1.9×10^{3})	**(1.4×10^{3}, 1.6×10^{3})**
Rastrigin-500	4.1×10^{3}	3.8×10^{3}	**3.3×10^{3}**
(10^{th}, 90^{th})	(3.9×10^{3}, 4.5×10^{3})	(3.6×10^{3}, 4×10^{3})	**(3×10^{3}, 3.6×10^{3})**
Rosenbrock-250	**0.0012**	2.6×10^{2}	0.27
(10^{th}, 90^{th})	**(6.1×10^{-9}, 4)**	(2×10^{2}, 3.2×10^{2})	(0.0016, 76)
Rosenbrock-500	**8.8×10^{2}**	5×10^{3}	9.4×10^{2}
(10^{th}, 90^{th})	**(7×10^{2}, 1.1×10^{3})**	(3.4×10^{3}, 1.4×10^{4})	(8.2×10^{2}, 1.2×10^{3})
Schwefel1.2-250	**1.6×10^{4}**	1.5×10^{5}	3.7×10^{4}
(10^{th}, 90^{th})	**(1.2×10^{4}, 2.1×10^{4})**	(1.3×10^{5}, 1.6×10^{5})	(2.8×10^{4}, 4.8×10^{4})
Schwefel1.2-500	**8.1×10^{5}**	1.6×10^{6}	1×10^{6}
(10^{th}, 90^{th})	**(7.1×10^{5}, 9.4×10^{5})**	(1.5×10^{6}, 1.8×10^{6})	(8.8×10^{5}, 1.2×10^{6})

apiary with 5 times as many particles per subswarm. For most of the benchmark functions, the $Ring_{40}$–$Ring_{5}$ apiary performs significantly better than either of the larger topologies. Likewise, the $Ring_{40}$–$Ring_{5}$ topology significantly outperforms the others for Sphere-250 and Sphere-500 (the table is omitted due to space). For such functions, the increased number of evaluations per iteration offsets any increased exploration provided by the larger swarms. On the other hand, both of the larger topologies are better for Rastrigin-250, Rastrigin-500, and Rosenbrock-500. As the number of local minima in Rastrigin increases exponentially with the number of dimensions, we conclude that larger swarms are preferable for highly multimodal objective functions.

Changing the communication between swarms can affect performance dramatically. Using a more connected outer topology, such as *Complete*, gives poor performance in serial PSO in addition to requiring more communication in parallel PSO. Sharing the best value of an arbitrary member of each subswarm instead of the best particle of each subswarm also reduces performance. Setting the number of subiterations to 100 is high enough to provide reasonable task granularity even for the least expensive benchmark functions in parallel PSO.



(a) Convergence plot (in serial)

(b) Apiary in serial and parallel

Figure 6.3: Convergence plots of the Apiary topology for the Rosenbrock function with respect to function evaluations and time.

6.4.3 Parallel Performance of Apiaries

Benchmark functions are extremely inexpensive, yet despite the high relative cost of communication, the Apiary topology performs extremely well in parallel. Figure 6.3 shows the results for the Rosenbrock function with both serial and parallel computation. Performing 100 iterations on 5 particles requires only 0.2 seconds, and parallel PSO took about 0.5 seconds per iteration. With any realistically expensive function, the overhead of 0.3 seconds would be negligible.

In a parallel context with a large number of spare processors, there may be limited additional overhead in increasing the number of subswarms. In this light, we revisit the conclusions from Section 6.4.2. In this context, the number of iterations of PSO is a more appropriate measure than the number of function evaluations [63]. With respect to iterations,

Table 6.5: Median best value at n iterations.

Function	$Ring_{40}$–$Ring_5$	$Ring_{200}$–$Ring_5$	$Ring_{40}$–$Ring_{25}$
Rastrigin-250	1.9×10^3	1.7×10^3	$\mathbf{1.5\times10^3}$
(10^{th}, 90^{th})	(1.7×10^3, 2.1×10^3)	(1.6×10^3, 1.9×10^3)	$\mathbf{(1.4\times10^3,\ 1.6\times10^3)}$
Rastrigin-500	4.2×10^3	3.8×10^3	$\mathbf{3.4\times10^3}$
(10^{th}, 90^{th})	(3.9×10^3, 4.5×10^3)	(3.6×10^3, 4×10^3)	$\mathbf{(3\times10^3,\ 3.6\times10^3)}$
Rosenbrock-250	3.6×10^2	$\mathbf{2.5\times10^2}$	2.9×10^2
(10^{th}, 90^{th})	(2.6×10^2, 4.4×10^2)	$\mathbf{(1.9\times10^2,\ 3.2\times10^2)}$	(2.1×10^2, 3.5×10^2)
Rosenbrock-500	2×10^4	$\mathbf{4.2\times10^3}$	1.5×10^4
(10^{th}, 90^{th})	(4.5×10^3, 3.1×10^7)	$\mathbf{(3\times10^3,\ 9.4\times10^3)}$	(3.5×10^3, 7.5×10^6)
Schwefel1.2-250	1.7×10^5	$\mathbf{1.4\times10^5}$	1.7×10^5
(10^{th}, 90^{th})	(1.4×10^5, 2×10^5)	$\mathbf{(1.3\times10^5,\ 1.6\times10^5)}$	(1.3×10^5, 2×10^5)
Schwefel1.2-500	1.8×10^6	$\mathbf{1.6\times10^6}$	1.8×10^6
(10^{th}, 90^{th})	(1.6×10^6, 2.2×10^6)	$\mathbf{(1.5\times10^6,\ 1.8\times10^6)}$	(1.5×10^6, 2.1×10^6)

Table 6.4 compares $Ring_{40}$–$Ring_5$ with $Ring_{200}$–$Ring_5$ and $Ring_{40}$–$Ring_{25}$, which loosely represent the situations where additional processors or time are available, respectively. If extra resources are available, they clearly provide improvements in the pursuit of better answers.

6.5 Conclusions and Future Work

Organizing particle swarms into communities of subswarms significantly improves the performance of PSO. We attribute the improvement to emergent behavior from the social interaction of particles. We speculate that small groups of particles might make progress on implicit subproblems. Likewise, subswarms might help other subswarms get unstuck if they have prematurely converged in individual dimensions. In any case, the behavior of particle swarm apiaries is not explained by amount of communication, but rather the structure of the swarms.

Furthermore, we have shown that apiaries are particularly well-suited to parallel computation. With low communication and adjustable task granularity, the topology is easily adapted to varying computational architectures. With an inexpensive benchmark function, parallel PSO was able to perform about 2 outer iterations per second and provide a speedup of 15 on 40 processors. For any non-trivial function, the performance would be even more pronounced. Unlike other multi-swarm topologies like DMS-PSO [19], which requires frequent global communication, the Apiary topology requires very little communication.

We believe there are several interesting areas that are open to future work. In particular, organizing subswarms into hierarchies is a promising possibility. Apiaries are effective with extremely small subswarms, so a hierarchical structure can be built with a low branching factor. For example, a three-layer apiary would only have $5^3 = 125$ particles, and a four-layer apiary would have $5^4 = 625$ particles, well within the range that can be computed on a medium-size cluster.

Acknowledgments

The Fulton Supercomputing Lab at Brigham Young University generously provided 10 processor-years of resources.

Chapter 7

Serial PSO Results Are Irrelevant in a Multi-core Parallel World

Published in Proceedings of CEC 2014 [71]

This chapter builds on previous chapters to identify the fundamental issues that arise in parallel optimization and to consider how the appropriate use of resources depends on the attributes of the objective function. On the one hand, processors are idle if they are waiting for communication, but on the other hand, they may work in vain if they lack up to date information from other processors.

Abstract

From multi-core processors to parallel GPUs to computing clusters, computing resources are increasingly parallel. These parallel resources are being used to address increasingly challenging applications. This presents an opportunity to design optimization algorithms that use parallel processors efficiently. In spite of the intuitively parallel nature of Particle Swarm Optimization (PSO), most PSO variants are not evaluated from a parallel perspective and introduce extra communication and bottlenecks that are inefficient in a parallel environment.

We argue that the standard practice of evaluating a PSO variant by reporting function values with respect to the number of function evaluations is inadequate for evaluating PSO in a parallel environment. Evaluating the parallel performance of a PSO variant instead requires reporting function values with respect to the number of iterations to show how the algorithm scales with the number of processors, along with an implementation-independent description of task interactions and communication. Furthermore, it is important to acknowledge the

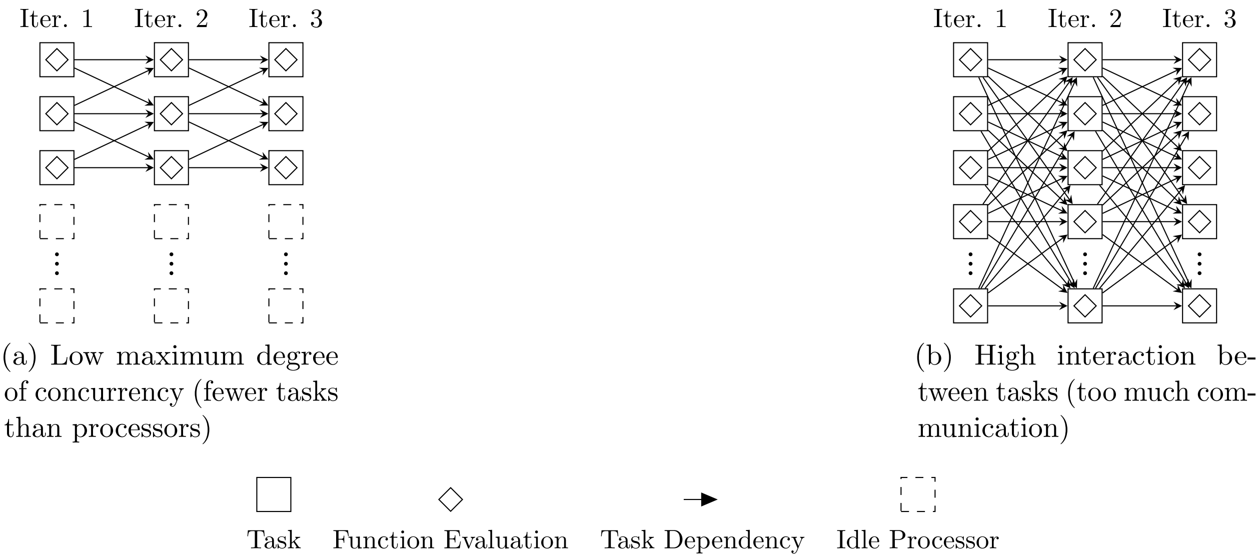
dependence of performance on specific properties of the objective function and computational resources. We discuss parallel evaluation of PSO, and we review approaches for increasing concurrency and for reducing communication which should be considered when discussing the scalability of a PSO variant. This discussion is essential both for designers who are defending the performance of an algorithm and for practitioners who are determining how to apply PSO for a given objective function and parallel environment.

7.1 Introduction

Despite the overwhelming parallel nature of modern hardware, contributions to Particle Swarm Optimization (PSO) are still evaluated from a purely serial perspective. Variants to PSO may improve performance in a serial environment but worsen performance in a parallel environment. For example, adaptive topologies ensure population diversity, but adaptation usually requires global information about the swarm and either reduces concurrency or increases communication costs. Researchers must understand these issues to evaluate PSO parameters and variants, and practitioners must understand them to apply PSO effectively.

For the case of serial PSO, Bratton and Kennedy [39] addressed common issues such as swarm size and motion equations in order to establish a common starting point for PSO, but these conclusions do not directly apply to parallel PSO. While practical approaches to parallel PSO are available, most PSO research does not consider issues raised by parallel computation, such as communication and scaling the number of processors. We are not aware of any general attempts to address the implementation-independent consequences of parallelization. Now that computing clusters, multicore processors, and powerful GPUs are commonplace, the behavior of parallel PSO must be a primary rather than a secondary concern.

We organize the issues raised by PSO in a parallel environment into two categories, processor scaling and task interaction, which both limit the degree of concurrency, the number of tasks that can be computed simultaneously [32]. First, a PSO variant may scale poorly



(a) Low maximum degree of concurrency (fewer tasks than processors)

(b) High interaction between tasks (too much communication)

(c) Legend

Figure 7.1: Task dependencies for parallel PSO in two simple inefficient cases.

as the number of processors increases, independent of communication. For example, if the maximum degree of concurrency is less than the total number of processors, then the average degree of concurrency is limited because the processors are never all active. Different ways to increase the number of concurrent tasks, such as increasing the swarm size, are not necessarily equally effective. Second, interaction between tasks limits the average degree of concurrency because tasks remain idle while waiting for communication. Additionally, communication can affect performance if non-local information, if available, would make the work more effective. Issues such as motion equations and topology determine the dependencies and communication between tasks. Figure 7.1 illustrates how maximum degree of concurrency and task interaction affect performance. In Figure 7.1a, the number of tasks is smaller than the number of processors, so several processors are unused. In Figure 7.1b, every task in one iteration interacts with every task in the next iteration, requiring a high level of communication. In a task dependency diagram, the maximum degree of concurrency is given by the number of tasks in a column, and the task interactions are given by the arrows.

Evaluation of any PSO variant must include discussion of the issues of processor scaling and task interaction from an implementation-independent parallel perspective. For serial

PSO, the implementation-independent performance of a PSO variant is usually measured with respect to the number of function evaluations. For parallel PSO, measuring the performance relative to the number of iterations gives the per-processor performance, which shows how well the PSO variant can exploit all available processors. Analysis of task interaction and communication of a variant gives an implementation-independent understanding of the overhead it introduces. Together with the per-evaluation cost of the objective function and the characteristics of the implementation and parallel hardware, the per-processor scaling and task interactions reveal how the variant will perform in a particular computational environment.

The paper proceeds as follows. Section 7.2 introduces parallel Particles Swarm Optimization by describing PSO, considering the relationship between its performance and the objective function, and reviewing approaches for the parallelization of PSO. The next two sections discuss the issues that limit the efficiency of parallel PSO. Section 7.3 considers scaling of processors independent of communication and ways to improve it. Section 7.4 considers the task interactions and communication introduced by a PSO variant in parallel and ways to reduce bottlenecks and overhead. Finally, Section 7.5 concludes with a plea to consider this parallel perspective when evaluating all variants of PSO.

7.2 Parallel PSO

This section gives an overview of Particle Swarm Optimization and its parallelization. Section 7.2.1 reviews PSO and the standard Constricted PSO motion equations. Section 7.2.2 discusses the properties of objective functions that are relevant to optimization and the role of benchmark functions in evaluating PSO. It also describes the benchmark functions used in this paper. Section 7.2.3 discusses the decomposition of PSO into tasks in parallel PSO.

7.2.1 Particle Swarm Optimization

Particle Swarm Optimization simulates the motion of particles in the domain of an objective function. These particles search for the global 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 standard variant [39]. Each particle's position $\mathbf{x}_0$ and velocity $\mathbf{v}_0$ 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 $\mathbf{x}_t$ and velocity $\mathbf{v}_t$ with respect to the personal best $\mathbf{x}^P_{t-1}$ and neighborhood best $\mathbf{x}^N_{t-1}$ from the preceding iteration:

$$\begin{aligned} v_{t,i} = \chi\big[v_{t-1,i} + \phi^P u^P_{t-1,i}(x^P_{t-1,i} - x_{t-1,i}) +& \\ \phi^N u^N_{t-1,i}(x^N_{t-1,i} - x_{t-1,i})\big]& \qquad (7.1) \\ x_{t,i} = x_{t-1,i} + v_{t,i}& \qquad (7.2) \end{aligned}$$

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 $\phi = \phi^P + \phi^N$ [24].

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 [64]. Topologies also determine the amount of communication between particles, which is especially important for parallel implementations of PSO.

7.2.2 Objective Functions

The objective function has properties which determine the behavior of PSO. Functions may be expensive or inexpensive in terms of the time per function evaluation, they may be unimodal or highly multimodal, and they have some number of input dimensions. Some properties of objective functions are easy to determine, while others are more elusive. In any case, these properties have a great effect on the performance of PSO and must be considered when tuning and running the algorithm. The usual ways of adapting PSO include motion equations, swarm topology, and swarm size. Parallelization also provides a number of techniques that must be considered for parallel PSO. The effect of these parameters for particular objective functions can only be identified using empirical experimentation.

The ideal motion equations, topology, and swarm size for PSO depend on the objective function. Under various benchmark functions, 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 [21].

Benchmark functions are intended to share interesting properties with real-life functions while being inexpensive to facilitate experimentation. We occasionally refer to a few well-known benchmark functions. The *Sphere* function or parabola is $f_S(\mathbf{x}) = \sum_{i=1}^{D} x_i^2$. Particles are initialized in the interval $[-50, 50]^D$. The *Griewank* function is $f_G(\mathbf{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{D} x_i^2 - \Pi_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right)$. We use the 15-dimensional variant with the feasible region $[-600, 600]^{15}$ (Griewank is more challenging with fewer dimensions than with more dimensions). The *Rastrigin* function is $f_R(\mathbf{x}) = \sum_{i=1}^{D} \left(x_i^2 - 10\cos\left(2\pi x_i\right) + 10\right)$. We use the 50-dimensional variant with the feasible region $[-5.12, 5.12]^{50}$.

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 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.

Unlike in serial PSO, the time for each evaluation of the objective function is an important consideration in parallel PSO. The evaluation time does not generally affect the behavior of PSO, but an expensive function evaluation decreases the relative cost of communication. In general, PSO is more sensitive to details of parallelization for inexpensive objective functions than for expensive functions because the time spent in communication is more likely to outweigh the time spent in function evaluations.

7.2.3 Parallelization of PSO

In order for an algorithm to be parallelized, its operations must be decomposed into tasks [32]. The most fine-grained decompositions that are generally possible with PSO correspond to a task for each evaluation of the objective function. In specific cases, it may be possible to decompose the objective function itself, but we limit the discussion and the term "parallel PSO" to approaches that work for arbitrary objective functions. Even in the case of a parallelized objective function, it may be beneficial to also use parallel PSO.

Particle Swarm Optimization is usually decomposed into a task for each function evaluation, with one task per particle per iteration [25, 28, 72]. However, this is not always the case, and parallel PSO may take several different forms depending on what work constitutes each task, which mapping technique is used, and how the tasks interact. Some of these choices may significantly affect the behavior of PSO, while others are implementation details that may affect parallel performance and scalability but do not require rethinking at the PSO level. For example, tasks may include the work of a single particle or multiple particles; combining several particles into a single task can reduce communication given an appropriate choice of swarm topology. Likewise, tasks may include only the function evaluation, with each particle's position, velocity, and neighborhood best updated in serial on a centralized master [25], or tasks may include the full particle update, with position, velocity, and neighborhood best updates performed in parallel [28].

The performance of parallel PSO depends on the parallel computing environment, including hardware. Networked clusters, multi-core computers, and graphics processing units (GPUs) are all common parallel processing platforms, but they have very different characteristics. Cluster are the standard way to scale to a large number of general-purpose processors, but communication over a network is slow. Multi-core computers have much lower communication costs but have a limited total processing power. A GPU is very effective for massively parallel computation but faces restrictions on the operations that can be performed. To a large degree, external considerations force a particular choice of computational platform.

For extremely inexpensive objective functions, graphics processing units (GPUs) are an attractive platform for parallelization, but they are not applicable in all situations. GPUs are extremely fast at performing floating point operations, and GPU-based implementations of PSO can improve performance by an order of magnitude compared to a single processor [73]. GPUs perform well for objective functions that are floating point heavy, but they are less effective for functions relying on integer operations or large amounts of data. Implementations of PSO that use texture mapping on GPUs additionally require independence between variables [74]. General-purpose parallel architectures, such as CUDA and OpenCL, offer a more flexible approach to GPU-based parallel PSO [73].

In parallel PSO, tasks can be distributed to processors with either dynamic mapping or static mapping techniques. With dynamic mapping, tasks are distributed at runtime, often with a centralized scheduler. With static mapping, each particle is fixed to a specific processor, which performs all updates to its state. Fully distributed implementations may even use peer-to-peer networks for communication between particles [75]. Assuming that position, velocity, and neighborhood best updates all occur in serial on a centralized processor gives a poor understanding of behavior for a more distributed implementation. In general, it is safe to make a simplifying assumption that parallel PSO is fully distributed.

7.3 Processor Scaling Independent of Communication

The ideal case where communication is free separates the issue of how an algorithm takes advantage of processors from the issue of task interaction. As the number of processors increases, the maximum degree of concurrency—the maximum number of tasks that can be computed simultaneously—must increase accordingly, or the additional processors are never utilized. In a task-dependency graph such as noted earlier in Figure 7.1, the maximum degree of concurrency is represented by the number of rows. While it is important to be able to increase the maximum degree of concurrency as the number of processors increases, it is also important to do so in the most effective way possible. A significant challenge for parallel PSO is to have as high of a marginal improvement in performance as possible as the number of processors increases.

Evaluating PSO algorithms as the number of processors increases or comparing PSO variants at a fixed number of processors is not possible from the traditional serial perspective. PSO algorithms are usually evaluated on their performance with respect to the number of function evaluations, but this is not appropriate for parallel PSO, where function evaluations are performed concurrently. Furthermore, criteria such as speedup or wallclock time per iteration are only appropriate for evaluating a specific implementation of parallel PSO because these measures are highly implementation-dependent. In a parallel context, the number of function evaluations divided by the number of processors is a much better implementation-independent scale with which to measure performance. If the number of particles is equal to the number of processors, this is simply the number of iterations, and in other cases, it is closely related. Comparing the performance of PSO with respect to the number of iterations as the number of processors increases shows the scaling behavior, which may in turn depend on the specific strategy for employing additional processors.

There are multiple ways to employ additional processors, and their effectiveness depends on the objective function and computational resources. For example, Figure 7.2 shows the success rate of three different topologies as the number of processors increases. For

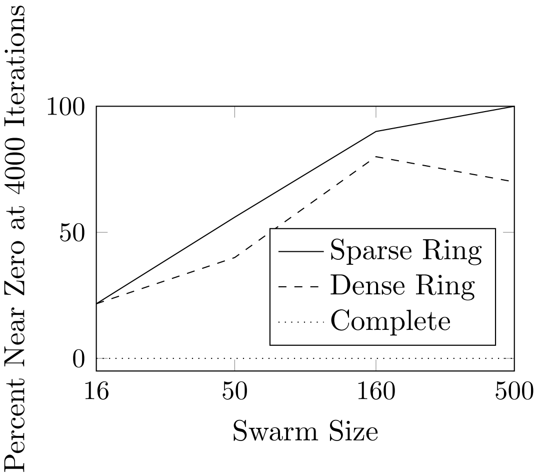


Figure 7.2: Percent of swarms attaining a value near the global optimum of the Griewank function at a fixed number of iterations. In the sparse ring topologies, each particle is connected to 2 neighbors; in the dense ring topologies, each particle is connected to 20% of the swarm; and in the complete topologies, each particle is connected to the entire swarm.

this particular objective function, the more densely connected swarms scale poorly with the number of processors compared to the more sparsely connected swarms. In the remainder of this section, we consider three approaches for increasing the maximum degree of concurrency of parallel PSO: independent runs, increased swarm size, and speculative evaluation.

7.3.1 Independent Runs

The naïve—and perhaps most common—way to increase the maximum degree of concurrency of PSO is to perform independent runs on different processors and take the best result after the runs have completed. A common use case for independent runs is to measure variability of PSO for a given objective function with a particular combination of parameters. While this may provide some insight into the nature of the objective function, it is especially useful for evaluating a variant of PSO. In fact, when evaluating parallel PSO relative to the number of iterations, it is far more efficient to perform repeated experiments as parallel independent sequential runs of PSO than by running a series of experiments on a parallel implementation.

Parallel independent runs of PSO will give better results than a single run of PSO with the same parameters if there is any variability whatsoever between runs, but this approach is generally less effective than more sophisticated approaches. Running n independent runs of PSO, each with k particles, is equivalent to a single parallel run of PSO with kn particles partitioned into n disjoint sets of k particles. Since there is no communication between the subswarms of particles, independent runs cannot share any information that might help improve the search.

7.3.2 Swarm Size

Increasing the swarm size is generally a much more effective use of parallel resources than simply running multiple serial copies of PSO. Unfortunately, many PSO variants have not been tested at multiple swarm sizes to determine how well they scale. Serial PSO is typically used with small swarms of about 50 particles, with slightly larger or smaller swarm sizes as indicated by customized testing for the specific objective function [39]. We review the general effects of increasing the swarm size and encourage testing the effects of swarm size in all evaluation of PSO variants.

All else equal, an increase in swarm size increases the exploration of PSO and decreases its variability between runs. If a particularly multimodal objective function requires more particles for adequate exploration than the number of available processors, then this simultaneously increases the maximum degree of concurrency. Small swarms converge prematurely for the Griewank and Rastrigin functions. In parallel PSO, where performance is plotted against iterations instead of function evaluations, the benefit of large swarms is even more pronounced. Figure 7.3 and Figure 7.4 show that for Griewank and Rastrigin respectively, large swarms are not only less prone to premature convergence, but in parallel PSO they also attain comparable values in fewer iterations.

Efficiently utilizing the processors is more challenging for functions that do not require as much exploration, but even in an extreme case of a unimodal function, increasing the swarm size up to the number of processors is always beneficial (aside from communication issues discussed in Section 7.4). Although the ideal swarm size for the Sphere function in

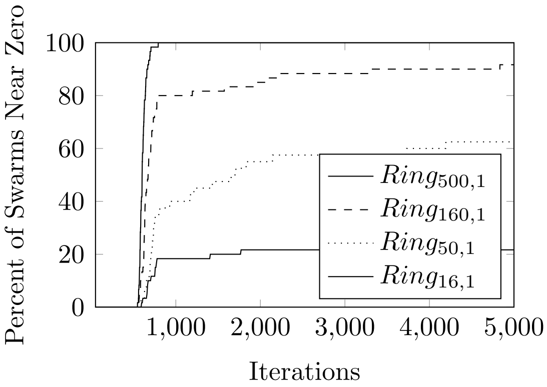


Figure 7.3: Success rate on Griewank with respect to iterations for $Ring_{n,1}$ with various swarm sizes.

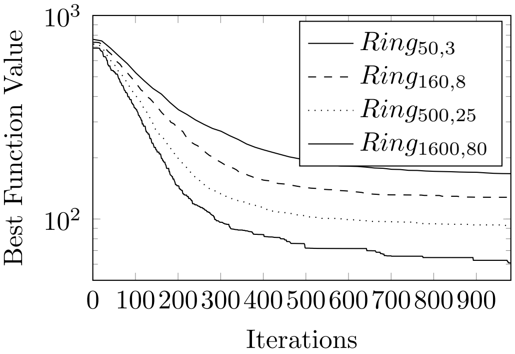


Figure 7.4: PSO performance on Rastrigin with respect to iterations with various topologies.

serial PSO is about 30 particles, Figure 7.5 shows that a greater swarm size always gives improvements in parallel PSO if additional processors are available. However, the marginal improvement diminishes as the swarm size increases (note the log scale), so it is important to find PSO variants that scale as well as possible. For example, organizing a large swarm of particles into subswarms is generally more effective than organizing the swarm into a large ring [40].

7.3.3 Speculative Evaluation

It seems intuitively obvious that if the number of particles is less than the number of processors, then some of those processors would be unused at each iteration. However, the work associated with a particle can be split into multiple tasks by reorganizing the work

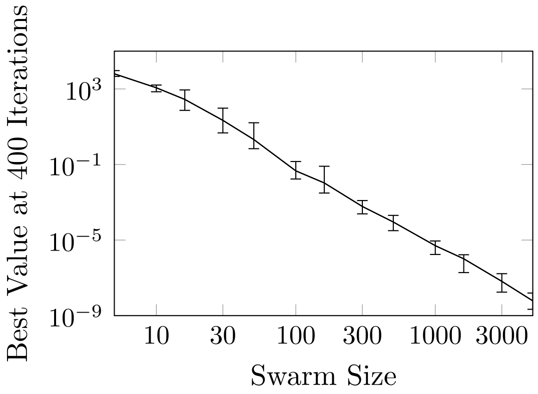


Figure 7.5: Performance of PSO for the Sphere function with a complete topology at various swarm sizes. If additional processors are available, increasing the swarm size is always beneficial, even in the extreme case of a unimodal function.

performed in consecutive iterations of PSO. In this manner, speculative evaluation allows PSO to perform two iterations concurrently [62]. Figure 7.6 depicts the task dependencies of PSO with speculative evaluation. At the cost of using multiple processors for each particle, and thus requiring the number of particles to be fewer than the number of processors, PSO with speculative evaluation can reach values much more quickly for appropriate objective functions. Where premature convergence is a concern, increasing swarm size may be a more valuable use of resources, but if exploration is adequate, speculative evaluation can halve the runtime of parallel PSO.

7.4 Task Interaction and Communication

Task interactions limit the performance of parallel PSO by causing processors to be idle while sending or receiving communication or to use outdated information. The effect of task interaction depends on the objective function and computational environment. For example, given a function with extremely expensive function evaluations, the amount of communication may be negligible relative to the time spent on function evaluation, but waiting for one straggling processor to complete the current iteration may leave a large number of processors idle for a long time. Evaluating a PSO variant requires analysis both of the amount of communication and the interactions that require idle waiting.

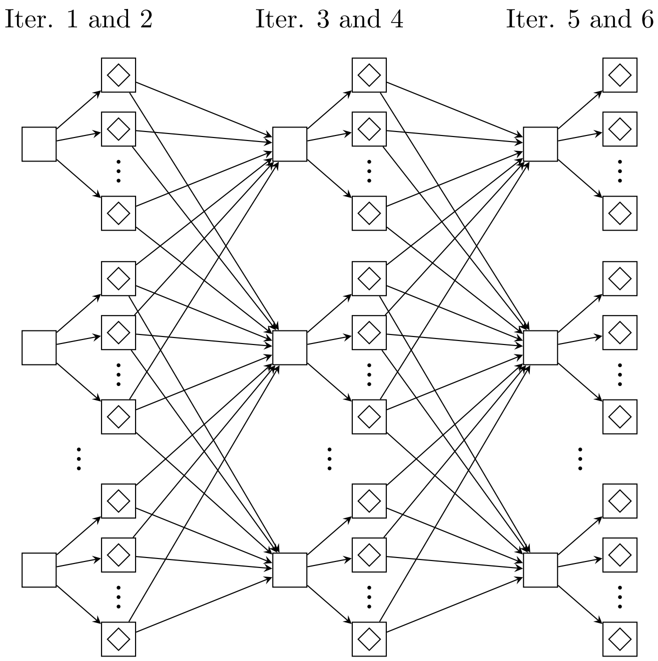


Figure 7.6: Task dependency diagram for parallel PSO with speculative evaluation.

Appropriate choice of topologies, motion equations, and other techniques can minimize the interactions and dependencies between tasks. We consider three approaches to reducing task interactions that apply in many situations: sparse topologies, subswarms, and asynchronous parallel PSO.

7.4.1 Sparse Topologies

The swarm topology determines which particles communicate at each iteration. Sparse topologies require less communication than dense topologies. PSO variants and topologies should be evaluated by the average number and size of messages per iteration that must be sent by each particle. For example, in a swarm of n particles, a ring topology requires 2 messages per particle per iteration, while a complete topology requires $n/2$ messages per particle per iteration. For standard PSO, the cost of messages is dominated by their number rather than size. Motion equations or dynamic topologies which require global information about the swarm introduce task interactions that are equivalent to using a complete topology, as in Figure 7.1b. Variants of PSO that require centralized coordination, or equivalently, communication between every pair of particles at each iteration, are not generally practical for parallel computation.

In some cases where heavy interaction shows improved results for serial PSO, it may be possible to make adaptations to be more appropriate for parallel PSO. For example, for some functions, serial PSO performs better with a fully connected topology than with a sparse topology such as a ring. By changing the motion equations to pass along the best value from any neighbor rather than the best value seen by the particle itself, sparse topologies behave like dense topologies by spreading information quickly through the swarm. In this case, communicating with 2 neighbors chosen randomly at each iteration requires only 2 messages per particle per iteration and gives almost the same performance as the complete topology, which requires a much higher overhead of $n/2$ messages per particle per iteration [63].

7.4.2 Subswarms

If evaluation of the objective function is sufficiently inexpensive relative to the costs of communication, then parallelizing PSO with one particle per processor is pointless. For example, if function evaluation is faster than sending a network packet, then it is cheaper to perform all evaluations locally. Parallelization becomes more favorable, even for inexpensive objective functions, if each processor performs PSO on a semi-independent subswarm.

Most attempts at subswarms for PSO have introduced sophisticated procedures for migrating particles between subswarms as with islands in genetic algorithms [19, 76], but these approaches require centralized coordination that increases communication and idleness. In contrast, the apiary topology [40] achieves similar or better results using the standard mechanism of swarm topology. Thus, communication follows the same structure as ordinary parallel PSO, as shown in the task-dependency graph in Figure 7.7. Note that the number of particles per subswarm and the number of iterations per task determine the task granularity, so it is straightforward to adapt these parameters according to the relative cost of communication. If the topology between subswarms requires k messages per subswarm, and if communication

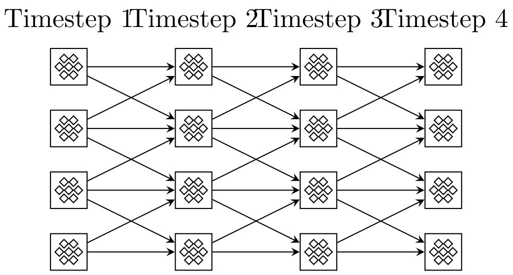


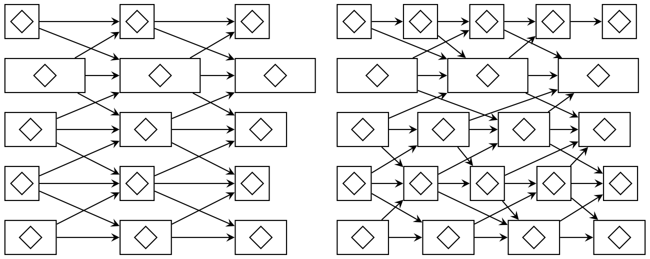
Figure 7.7: Task-dependency graphs for parallel PSO with subswarms The squares represent tasks, and the diamonds represent function evaluations (with multiple particles and iterations in each timestep).

between subswarms occurs every m iterations, then the average number of messages per subswarm per iteration is k/m.

7.4.3 Synchronous and Asynchronous Parallel PSO

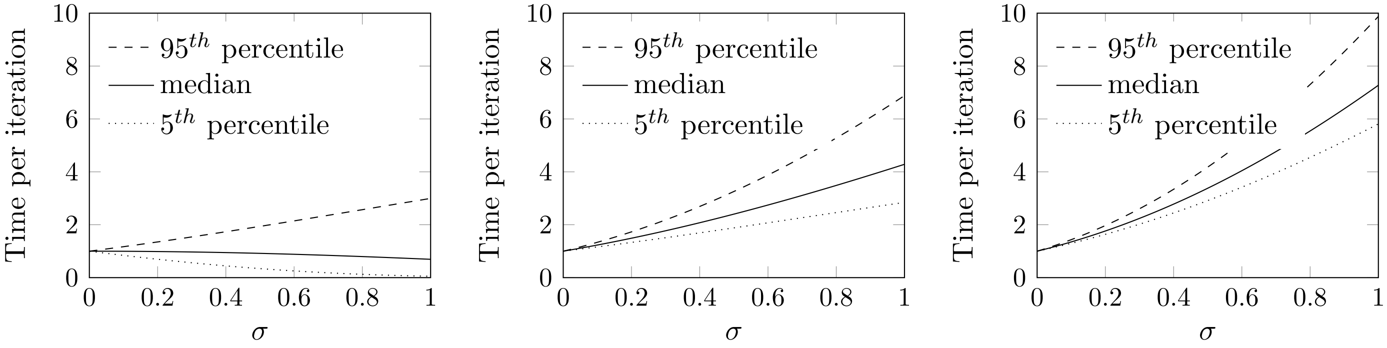
Asynchronous parallel PSO [26, 27] is a modification to the standard algorithm which removes the synchronization point at the end of each iteration. At each iteration of PSO, each particle must update its neighborhood best. This calculation requires the position and the result of the function evaluation from each of its neighbors. Synchronous parallel PSO, which exactly reproduces the computations of serial PSO, requires that computation associated with a particle wait until the results from the previous iteration are available from all of its neighbors. However, in asynchronous parallel PSO, particles iterate independently and communicate asynchronously. If a particle is ready to update its neighborhood best but has not received information about all of its neighbors, it may use information from the previous iteration.[1] Figure 7.8 shows task dependency diagrams for synchronous and asynchronous parallel PSO. Asynchronous iteration is particularly beneficial in situations such as a cluster with heterogeneous processors, an objective function with varying evaluation times, or a cluster with a large number of processors.

[1] Asynchronous parallel PSO has been compared to the "asynchronous updates" variant of serial PSO [26]. 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.



(a) Synchronous parallel PSO (b) Asynchronous parallel PSO

Figure 7.8: Task dependency diagrams for synchronous and asynchronous parallel PSO with heterogeneous processors. In this particular example, asynchronous parallel PSO performs 21 function evaluations in the same time that synchronous parallel PSO performs 15 evaluations.



(a) Asynchronous PSO (single task) (b) Synchronous PSO with 50 processors (c) Synchronous PSO with 1000 processors

Figure 7.9: Probability distributions of the time per iteration (i.e., maximum task time) for synchronous parallel PSO. Individual task times are i.i.d. with a Gamma distribution of mean 1 and varying standard deviations.

There are a few slightly different variants of asynchronous parallel PSO. In a partially asynchronous implementation, particles might wait for some but not all neighbors to complete before proceeding [51]. In some master-slave implementations, particles never get more than one iteration ahead of others [26, 27]. However, in a fully distributed implementation, particles might never wait for information, and one particle could complete many more iterations than another particle [75].

Synchronous and asynchronous parallel PSO are both valuable approaches. The benefits of the synchronous PSO include its simplicity, repeatability, and comparability with standard PSO, which may be essential in research applications. If the evaluation time varies significantly or if processors are heterogeneous, then asynchronous parallel PSO may provide a significant performance improvement over synchronous parallel PSO [26, 27]. However, its slower communication can make asynchronous parallel PSO require more iterations to converge. When evaluation times are consistent and processors are homogeneous, synchronous and asynchronous parallel PSO are comparable with respect to time [51]. Choosing between synchronous and asynchronous parallel PSO is a tradeoff between maximizing the number of function evaluations and having the location of the particles better informed.

If the time for each task is i.i.d. with known distribution, then we can find the distribution of the time per iteration of synchronous parallel PSO. Specifically, let $X_1, X_2, \ldots, X_n$ be i.i.d. random variables with c.d.f. $F(x)$ and p.d.f. $f(x)$ which represent the number of seconds required to perform a function evaluation and communicate the results for each of n concurrent tasks. Then for synchronous parallel PSO, the following iteration can begin after $Y = \max_{1 \leq i \leq n} X_i$ seconds. The distribution of Y is given by the c.d.f. $G(y) = F^n(y)$ and the p.d.f. $g(y) = nF^{n-1}(y)f(y)$. Thus, the median time per iteration is $F^{-1}(2^{-\frac{1}{n}})$ seconds.

This statistical result shows how the cost of synchronous parallel PSO increases with the number of processors. We illustrate this with the case where task times are Gamma distributed with an expected value of 1 second and a standard deviation of σ. Thus, using the inverse-scale parameterization of the Gamma, $X_i \sim \text{Gamma}(\frac{1}{\sigma^2}, \frac{1}{\sigma^2})$. This distribution is shown for varying values of σ in Figure 7.9a. For asynchronous parallel PSO, this represents the time for each task and has an expected value of 1. For synchronous parallel PSO, the time required for each task is effectively lengthened by the need to wait for all other tasks in the same iteration. The distribution over the slowest task time, $\max X_i$, is shown for 50 processors in Figure 7.9b and for 1000 processors in Figure 7.9c with varying values of σ. Similar plots can be made for any distribution with a known c.d.f. and can even be approximated from a set of empirical samples.

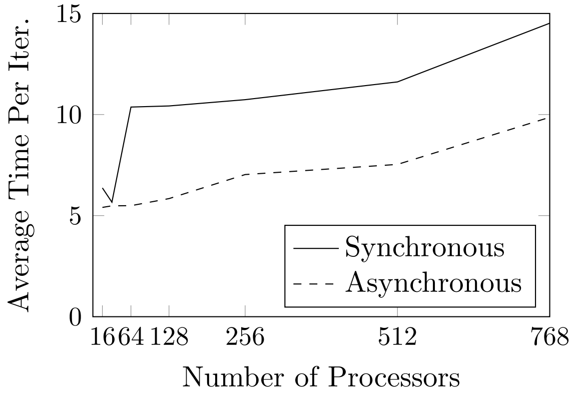


Figure 7.10: Average time per iteration as the number of processors changes. Each function evaluation takes about 5 seconds with very little variance in task times apart from communication.

In practice, there may be variance in task times even if function evaluation times are homogeneous, and task times can be longer for synchronous than for asynchronous parallel PSO because communication is more expensive when all processors are communicating at the same time. Figure 7.10 shows the average time per iteration of synchronous and asynchronous parallel PSO for tasks with very little variance in function evaluation times. At least for this particular implementation of parallel PSO, the benefit of asynchronous parallel PSO is even greater than might be expected.

While there are specific situations where it makes sense to use synchronous parallel PSO, it is important for PSO variants to be compatible with asynchronous iteration. If a variant requires that all particles iterate in lockstep, then it will always be inefficient on large clusters and with objective functions with varying evaluation times.

7.5 Conclusion

We have examined PSO in a parallel context, first by considering how its performance scales independently of communication, and second by considering the task interactions and communication that it requires. Based on this perspective, we have reviewed approaches to improve the parallel performance of PSO. Swarms with topologies based on sparse rings, random neighborhoods, and subswarms provide a variety of flexible options for using communication efficiently.

When comparing PSO variants from a parallel perspective, evaluation must include two important results:

- First, the performance of PSO per iteration at different numbers of processors indicates how well the algorithms use function evaluations as the number of processors scales.

- Second, the number and size of messages per particle per iteration indicates the amount of communication required. Furthermore, any communication beyond that required by the PSO topology, such as centralized coordination, must be identified.

Furthermore, designers should demonstrate that an algorithm does not introduce any centralized bottlenecks or incompatibilities with distributed PSO and asynchronous iteration. All of this information, combined with details about a particular objective function and computational environment, determine the parallel behavior of PSO variants.

In modern computational environments, parallel computation is central to the evaluation of Particle Swarm Optimization. Results that demonstrate an improvement only for serial PSO are insufficient.

Part III

Inference of Search Directions

The variants of parallel Particle Swarm Optimization discussed in Part II demonstrate that effective parallel optimization algorithms can use communication sparingly without requiring centralized coordination. We now consider how the need for interaction can be further reduced by identifying search directions along which optimization is less interdependent. Existing serial algorithms use Principal Component Analysis to define a rotation of basis that reduces the dependence between search directions. If not for the communication and coordination required by centralized PCA computation, this approach would allow processors to work on separate search directions efficiently. Chapter 8 proposes a model for decentralized inference of search directions that removes the need for centralized computation. It also introduces a new statistical distribution which is used by this model.

Chapter 8

Inference of Search Directions for Exploiting Separability in Parallel Optimization

Many optimization algorithms adapt search directions to exploit latent separability, but the adaptation techniques require centralized coordination that is inefficient in a parallel computational environment. We propose a statistical model that can be used on a processor to adapt its search direction with autonomous separation from other processors' directions while preferring promising directions. As part of this model, we introduce a new BinghamConjugate distribution which is a conjugate prior of the antipodally symmetric Bingham distribution that is used to model axial data. We describe the model, discuss how to perform inference efficiently in the model, and demonstrate that it can successfully adapt to narrow in on a search direction.

8.1 Introduction

For many optimization problems, parallel optimization algorithms are limited by the cost of communication and coordination between processors. With parallel computation as an afterthought, optimization algorithms are not designed to address these fundamental concerns.

The property of separability is naturally linked to parallel optimization. An n-dimensional function is *separable* if it can be expressed as $f(\mathbf{x}) = \sum_{i=1}^{n} f_i(x_i)$. Although functions do not in general satisfy this definition, it represents an ideal best case. If a function is separable, then optimization can be decomposed into a set of independent single-dimensional optimization problems. This might motivate an embarrassingly parallel algorithm that merely performs line search separately along each axis, which would require no communication or coordination other than assembling the final result. Of course, such an algorithm would rely on an extreme assumption of separability and would not be generally applicable.

Although objective functions are not separable in general, many serial optimization algorithms exploit separability, even if it is only present in a more loose sense. Popular algorithms for continuous optimization, such as Particle Swarm Optimization, Evolution Strategies, and Differential Evolution, can be interpreted as relying on implicit assumptions of loose separability. For example, Differential Evolution calculates the differences between points in a population and adds them to others, which assumes that independent improvements in two different directions can often be successfully combined. The time-honored Coordinate Descent algorithm, which has been parallelized [77], is based on a more explicit assumption of separability. This algorithm cycles through the directions of a fixed basis, performing a line search along each. Adaptive Coordinate Descent uses Principal Component Analysis to dynamically determine a rotated basis for coordinate descent [78]. Due to this dynamically rotated basis, Adaptive Coordinate Descent performs well for functions like the well-known Rosenbrock function, which has a parabola-shaped valley that is particularly challenging for coordinate descent [79]. However, parallelizations of these algorithms based on PCA would require communication of all sampled points from all processors every time a new set of directions is selected.

We address the problem of parallel adaptation by mathematically modeling autonomous separation. Specifically, we present a statistical model for inferring search directions on a single processor, given the current directions of other processors and a noisy test that returns “success” for promising directions that indicate loose separability. As an example, a test could indicate whether a pair of function evaluations along a given direction show an improvement consistent with separability in that direction (i.e., $f(\mathbf{x}_1 + \mathbf{v}) < f(\mathbf{x}_2 + \mathbf{v})$ if $f(\mathbf{x}_1 + \mathbf{v}) < f(\mathbf{x}_1)$ and $f(\mathbf{x}_1) < f(\mathbf{x}_2)$). This approach allows new directions to be chosen frequently, with directions communicated only intermittently. Furthermore, it does not require all sampled points to be processed in a centralized manner. The model avoids overlap with directions pursued by other processors and refines its predictions to narrow in on directions where the test succeeds. Given the current directions of other processors, a BinghamConjugate distribution defines the prior probabilities of directions. A series of Bingham samples using this prior distribution gives a variety of directions to try, and the successful samples are used to update the posterior BinghamConjugate distribution. We note that a set of BinghamConjugate distributions jointly defines a noisy linear transformation that can represent rotation, reflection, and shearing of the basis. With a prior distribution centered on a direction that is orthogonal to the modes of the other distributions, this model addresses the concern of a single processor adapting its direction with minimal coordination from others.

The remainder of the paper proceeds as follows. Section 8.2 discusses prerequisite statistical distributions by reviewing the Bingham distribution, introducing the new BinghamConjugate distribution, and proposing an MCMC algorithm for sampling from the BinghamConjugate distribution using Gibbs sampling on variables induced by eigendecomposition. Section 8.3 defines the statistical model for adapting search directions, which uses the Bingham and BinghamConjugate distributions. Section 8.4 demonstrates that the model can successfully narrow in on a direction. Finally, Section 8.5 concludes and discusses future work.

8.2 Bingham and BinghamConjugate Distributions

Directional distributions are probability distributions whose support is the surface of a hypersphere [80]. Antipodal symmetry, where a density $f(\cdot)$ satisfies $f(-\mathbf{x}) = f(\mathbf{x})$, is useful for modeling directions that represent bidirectional lines rather than unidirectional rays. The Bingham distribution is an antipodally symmetric directional distribution. It is useful for modeling axial directional data, such as the orientation of calcite grains in limestone [81]. A special case of the Bingham distribution has been used for analyzing two-dimensional landmark data [82]. Efficient sampling algorithms are available for the distribution [83, 84], and it is possible to perform Bayesian inference on the eigenvalues of the parameter of the Bingham distribution [85]. However, the lack of a conjugate prior has limited the use of the Bingham distribution in Bayesian analysis.

The probability density function of the Bingham distribution is[1]:

$$f_{\boldsymbol{x}}(\mathbf{x}) = [\mathcal{B}(\mathbf{A})]^{-1} \exp(-\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x}),\ \mathbf{x} \in S^{p-1} \tag{8.1}$$

where $\mathbf{A}$ is a symmetric $p \times p$ matrix, S^{p-1} is the unit sphere in $\mathbb{R}^p$, and:

$$\mathcal{B}(\mathbf{A}) = \int \exp(-\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x})\, dS^{p-1}(\mathbf{x}). \tag{8.2}$$

This distribution is equivalent to a centered multivariate normal constrained to the surface of the unit sphere. The Bingham distribution is a member of the exponential family, and its normalizing constant, while not expressible in closed form, is a confluent hypergeometric function of matrix argument with an available saddlepoint estimation algorithm [86]. The appendix in Section 8.7 reviews and proves known mathematical properties related to the Bingham distribution that are used throughout this section.

A few other distributions related to the Bingham distribution deserve mention. The Projected Normal distribution [87], is a multivariate normal projected to the surface of the unit sphere. This distribution has not been developed in its antipodally symmetric form, and its density function is intractable. The Complex Bingham distribution [82] can be considered a special case of the Bingham distribution where eigenvalues of the parameter matrix have even multiplicity. In this special case, simulation reduces to the truncated multivariate exponential distribution [88]. The normalizing constant for the Complex Bingham distribution can be derived in closed form, and a simple formula is available if no two eigenvalues are equal [82]. Unfortunately, these results are not applicable to the Bingham distribution in general.

[1] We prefer this more convenient form [84, 86] to the original $f_{\boldsymbol{x}}(\mathbf{x}) = [\mathcal{B}(-\mathbf{A})]^{-1} \exp(\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x})$.

We define a new distribution which we refer to as the Bingham conjugate distribution. If $\boldsymbol{A} \sim \text{BinghamConjugate}(\mathbf{V}, \text{n})$, then the probability density of $\boldsymbol{A}$ is the function:

$$f_{\boldsymbol{A}}(\mathbf{A}) = \mathcal{C}(\mathbf{V}, n)^{-1}[\mathcal{B}(\mathbf{A})]^{-n}\, \mathrm{etr}(-\mathbf{V}\mathbf{A}),\ \ \mathbf{A} \text{ pos. def.} \tag{8.3}$$

where $\mathbf{V}$ is a $p \times p$ positive definite matrix, $n < \mathrm{tr}(\mathbf{V})$, and $\mathcal{C}(\mathbf{V}, n)$ is the normalizing constant.

Markov Chain Monte Carlo (MCMC) sampling for matrix-valued distributions is not straightforward. One inefficient algorithm is Independent Metropolis–Hastings sampling using a Wishart proposal distribution, but this has an extremely low rate of acceptance. In Independent Metropolis–Hastings, each candidate sample is distributed independently of the previous sample, rather than centered on the previous sample as in a typical Metropolis–Hastings algorithm [89, p. 276]. In this case, the proposal value is a $\text{Wishart}(2\mathbf{V}^{-1}, p+1)$ sample[90], and the acceptance ratio is $(\mathcal{B}(\mathbf{A}) - \mathcal{B}(\tilde{\mathbf{A}}))^n$. For a 10-dimensional BinghamConjugate distribution, the naïve sampler accepted only 15 out of 4.6×10^9 proposed values after an initial burn, and it appears to converge to an acceptance rate no greater than 10^{-9}. Although this algorithm is inefficient, we describe a MCMC sampling algorithm that is efficient.

Section 8.2.1 describes properties of this new distribution and shows that it is a conjugate prior of the Bingham distribution. Section 8.2.2 presents an MCMC sampler for this matrix-values distribution. Additionally, the appendix in Section 8.7 includes a proof of results presented in this paper, and appendix in Section 8.8 briefly reviews existing algorithms which are required for implementing the sampler.

8.2.1 Bingham Conjugate Distribution

We establish that Equation 8.3 is a valid probability density function:

Theorem 1. *The integral of the unnormalized probability density function of the Bingham conjugate distribution,*

$$\int_{\mathbf{A}>0} [\mathcal{B}(\mathbf{A})]^{-n} \operatorname{etr}(-\mathbf{V}\mathbf{A}) d\mathbf{A}, \tag{8.4}$$

converges if $\mathbf{V}$ *is positive definite and* $n < \operatorname{tr}(\mathbf{V})$*.*

The proof of this theorem is in the appendix in Section 8.7.

The parameter of the Bingham conjugate distribution is related to a sufficient statistic of the Bingham distribution. The probability density function of a matrix $\boldsymbol{X}$ of independent Bingham-distributed samples can be rewritten as:

$$\begin{aligned} f_{\boldsymbol{X}}(\mathbf{X}) &= \prod_{i=1}^{n} [\mathcal{B}(\mathbf{A})]^{-1} \exp\left(-\mathbf{x}_i^{\mathsf{T}} \mathbf{A} \mathbf{x}_i\right) \\ &= [\mathcal{B}(\mathbf{A})]^{-n} \operatorname{etr}\left(-\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^{\mathsf{T}} \mathbf{A}\right) \\ &= [\mathcal{B}(\mathbf{A})]^{-n} \operatorname{etr}\left(-\mathbf{X}\mathbf{X}^{\mathsf{T}}\mathbf{A}\right), \end{aligned} \tag{8.5}$$

which only depends on the data through the function $\mathbf{X}\mathbf{X}^{\mathsf{T}}$, so by the Fisher–Neyman factorization theorem, $\boldsymbol{X}\boldsymbol{X}^{\mathsf{T}}$ is a sufficient statistic for $\mathbf{A}$.

The new distribution is a conjugate prior of the Bingham distribution. Suppose $\boldsymbol{A} \sim \operatorname{BinghamConjugate}(\mathbf{V}, n)$, and $\boldsymbol{A}$ is the prior distribution of $\mathbf{x} \sim \operatorname{Bingham}(\mathbf{A})$. According to Bayes' theorem, the posterior density is:

$$f_{\boldsymbol{A}|\boldsymbol{x}}(\mathbf{A} \mid \mathbf{x}) \propto [\mathcal{B}(\mathbf{A})]^{-(n+1)} \operatorname{etr}(-(\mathbf{V} + \mathbf{x}\mathbf{x}^{\mathsf{T}})\mathbf{A}). \tag{8.6}$$

Furthermore, $\operatorname{tr}(\mathbf{V} + \mathbf{x}\mathbf{x}^{\mathsf{T}}) = \operatorname{tr}(\mathbf{V}) + \|\mathbf{x}\| = \operatorname{tr}(\mathbf{V}) + 1 > n + 1$. Thus, $\boldsymbol{A} \mid \mathbf{x} \sim \operatorname{BinghamConjugate}(\mathbf{V} + \mathbf{x}\mathbf{x}^{\mathsf{T}}, n + 1)$. With m observations represented as column vectors of a matrix $\mathbf{X}$, the posterior distribution is $\operatorname{BinghamConjugate}(\mathbf{V} + \mathbf{X}\mathbf{X}^{\mathsf{T}}, n + m)$. The posterior update increments the parameter n by the number of observations and adds $\mathbf{X}\mathbf{X}^{\mathsf{T}}$, the sufficient statistic of the Bingham distribution, to the parameter $\mathbf{V}$.

One way to construct the parameter $\mathbf{V}$ is as a scatter matrix from a collection $\{\mathbf{y}_1, \ldots, \mathbf{y}_k\}$ of k pseudo-samples of maximum rank on the unit sphere. In this case, $\operatorname{tr}(\mathbf{V}) = \sum_{i=1}^{k} \operatorname{tr}(\mathbf{y}_i \mathbf{y}_i^{\mathsf{T}}) = \sum_{i=1}^{k} \|\mathbf{y}_i\| = k$, so the parameter n must be less than the number k of pseudo-samples.

8.2.2 Sampling Algorithm

We sample from the Bingham conjugate distribution using Gibbs sampling on variables $\boldsymbol{Q}$ and $\boldsymbol{\Lambda}$ induced by the eigendecomposition of the random matrix: $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^{\mathsf{T}}$. This change of variables has Jacobian $\prod_{i<j} |\lambda_i - \lambda_j|$ [91], and because $\mathcal{B}(\mathbf{A}) = \mathcal{B}(\mathbf{\Lambda})$, we may also write $\mathcal{B}$ as $\mathcal{B}(\lambda_1, \ldots, \lambda_p)$. With this change of variables, the joint density of $\boldsymbol{\Lambda}$ and $\boldsymbol{Q}$ is:

$$
\begin{aligned}
f_{\boldsymbol{Q},\boldsymbol{\Lambda}}\left(\mathbf{Q},\, \mathbf{\Lambda}\right) &\propto [\mathcal{B}(\mathbf{\Lambda})]^{-n} \operatorname{etr}\left[-\mathbf{V}\left(\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{\mathsf{T}}\right)\right] \prod_{j<k} |\lambda_j - \lambda_k| \\
&= [\mathcal{B}(\mathbf{\Lambda})]^{-n} \operatorname{etr}\left[-\left(\mathbf{Q}^{\mathsf{T}}\mathbf{V}\mathbf{Q}\right)\mathbf{\Lambda}\right] \prod_{j<k} |\lambda_j - \lambda_k| \\
&= [\mathcal{B}(\mathbf{\Lambda})]^{-n} \exp\left[-\sum_{i=1}^{p} \left(\mathbf{Q}^{\mathsf{T}}\mathbf{V}\mathbf{Q}\right)_{ii} \lambda_i\right] \prod_{j<k} |\lambda_j - \lambda_k|
\end{aligned}
$$

because $\mathbf{\Lambda}_{i,j} = 0$ for i and j where $i \neq j$. Thus:

$$
\begin{aligned}
f_{\boldsymbol{Q},\boldsymbol{\Lambda}}\left(\mathbf{Q},\, \mathbf{\Lambda}\right) &\propto [\mathcal{B}(\mathbf{\Lambda})]^{-n} \exp\left(-\sum_{i=1}^{p} \mathbf{q}_i^{\mathsf{T}}\mathbf{V}\mathbf{q}_i \lambda_i\right) \prod_{j<k} |\lambda_j - \lambda_k| \\
&= [\mathcal{B}(\mathbf{\Lambda})]^{-n} \left[\prod_{i=1}^{p} \exp\left(-\left(\mathbf{q}_i^{\mathsf{T}}\mathbf{V}\mathbf{q}_i\right)\lambda_i\right)\right] \prod_{j<k} |\lambda_j - \lambda_k| .
\end{aligned}
\tag{8.7}
$$

The variables $\boldsymbol{\Lambda}$ and $\boldsymbol{Q}$ are both sampled with Gibbs sampling, which we outline here and then describe in more detail in Sections 8.2.2.1 and 8.2.2.2. First, each eigenvalue λ_i is sampled as a constrained exponential distribution by slice sampling with auxiliary variables U and $V_{i,j}$. Alternatively, it may be possible to sample from the eigenvalues using a Metropolis–Hastings approach [85], but this would likely introduce greater autocorrelation than the presented direct sampling approach. Second, the matrix of eigenvectors $\boldsymbol{Q}$ is sampled by rotating pairs of vectors with Metropolis–Hastings. The proposed angle of rotation is drawn from a normal-variate candidate distribution. The eigenvalues and eigenvectors are initialized by setting $\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{\mathsf{T}} = \mathbf{V}^{-1}$.

8.2.2.1 Sampling the Eigenvalues

The eigenvalues are sampled as constrained exponential random variables with the introduction of auxiliary variables. We introduce the following auxiliary variables for slice sampling [92] of the density in equation (8.7):

$$U \sim \mathcal{U}\left(0,\, [\mathcal{B}(\mathbf{\Lambda})]^{-n}\right) \tag{8.8}$$

$$V_{j,k} = V_{k,j} \sim \mathcal{U}\left(0,\, |\lambda_j - \lambda_k|\right). \tag{8.9}$$

Sampling of U and $V_{j,k}$ from their full conditionals is straightforward from their definitions (note that in practice, it is numerically necessary to sample $\ln U$ directly in log-space: $\ln U \sim -Y - n \ln \mathcal{B}(\mathbf{\Lambda})$ where $Y \sim$ Exponential(1)). The full conditional of λ_i given the auxiliary variables is Exponential($\mathbf{q}_i^{\mathsf{T}}\mathbf{V}\mathbf{q}_i$) constrained such that the new bounds imposed by λ_i on U and $V_{j,k}$ are consistent with their current values u and $v_{j,k}$. This constraint is equivalent to the set:

$$S_i(u, (v_{i,j})_j) = \{\lambda_i; \mathcal{B}(\lambda_i, (\lambda_j)_{j\neq i})^{-n} \geq u\} \cap \bigcap_{j\neq i} \{\lambda_i; |\lambda_i - \lambda_j| \geq v_{i,j}\}. \tag{8.10}$$

We define the function $h(\lambda_i) = \mathcal{B}(\lambda_i, (\lambda_j)_{j\neq i}) - u^{-1/n}$ which is monotone decreasing and convex (by Theorem 2 in the appendix in Section 8.7) and whose root is a lower bound on λ_i. If $h(0) < 0$, then the root of h is less than 0, and thus u does not constrain λ_i. Otherwise, the root r of $h(0)$ can be found by a root finding algorithm such as Brent's method. To find a suitable bracket for a root finder, note that $h(0) > 0$ and $h(2^k \lambda_{t-1,i}) < 0$ for some reasonably small integer k. Thus, the set constraining λ_i simplifies to an intersection of intervals:

$$S_i(u, (v_{i,j})_j) = (\max(0, r),\, \infty) \cap \bigcap_{j \neq i} (\lambda_j - v_{i,j},\, \lambda_j + v_{i,j}), \tag{8.11}$$

which can be converted to a disjoint intersection by sorting the list of intervals by their lower bounds and merging overlapping intervals, which is then trivially converted to a disjoint union of intervals. To sample from the exponential distribution constrained to the disjoint union of intervals $\bigcup_k (a_k,\, b_k)$, first sample j proportional to the probability mass of each interval: $\exp[-(\mathbf{q}_i^\top \mathbf{V} \mathbf{q}_i) a_k] - \exp[-(\mathbf{q}_i^\top \mathbf{V} \mathbf{q}_i) b_k]$ (after normalization). Then, if $b_j = \infty$ sample from the left-truncated exponential distribution with:

$$\lambda_i = a_j - (\mathbf{q}_i^\top \mathbf{V} \mathbf{q}_i)^{-1} \ln[\mathcal{U}(0,\, 1)], \tag{8.12}$$

or if $b_i < \infty$, sample from the left- and right-truncated exponential distribution with:

$$\lambda_i = a_j - (\mathbf{q}_i^\top \mathbf{V} \mathbf{q}_i)^{-1} \ln \left[\mathcal{U} \left(e^{-\mathbf{q}_i^\top \mathbf{V} \mathbf{q}_i (b_j - a_j)},\, 1 \right) \right]. \tag{8.13}$$

8.2.2.2 Sampling the Eigenvectors

The eigenvectors are constrained to be an orthogonal set, so it is impossible to sample from a single eigenvector independent of the others. However, it is possible to jointly sample from a pair of eigenvectors which span a plane that is orthogonal to all of the other fixed eigenvectors. Rotating these two vectors within this plane preserves orthogonality of the entire set of eigenvectors. Sampling the entire set of eigenvectors uses a multi-stage Gibbs sampler that iteratively rotates each of the $p(p-1)/2$ pairs of eigenvectors.

Each pair of eigenvectors is rotated by a Metropolis–Hastings sampler using a normal-variate proposal for the angle of rotation. Suppose $\mathbf{Q}$ is the current matrix of eigenvectors and that the candidate eigenvalue matrix $\tilde{\mathbf{Q}}$ is equivalent to $\mathbf{Q}$ but with columns i and j rotated within their span by an angle of θ. Specifically, rotating the vectors $\mathbf{q}_i$ and $\mathbf{q}_j$ gives the new vectors:

$$\begin{aligned}\tilde{\mathbf{q}}_i &= (\cos\theta)\mathbf{q}_i + (\sin\theta)\mathbf{q}_j\\ \tilde{\mathbf{q}}_j &= -(\sin\theta)\mathbf{q}_i + (\cos\theta)\mathbf{q}_j.\end{aligned}$$

From the joint density of $\boldsymbol{\Lambda}$ and $\boldsymbol{Q}$ in equation (8.7) and because the proposal distribution is symmetric, the acceptance ratio for $\tilde{\mathbf{Q}}$ is:

$$\begin{aligned}\frac{f_{\boldsymbol{Q},\boldsymbol{\Lambda}}\left(\tilde{\mathbf{Q}},\,\mathbf{\Lambda}\right)}{f_{\boldsymbol{Q},\boldsymbol{\Lambda}}(\mathbf{Q},\,\mathbf{\Lambda})} &= \exp\left(\sum_k \lambda_k \mathbf{q}_k^\mathsf{T}\mathbf{V}\mathbf{q}_k - \sum_k \lambda_k \tilde{\mathbf{q}}_k^\mathsf{T}\mathbf{V}\tilde{\mathbf{q}}_k\right)\\ &= \exp\left(\lambda_i\left(\mathbf{q}_i^\mathsf{T}\mathbf{V}\mathbf{q}_i - \tilde{\mathbf{q}}_i^\mathsf{T}\mathbf{V}\tilde{\mathbf{q}}_i\right) + \lambda_j\left(\mathbf{q}_j^\mathsf{T}\mathbf{V}\mathbf{q}_j - \tilde{\mathbf{q}}_j^\mathsf{T}\mathbf{V}\tilde{\mathbf{q}}_j\right)\right).\end{aligned} \tag{8.14}$$

In terms of θ instead of $\tilde{\mathbf{q}}_i$ and $\tilde{\mathbf{q}}_j$, the acceptance ratio is:

$$\exp((\lambda_i - \lambda_j)(\sin^2\theta)\mathbf{q}_i^\mathsf{T}\mathbf{V}\mathbf{q}_i + (\lambda_j - \lambda_i)(\sin 2\theta)\mathbf{q}_i^\mathsf{T}\mathbf{V}\mathbf{q}_j + (\lambda_j - \lambda_i)(\sin^2\theta)\mathbf{q}_j^\mathsf{T}\mathbf{V}\mathbf{q}_j). \tag{8.15}$$

In practice, manually tuning the proposal distribution is undesirable, particularly for applications with greater than three dimensions. We develop a rule of thumb that serves as an acceptable standard deviation of the candidate normal distribution. Note that $\mathbf{q}_i^\mathsf{T}\mathbf{V}\mathbf{q}_i$ lies between the minimum and maximum eigenvalues of $\mathbf{V}$, and if $\mathbf{q}_i$ is the i^{th} eigenvector of $\mathbf{V}$, then $\mathbf{q}_i^\mathsf{T}\mathbf{V}\mathbf{q}_i$ is equal to the i^{th} eigenvalue of $\mathbf{V}$. Thus, we set the standard deviation of the candidate distribution for the rotation between the i^{th} and j^{th} eigenvectors of $\boldsymbol{Q}$ to:

$$\sigma_{i,j} = |(\lambda_i - \lambda_j)(l_i - l_j)|^{-1/2} \tag{8.16}$$

where l_i and l_j are the i^{th} and j^{th} eigenvalues of $\mathbf{V}$ respectively. While this rule of thumb seems to work well in practice, we expect that manual tuning or some other formula may work better in specific cases.

Alternatively, the eigenvectors can be sampled with Independent Metropolis–Hastings, which results in a much lower acceptance ratio and higher autocorrelation, but which does not involve tuning parameters. The marginal distribution of the eigenvectors of a Wishart distribution is intractable, so this requires the introduction of an auxiliary variable $\boldsymbol{M}$, which for convenience has density:

$$f_{\boldsymbol{M}}(\mathbf{M}) = \begin{cases} c_{\boldsymbol{M}}^{-1} \prod_{i<j} |\mu_i - \mu_j| & \text{if } 0 < \mu_i < k_{\boldsymbol{M}}, \\ 0 & \text{otherwise.} \end{cases}$$

where $k_{\boldsymbol{M}}$ is an arbitrary constant and $c_{\boldsymbol{M}} = \int_{0<\mu_i<k_{\boldsymbol{M}}} \prod_i \prod_{j=1}^{i-1} |\mu_i - \mu_j| d\mu_i$. Suppose the current sample of $\boldsymbol{A}$ is $\mathbf{A}_{t-1} = \mathbf{Q}_{t-1}\boldsymbol{\Lambda}_{t-1}\mathbf{Q}_{t-1}^{\mathsf{T}}$ and the current sample of $\boldsymbol{M}$ is $\mathbf{M}_{t-1}$. Set $\tilde{\mathbf{Q}}$ and $\tilde{\mathbf{M}}$ to the eigendecomposition of a sample $\tilde{\mathbf{Q}}\tilde{\mathbf{M}}\tilde{\mathbf{Q}}^{\mathsf{T}}$ from the Wishart$(2\mathbf{V}^{-1}, p+1)$ distribution [90]. After simplification, the Metropolis–Hastings acceptance ratio is:

$$\begin{cases} \operatorname{etr}\left[\mathbf{V}\left(\tilde{\mathbf{Q}}\left(\tilde{\mathbf{M}} - \boldsymbol{\Lambda}_{t-1}\right)\tilde{\mathbf{Q}}^{\mathsf{T}} - \mathbf{Q}_{t-1}\left(\mathbf{M}_{t-1} - \boldsymbol{\Lambda}_{t-1}\right)\mathbf{Q}_{t-1}^{\mathsf{T}}\right)\right] & \text{if all } \tilde{\mu}_i < k_{\boldsymbol{M}}, \\ 0 & \text{otherwise.} \end{cases} \tag{8.17}$$

This algorithm works, but in most situations, sampling by pairwise rotation of eigenvectors results in significantly lower auto-correlation (see Section 8.4).

8.3 Model

We propose a model by which a processor can autonomously adapt its search direction while maintaining separation from other processors' directions. Adaptation is guided by a given test for indicating whether a direction is promising. The test yields either true or false for a given direction and is not assumed to consistently give the same result for any specific input.

At the heart of the model are a BinghamConjugate-distributed random matrix $\boldsymbol{A}$ and a Bingham-distributed random vector $\boldsymbol{y}$. The prior parameters of $\boldsymbol{A}$ favor directions clustered around the direction that is orthogonal to the directions of other processors. Because the

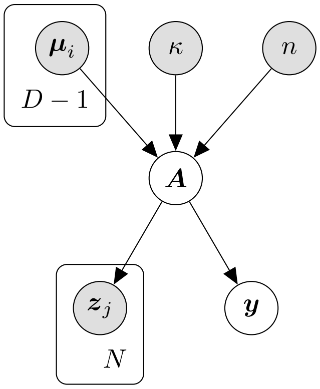


Figure 8.1: Conditional independence assumptions of the model.

matrix-valued parameter is most naturally interpreted as a scatter matrix, we use samples from a Von Mises Fisher distribution to generate it. The posterior parameters of $\boldsymbol{A}$ are updated in response to observations of directions for which the test returns success. Samples from $\boldsymbol{y}$ predict future promising values according to these posterior parameters.

Figure 8.1 illustrates the conditional independence assumptions of the model. The vectors $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_{p-1}$, where p is the number of dimensions, represent the set of other processors' directions, which are to be avoided. Let $\boldsymbol{\mu}_p$ be one of the two antipodally symmetric orthonormal vectors that are orthogonal to $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_{p-1}$. This vector $\boldsymbol{\mu}_p$, along with the concentration parameter κ, defines the distribution of an auxiliary random variable $\boldsymbol{v}$. The expected scatter matrix of $\boldsymbol{v}$ and the degrees of freedom parameter n determine the parameters of the prior distribution of the BinghamConjugate random matrix $\boldsymbol{A}$. The model is defined by the random variables with the following distributions:

$$\boldsymbol{v} \sim \mathrm{VonMisesFisher}(\boldsymbol{\mu}_p,\, \kappa) \tag{8.18}$$

$$\boldsymbol{A} \sim \mathrm{BinghamConjugate}\left(n\,\mathbb{E}\left[\boldsymbol{v}\boldsymbol{v}^{\mathsf{T}}\right],\, n\right) \tag{8.19}$$

$$\boldsymbol{z}_j \sim \mathrm{Bingham}(\boldsymbol{A}) \tag{8.20}$$

$$\boldsymbol{y} \sim \mathrm{Bingham}(\boldsymbol{A}). \tag{8.21}$$

In intuitive terms, $\boldsymbol{\mu}_p$ and $-\boldsymbol{\mu}_p$ are the modes of $\boldsymbol{y}$, n is the number of pseudo-samples represented in the prior distribution (raising n increases the confidence of the prior), and κ is the dispersion of prior pseudo-samples. The set of observations $\{\boldsymbol{z}_j\}_N$ represent successful directions from past exploration and reinforce the posterior distribution, which has distribution:

$$\boldsymbol{A} \sim \text{BinghamConjugate}\left(n\,\mathbb{E}[\boldsymbol{v}\boldsymbol{v}^\mathsf{T}] + \sum_{i=1}^{N} \boldsymbol{z}_j \boldsymbol{z}_j^\mathsf{T},\; n+N \right). \tag{8.22}$$

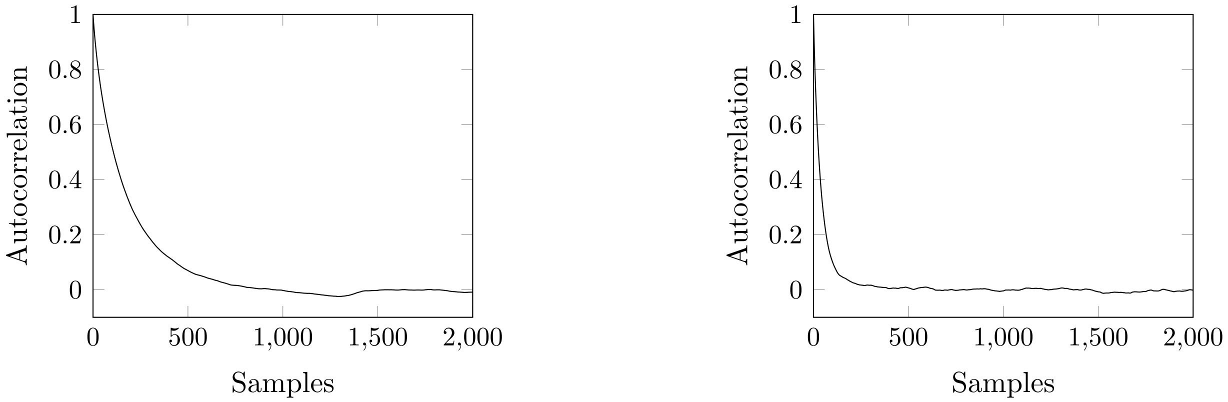
The posterior predictive $\boldsymbol{y}$ is a random variable for a new direction with probabilities determined by this posterior distribution. Samples from $\boldsymbol{y}$ are inclined to be orthogonal from other processors' directions (due to the prior) and are attracted to the previous successful observations. Thus, samples from $\boldsymbol{y}$ are promising directions which can be used for future exploration.

Inference in this model is straightforward and efficient. Given other processors' directions $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_{p-1}$, the parameter $\boldsymbol{\mu}_p$ is computed by applying the Gram-Schmidt process to these vectors along with an arbitrary random vector. For $\mathbf{e}_1 = [10\ldots0]^\mathsf{T}$, the value $\mathbb{E}\left[\mathbf{e}_1\mathbf{e}_1^\mathsf{T}\right]$ can be precomputed for a given value of κ by averaging the scatter matrix from a large number of samples from the distribution VonMisesFisher$(\mathbf{e}_1, \kappa)$. Then the term $\mathbb{E}\left[\boldsymbol{v}\boldsymbol{v}^\mathsf{T}\right]$ is a simple rotation of $\mathbb{E}\left[\mathbf{e}_1\mathbf{e}_1^\mathsf{T}\right]$. The posterior distribution is conveniently updated by observations with the closed-form update in Equation 8.22. Finally, the distribution of $\boldsymbol{y}$ is efficiently sampled with the **BinghamSampler** algorithm in Section 8.8.

8.4 Results

We consider first the performance of the BinghamConjugate sampler and then the behavior of the full model.

Figure 8.2 shows the autocorrelation of the first two eigenvalues of a three-dimensional BinghamConjugate distribution using slice sampling for the eigenvalues and Metropolis–



(a) First Eigenvalue

(b) Second Eigenvalue

Figure 8.2: Autocorrelation plots for the first two eigenvalues of samples from the Bingham-Conjugate distribution.

Hastings pairwise rotations for the eigenvectors. The parameter V is the scatter matrix of 30 samples from a von Mises–Fisher distribution with $\kappa = 10$, and the parameter $n = 33$. The autocorrelation of the first eigenvalue is very low by 500 samples and is 0 by 1000 samples. The autocorrelation of the second eigenvalue (and the third eigenvalue) drops off more quickly and is 0 by 250 samples. For the same distribution, using Independent Metropolis–Hastings with for the eigenvectors with an auxiliary variable, the autocorrelation is worse by more than an order of magnitude.

In evaluating the behavior of this model, the test for indicating whether a new direction is promising is defined by a function of the angle between the new direction and a fixed target direction. If the new direction is close to the target direction, then the test returns "success" with high probability, and if the new direction is far from the target direction, then the test returns "failure" with high probability. Specifically, we fix $\boldsymbol{\mu}_p$, choose a target direction $\boldsymbol{\tau}$ a fixed angle away, and iteratively sample from $\boldsymbol{y}$, computing the angle between the sampled vector $\mathbf{y}$ and $\boldsymbol{\tau}$. The probability of the test returning success is given by the following Gaussian function of the angle: $\exp\left(-C\alpha^2\right)$, where $\alpha = \cos^{-1}\left(|\mathbf{y}^\mathsf{T}\boldsymbol{\mu}_p|\right)$ is the angle between the directions $\mathbf{y}$ and $\boldsymbol{\mu}_p$. This function assigns probability 1 at the target direction and assigns probability approaching 0 as α increases. This function is shown in Figure 8.3 for

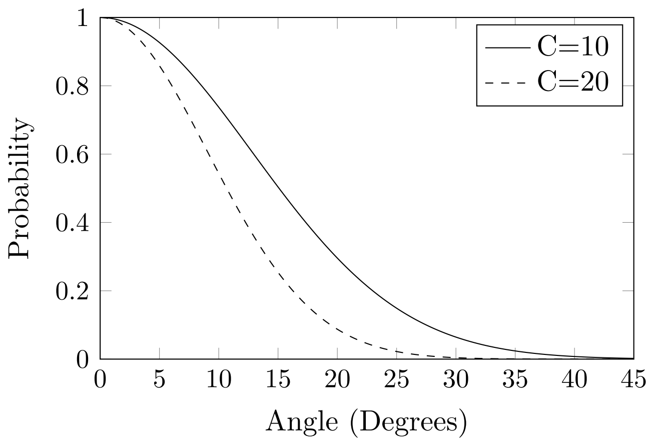


Figure 8.3: Function for determining the probability of success from the angle $|\mathbf{y}^\mathsf{T}\boldsymbol{\mu}_p|$ with concentration parameters of 10 and 20.

values $C = 10$ and $C = 20$ of the concentration parameter. If the sample is labeled a success, then it is added as the value of the next $\mathbf{z}_j$.

We demonstrate results with the target direction $\boldsymbol{\tau}$ an angle of 15 degrees from $\boldsymbol{\mu}_p$ and with parameters $\kappa = 6$ and $n = p$, which is the loosest prior value of n. We first consider $p = 3$, the number of dimensions for which geometry most closely aligns with intuition. We next consider $p = 5$ for two different values of C to demonstrate the effect of the concentration parameter. Finally, we consider $p = 10$ to show the behavior for a higher number of dimensions. Note that the value of the concentration parameter is not directly comparable between different values of p.

Figure 8.4 shows the average angle of samples from the posterior predictive distribution of $\boldsymbol{y}$ at each iteration with $p = 3$ and $C = 20$. This value of C corresponds with the function shown in Figure 8.3 that selects the probability of a sample being labeled a success. Directions within about 10 degrees from the target have a greater than 50% probability, while directions further away have a less than 50% probability. At iteration 300, the samples from the posterior predictive distribution have a mean angle from the target direction of less than 12 degrees.

Figure 8.5 shows results with $p = 5$. As the number of dimensions increases, the average angle between a point sampled from the prior and the target direction increases (in a

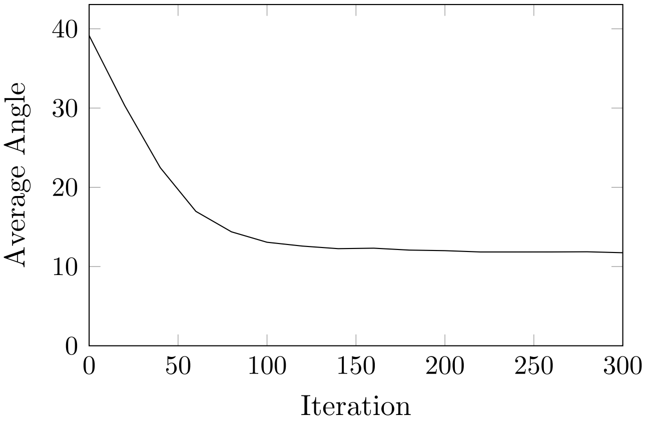
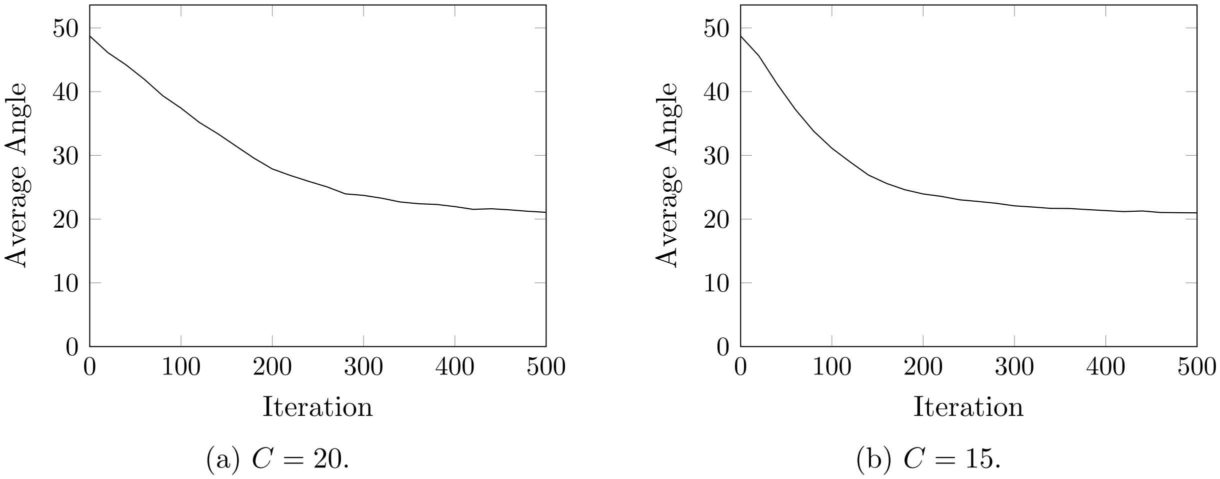


Figure 8.4: Average angle of predicted values of $\boldsymbol{y}$ when $p = 3$ and $C = 20$. The angle test returns a probability of success of 50% at around 10 degrees from the target direction.



(a) $C = 20$.

(b) $C = 15$.

Figure 8.5: Average angle of predicted values of $\boldsymbol{y}$ when $p = 5$ for different values of C. The angle test returns a probability of success of 50% at around 15 degrees from the target direction.

sense, points are further apart in higher dimensional space). In Figure 8.5a, the concentration parameter is $C = 20$. In this case, the success rate is only 10%, so convergence is slower than in the 3-dimensional case. In Figure 8.5b, the concentration parameter is set to a lower value of $C = 15$, so the success rate is 15% and the convergence is reasonable.

Figure 8.6 shows results with $p = 10$ and $C = 10$, where the success rate is 15%. Directions within about 15 degrees from the target have a greater than 50% probability of being labeled successes, while directions further away have a less than 50% probability. In the 10-dimensional case, the model converges to a posterior predictive distribution that is

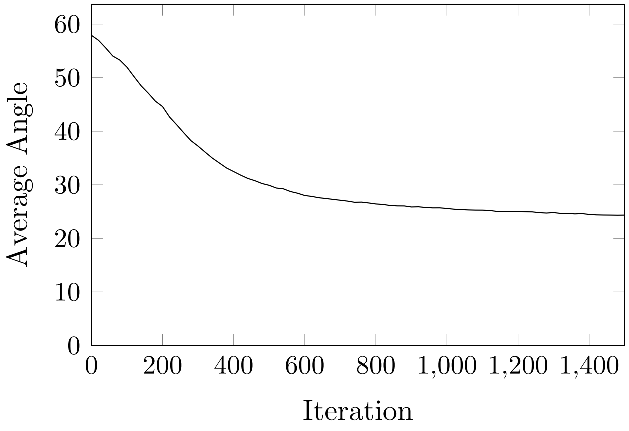


Figure 8.6: Average angle of predicted values of $\boldsymbol{y}$ when $p = 10$ and $C = 10$. The angle test returns a probability of success of 50% at around 15 degrees from the target direction.

less tight than in the 3-dimensional case, but it still does a reasonable job of finding the target search direction. In particular, note that the net improvement from iteration 0 to iteration 1,500 is 33 degrees, which is greater than the net improvement of 27 degrees in the 3-dimensional case (shown in Figure 8.4).

8.5 Conclusion

We have presented a new distribution that is a conjugate prior of the Bingham distribution. We have proved that the Bingham conjugate distribution has finite integral and is therefore a valid distribution. We have also detailed a practical Gibbs sampling algorithm for sampling from the eigenvalues and eigenvectors of this distribution and have demonstrated that this algorithm has manageably low autocorrelation. The Bingham conjugate distribution allows for closed-form inference of the Bingham distribution and thus facilitates Bayesian analysis of antipodally symmetric directional data.

We have presented a model based on the conjugate pair of the BinghamConjugate and Bingham distributions which addresses the need in parallel optimization for adapting a processor's search direction. The model constrains directions to ensure separation from other processors' directions, and it focuses on promising directions given a test, such as one that identifies directions that can exploit latent separability. Unlike techniques based on Principal

Component Analysis which require all sampled points to be communicated for centralized processing, this approach allows for inference to be performed locally. This inference requires only the mode of the distribution of directions of each processor (which need not be updated frequently) and sufficient evaluated points from other processors for performing the test. We have demonstrated that the model can successfully perform inference on search directions, and we have discussed techniques for performing this inference efficiently.

This work raises many interesting research questions. First, we anticipate that there may be many possible tests for identifying promising directions and detecting separability. For example, a test could take a point $\mathbf{z}$ and perform line search from this point along a direction $\mathbf{y}$, returning true if an improved point is found. Another test could take two given points $\mathbf{x}_1$ and $\mathbf{x}_2$ where $f(\mathbf{x}_1) < f(\mathbf{x}_2)$, perform line search from point $\mathbf{x}_1$ along direction $\mathbf{y}$ to the point $(\mathbf{x}_1 + d\mathbf{y})$, and return true if $(f(\mathbf{x}_1 + d\mathbf{y}) < f(\mathbf{x}_1)) = (f(\mathbf{x}_2 + d\mathbf{y}) < f(\mathbf{x}_2))$. Second, the model detailed in Section 8.3 may be further refined. For example, the model may be expanded to incorporate negative test results or adapted for use with other types of tests that may not be binary. We note that there is an efficient algorithm, **BinghamConstant** in Section 8.8, for estimating the normalizing constant of the Bingham distribution, which would facilitate inference and sampling in mixture models involving the BinghamConjugate and Bingham distributions. Third, we foresee a wide variety of parallel optimization algorithms based on this model and additional derivative models. These algorithms may vary in what type of line search is performed, which type of test is used for identifying successful directions, how often new directions are communicated to other processors, how new priors are chosen after receiving updated directions from other processors, which points and evaluations are shared, and how old observations are pruned. Finally, all of these factors when applied to a set of interacting processors may bring about interesting emergent behaviors that are worthy of study.

8.6 Appendix: Properties of the Bingham Distribution

Rotating a Bingham random variable is equivalent to rotating its parameter matrix as in, for example, eigendecomposition. Suppose that $\boldsymbol{x} \sim \text{Bingham}(\mathbf{A})$ and that $\mathbf{Q}$ is an orthogonal matrix. Then for all $\mathbf{x}$, $\|\mathbf{Qx}\| = \|\mathbf{Q}^{\mathsf{T}}\mathbf{x}\| = \|x\|$, so $\mathbf{x} \in S^{p-1}$ if and only if $\mathbf{Qx} \in S^{p-1}$. The Jacobian of the linear transformation $\mathbf{x} = \mathbf{Q}^{\mathsf{T}}\mathbf{y}$ is simply $|\det(\mathbf{Q}^{\mathsf{T}})| = 1$, so the density of $\boldsymbol{y} = \mathbf{Q}\boldsymbol{x}$ is:

$$
\begin{aligned}
f(\mathbf{y}) &\propto \exp\left(-\left(\mathbf{Q}^{\mathsf{T}}\mathbf{y}\right)^{\mathsf{T}} \mathbf{A} \left(\mathbf{Q}^{\mathsf{T}}\mathbf{y}\right)\right), \; \mathbf{y} \in S^{p-1} \\
&= \exp\left(-\mathbf{y}^{\mathsf{T}} \left(\mathbf{QAQ}^{\mathsf{T}}\right) \mathbf{y}\right).
\end{aligned}
$$

Therefore, if $\boldsymbol{x} \sim \text{Bingham}(\mathbf{A})$, then $\mathbf{Q}\boldsymbol{x} \sim \text{Bingham}(\mathbf{QAQ}^{\mathsf{T}})$. As a consequence, we can assume without loss of generality that the parameter matrix is a diagonal matrix of eigenvalues.

The density of the Bingham distribution is invariant under addition of the eigenvalues of the parameter matrix by a constant. Suppose, without loss of generality, that $\mathbf{A}$ is a diagonal matrix of eigenvalues. Then the density of $\boldsymbol{x} \sim \text{Bingham}(\mathbf{A} + k\mathbf{I})$ is:

$$
\begin{aligned}
f(\mathbf{x}) &\propto \exp\left(-\mathbf{x}^{\mathsf{T}} \left(\mathbf{A} - k\mathbf{I}\right) \mathbf{x}\right), \; \mathbf{x} \in S^{p-1} \\
&= \exp\left(-\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x} - k\mathbf{x}^{\mathsf{T}}\mathbf{x}\right) \\
&= \exp\left(-\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x} - k\right)
\end{aligned}
$$

because $\mathbf{x}^{\mathsf{T}}\mathbf{x} = \|\mathbf{x}\|^2 = 1$. Therefore, $\boldsymbol{x} \sim \text{Bingham}(\mathbf{A})$. As a consequence, we can assume without loss of generality that the smallest eigenvalue of $\mathbf{A}$ is 0, and thus $\mathbf{A}$ is a positive semi-definite matrix.

The Bingham distribution is equivalent to a centered multivariate normal constrained to the surface of the unit sphere. To show this, first let $\boldsymbol{y} \sim \mathcal{N}\left(\mathbf{0}, \mathbf{A}^{-1}\right)$, and let $\boldsymbol{x} \sim$

Bingham($\frac{1}{2}\mathbf{A}$). Then $\boldsymbol{y}$ and $\boldsymbol{x}$ have densities:

$$f_{\boldsymbol{y}}(\mathbf{y}) = (2\pi)^{-p/2}\det(\mathbf{A})^{1/2}\exp\left(-\frac{1}{2}\mathbf{y}^{\mathsf{T}}\mathbf{A}\mathbf{y}\right)$$
$$f_{\boldsymbol{x}}(\mathbf{x}) \propto \exp\left(-\mathbf{x}^{\mathsf{T}}\left(\frac{1}{2}\mathbf{A}\right)\mathbf{x}\right),\ \mathbf{x} \in S^{p-1}.$$

Next, find the distribution of $\boldsymbol{y}$ conditioned on $\|\boldsymbol{y}\| = 1$ by restricting it to the sphere S^{p-1} and renormalizing:

$$\begin{aligned} f_{\boldsymbol{y}|\|\boldsymbol{y}\|=1}(\mathbf{y}) &= \frac{f_{\boldsymbol{y}}(\mathbf{y})}{\int f_{\boldsymbol{y}}(\mathbf{x})dS^{p-1}(\mathbf{x})},\ \mathbf{y} \in S^{p-1} \\ &\propto \exp\left(-\mathbf{y}^{\mathsf{T}}\left(\frac{1}{2}\mathbf{A}\right)\mathbf{y}\right),\ \mathbf{y} \in S^{p-1} \\ &\propto f_{\boldsymbol{x}}(\mathbf{y}) \end{aligned}$$

where S^{p-1} is the unit sphere in $\mathbb{R}^p$. Since proportional probability densities are equal, $f_{\boldsymbol{y}|\|\boldsymbol{y}\|=1} = f_{\boldsymbol{x}}$. Therefore, the distribution of $\boldsymbol{y} \mid \|\boldsymbol{y}\| = 1$ has the same density as the Bingham($\frac{1}{2}\mathbf{A}$) distribution.

8.7 Appendix: Proofs

In this appendix, we provide a series of proofs culminating in a proof of Theorem 1. These proofs demonstrate that the BinghamConjugate density has a finite integral and thus that the BinghamConjugate distribution is valid.

Theorem 2. *The Bingham normalizing constant, $\mathcal{B}(\mathbf{\Lambda})$, is a decreasing convex function of the eigenvalues.*

Proof. The partial derivatives of the Bingham normalizing constant are (using the Leibniz integration rule):

$$\frac{\partial}{\partial\lambda_i}\mathcal{B}(\mathbf{\Lambda}) = \frac{\partial}{\partial\lambda_i}\int \exp\left(-\sum_{i=1}^{p} x_i^2\lambda_i\right)\, dS^{p-1}(\mathbf{x})$$

$$
\begin{aligned}
&= \int \frac{\partial}{\partial \lambda_i} \exp\left(-\sum_{i=1}^{p} x_i^2 \lambda_i\right) dS^{p-1}(\mathbf{x}) \\
&= -\int x_i^2 \exp\left(-\sum_{i=1}^{p} x_i^2 \lambda_i\right) dS^{p-1}(\mathbf{x}) \\
&< 0.
\end{aligned}
$$

Therefore, $\mathcal{B}(\mathbf{\Lambda})$ is decreasing.

Likewise, the Hessian matrix of the Bingham normalizing constant has entries:

$$
\begin{aligned}
\frac{\partial^2}{\partial \lambda_i \partial \lambda_j} \mathcal{B}(\mathbf{\Lambda}) &= \int \frac{\partial^2}{\partial \lambda_i \partial \lambda_j} \exp\left(-\sum_{i=1}^{p} x_i^2 \lambda_i\right) dS^{p-1}(\mathbf{x}) \\
&= \int x_i^2 x_j^2 \exp\left(-\sum_{i=1}^{p} x_i^2 \lambda_i\right) dS^{p-1}(\mathbf{x}) \\
&= \mathbb{E}\left[X_i^2 X_j^2\right] \\
&= \operatorname{cov}\left(X_i^2,\, X_j^2\right),
\end{aligned}
$$

where X_i is the i^{th} element of the random vector $\boldsymbol{x} \sim \text{Bingham}(\mathbf{\Lambda})$. The Hessian matrix of $\mathcal{B}(\mathbf{\Lambda})$ is a covariance matrix, so it is positive semi-definite. Therefore, $\mathcal{B}(\mathbf{\Lambda})$ is convex. □

Lemma 1. $\mathcal{B}(\mathbf{\Lambda}) > 2^{-1}\pi^{-p}\mathcal{B}_c(\mathbf{\Lambda})$, *where* $\mathcal{B}$ *is the normalizing constant of the Bingham distribution, and* $\mathcal{B}_c$ *is the normalizing constant of the complex Bingham distribution with eigenvalues* $\mathbf{\Lambda}$.

Proof. The change of variables from $\mathbf{x} \in \mathcal{S}^{p-1}$, a vector on the unit sphere, to $\mathbf{s} \in \Delta^{p-1}$, a vector on the unit simplex, is given by the transformation $s_i = x_i^2$ and has Jacobian $\prod_{i=1}^{p} s_i^{-1/2}$. Thus, the Bingham constant can be rewritten as:

$$
\mathcal{B}(\Lambda) = \int_{\Delta^{p-1}} \exp\left(-\sum_{i=1}^{p} \lambda_i s_i\right) \prod_{i=1}^{p} s_i^{-1/2} ds_1, \ldots, ds_{k-1}. \tag{8.23}
$$

The term $\prod_{i=1}^{p} s_i^{-1/2} > 1$ because $0 < s_i < 1$. Therefore, this integral is strictly greater than:

$$\int_{\Delta^{p-1}} \exp\left(-\sum_{i=1}^{p} \lambda_i s_i\right) ds_1, \ldots, ds_{k-1}. \tag{8.24}$$

This integral is also obtained from the complex Bingham constant by transformation from the complex unit sphere to the unit simplex and polar coordinates (noting the difference in sign convention) [82]. Specifically, the integral in Equation 8.24 is equal to $2^{-1}\pi^{-p}\mathcal{B}_c(\Lambda)$. □

The complex Bingham distribution normalizing constant is a closed form expression which, under the sign convention we use, is [82]:

$$\mathcal{B}_c(\mathbf{\Lambda}) = 2\pi^p \sum_{i=1}^{p} \left(e^{-\lambda_i} \prod_{j \neq i} (\lambda_j - \lambda_i)^{-1} \right). \tag{8.25}$$

Lemma 2. *For any $p \in \mathbb{N}$, there is a number a such that the p-dimensional Bingham function is bounded below by $ag(\mathbf{\Lambda})$, where $g(\mathbf{\Lambda}) = e^{-\lambda_{i^\star}} \prod_{j=1}^{p} (\lambda_j - \lambda_{i^\star} + 1)^{-1}$, $\mathbf{\Lambda}$ is a diagonal matrix of eigenvalues $\lambda_i > 0$, and $i^\star = \arg\min_i \lambda_i$. In other words,*

$$\mathcal{B}_c(\mathbf{\Lambda}) > ag(\mathbf{\Lambda}).$$

Proof. The ratio

$$\frac{\mathcal{B}_c(\mathbf{\Lambda})}{g(\mathbf{\Lambda})} = 2\pi^p \sum_{i=1}^{p} \left(e^{-(\lambda_i - \lambda_{i^\star})} (\lambda_i - \lambda_{i^\star} + 1) \prod_{j \neq i} \frac{\lambda_j - \lambda_{i^\star} + 1}{\lambda_j - \lambda_i} \right)$$

is continuous and positive because $\mathcal{B}_c$ and g are both continuous and positive. The limit of each term for $i \neq i^\star$ in the summation is:

$$\lim_{\lambda_i \to \infty} e^{-(\lambda_i - \lambda_{i^\star})} (\lambda_i - \lambda_{i^\star} + 1) \prod_{j \neq i} \frac{\lambda_j - \lambda_{i^\star} + 1}{\lambda_j - \lambda_i} = 0.$$

If any eigenvalues are equal to each other, their terms combine by l'Hôpital's rule on $(e^{-\lambda_i} - e^{-\lambda_j})/(\lambda_j - \lambda_i)$, and the limit of the terms as the variables jointly approach infinity remains 0. The limit of the ratio as all of the eigenvalues go to infinity is:

$$\lim_{\lambda_{i^\star} \to \infty} \frac{\mathcal{B}_c(\mathbf{\Lambda})}{g(\mathbf{\Lambda})} \geq 2\pi^p.$$

Let ϵ be a number such that $0 < \epsilon < 2\pi^p$. Then there is a number b such that $\mathcal{B}_c(\mathbf{\Lambda})/g(\mathbf{\Lambda}) > 2\pi^p - \epsilon$ if all $\lambda_i > b$. By the extreme value theorem, $\mathcal{B}_c(\mathbf{\Lambda})/g(\mathbf{\Lambda})$ has a minimum value d in the p-dimensional interval $[0, b]^p$. Since $\mathcal{B}_c$ and g are both positive, $d > 0$. Let $a = \min(d,\, 2\pi^p - \epsilon)$. Then:

$$\frac{\mathcal{B}_c(\mathbf{\Lambda})}{a g(\mathbf{\Lambda})} > 1.$$

□

Lemma 3. *The Bingham normalizing constant of a positive definite matrix* $\mathbf{A}$ *is bounded by:*

$$\mathcal{B}(\mathbf{A}) < \frac{1}{2}\pi^{-p/2}\Gamma(p/2).$$

Proof. Since $\mathbf{A}$ is positive definite, $\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x} > 0$ and:

$$\begin{aligned} \mathcal{B}(\mathbf{A}) &= \int \exp\left(-\mathbf{x}^{\mathsf{T}}\mathbf{A}\mathbf{x}\right)\, dS^{p-1}(\mathbf{x}) \\ &< \int e^0\, dS^{p-1}(\mathbf{x}) \\ &= \frac{1}{2}\pi^{-p/2}\Gamma(p/2). \end{aligned} \tag{8.26}$$

□

Proof of Theorem 1. We demonstrate that Equation 8.4 is a convergent integral.

Case 1: $n > 0$.

Suppose $\mathbf{V}$ is a $p \times p$ positive definite matrix with $\mathrm{tr}(V) > n$. For any fixed $p \times p$ matrix $\mathbf{A}$ with eigenvalues Λ, let $i^\star = \min_i \lambda_i$. Let $\mathcal{B}_c(\mathbf{A})$ be the complex Bingham normalizing constant function. By the inequalities of Lemmas 1 and 2,

$$\int_{\mathbf{A}>0} [\mathcal{B}(\mathbf{A})]^{-n} \,\mathrm{etr}(-\mathbf{VA}) d\mathbf{A} < a^n \int_{\mathbf{A}>0} e^{n\lambda_{i^\star}} \left(\prod_{j=1}^{p} (\lambda_j - \lambda_{i^\star} + 1)^n \right) \mathrm{etr}(-\mathbf{VA}) d\mathbf{A}.$$

By the eigenvalue decomposition change of variables, the right-hand side is equal to:

$$a^n \int_{\mathbf{A}>0} e^{n\lambda_{i^\star}} \,\mathrm{etr}(-\mathbf{VQ\Lambda Q}^\mathsf{T}) \left(\prod_{j=1}^{p} (\lambda_j - \lambda_{i^\star} + 1)^n \right) \prod_{j<i} |\lambda_i - \lambda_j| d\mathbf{Q} \prod d\lambda_i,$$

which is less than:

$$a^n \int_{\mathbf{A}>0} e^{n\lambda_{i^\star}} \,\mathrm{etr}(-\mathbf{VQ\Lambda Q}^\mathsf{T}) \left(\prod_{j=1}^{p} (\lambda_j - \lambda_{i^\star} + 1)^{n+p-1} \right) d\mathbf{Q} \prod d\lambda_i$$

because $|\lambda_i - \lambda_j| \leq \max(\lambda_i, \lambda_j) - \lambda_{i^\star} < (\lambda_i - \lambda_{i^\star} + 1)(\lambda_j - \lambda_{i^\star} + 1)$, and each λ_i occurs in $p-1$ such comparisons. Rearranging terms, the above expression is equal to:

$$a^n \int_{\mathbf{A}>0} \exp\left(-(\mathrm{tr}(\mathbf{V}) - n)\lambda_{i^\star} - \sum_{j \neq i^\star} \left(\mathbf{q}_j^\mathsf{T} \mathbf{V} \mathbf{q}_j\right) (\lambda_j - \lambda_{i^\star}) \right) \left(\prod_{j=1}^{p} (\lambda_j - \lambda_{i^\star} + 1)^{n+p-1} \right) d\mathbf{Q} \prod d\lambda_i. \quad (8.27)$$

The terms for each eigenvalue other than the smallest can be expressed as an integral:

$$\begin{aligned}
&\int_{\lambda_{i^\star}}^{\infty} (\lambda_j - \lambda_{i^\star} + 1)^{n+p-1} \exp\left[- \left(\mathbf{q}_j^\mathsf{T} \mathbf{V} \mathbf{q}_j\right) (\lambda_j - \lambda_{i^\star}) \right] d\lambda_j \\
&\quad = \exp\left(\mathbf{q}_j^\mathsf{T} \mathbf{V} \mathbf{q}_j\right) \int_{1}^{\infty} \lambda_j^{n+p-1} \exp\left[- \left(\mathbf{q}_j^\mathsf{T} \mathbf{V} \mathbf{q}_j\right) \lambda_j \right] d\lambda_j \\
&\quad < \exp\left(\mathbf{q}_j^\mathsf{T} \mathbf{V} \mathbf{q}_j\right) \int_{0}^{\infty} \lambda_j^{n+p-1} \exp\left[- \left(\mathbf{q}_j^\mathsf{T} \mathbf{V} \mathbf{q}_j\right) \lambda_j \right] d\lambda_j \\
&\quad = \Gamma(n+p) \left(\mathbf{q}_j^\mathsf{T} \mathbf{V} \mathbf{q}_j\right)^{-(n+p)} \exp\left(\mathbf{q}_j^\mathsf{T} \mathbf{V} \mathbf{q}_j\right). \qquad (8.28)
\end{aligned}$$

Combining this back into Equation 8.27 shows that the Bingham constant is less than:

$$\begin{aligned}
&a^n \int \int_0^\infty \exp\left[-(\mathrm{tr}(\mathbf{V})-n)\lambda_{i^\star}\right] d\lambda_{i^\star} \prod_{j\neq i^\star} \left[\Gamma(n+p)\left(\mathbf{q}_j^\mathsf{T}\mathbf{V}\mathbf{q}_j\right)^{-(n+p)} \exp\left(\mathbf{q}_j^\mathsf{T}\mathbf{V}\mathbf{q}_j\right)\right] d\mathbf{Q}\\
&\quad= a^n\Gamma(n+p)^{p-1}(\mathrm{tr}(\mathbf{V})-n)^{-1} \int \left(\mathbf{q}_j^\mathsf{T}\mathbf{V}\mathbf{q}_j\right)^{-(n+p)(p-1)} \exp\left(\mathbf{q}_j^\mathsf{T}\mathbf{V}\mathbf{q}_j\right) d\mathbf{Q}\\
&\quad= a^n\Gamma(n+p)^{p-1}(\mathrm{tr}(\mathbf{V})-n)^{-1} \int \eta^{-(n+p)(p-1)} e^{\eta} d\mathbf{Q}\\
&\quad= a^n\Gamma(n+p)^{p-1}(\mathrm{tr}(\mathbf{V})-n)^{-1} \eta^{-(n+p)(p-1)} e^{\eta}, \qquad (8.29)
\end{aligned}$$

where η is the maximum eigenvalue of $\mathbf{V}$.

Case 2: $n \leq 0$. By Lemma 3,

$$\int_{\mathbf{A}>0} \mathrm{etr}(-\mathbf{V}\mathbf{A})[\mathcal{B}(\mathbf{A})]^{-n} d\mathbf{A} < 2^n \pi^{pn/2} \Gamma(p/2)^{-n} \int_{\mathbf{A}>0} \mathrm{etr}(-\mathbf{V}\mathbf{A}) d\mathbf{A}. \qquad (8.30)$$

We evaluate the integral $\int_{\mathbf{A}>0} \mathrm{etr}(-\mathbf{V}\mathbf{A})d\mathbf{A}$ using two changes of variables [93] related to the triangular matrix transformation method for deriving the Wishart normalizing constant [94].

The first change of variables sets $\mathbf{U} = \mathbf{L}^\mathsf{T}\mathbf{A}\mathbf{L}$, where $\mathbf{L}$ is a lower triangular matrix defined by the Cholesky decomposition $\mathbf{L}\mathbf{L}^\mathsf{T} = \mathbf{V}$. Thus, $\mathrm{tr}(\mathbf{V}\mathbf{A}) = \mathrm{tr}(\mathbf{L}\mathbf{L}^\mathsf{T}\mathbf{A}) = \mathrm{tr}(\mathbf{L}^\mathsf{T}\mathbf{A}\mathbf{L}) = \mathrm{tr}(\mathbf{U})$. The transformation $\mathbf{A} \to \mathbf{U}$ has Jacobian $\det(\mathbf{L})^{p+1} = \det(\mathbf{V})^{-\frac{p+1}{2}}$.

$$\begin{aligned}
\int_{\mathbf{A}>0} \mathrm{etr}(-\mathbf{V}\mathbf{A})d\mathbf{A} &= \int_{\mathbf{U}>0} \mathrm{etr}(-\mathbf{U}) \det(\mathbf{V})^{-\frac{p+1}{2}} d\mathbf{U}\\
&= \det(\mathbf{V})^{-\frac{p+1}{2}} \int_{\mathbf{U}>0} \mathrm{etr}(-\mathbf{U}) d\mathbf{U}
\end{aligned} \qquad (8.31)$$

The second change of variables sets $\mathbf{X}\mathbf{X}^\mathsf{T} = \mathbf{U}$ by Cholesky decomposition. In general, a matrix $\mathbf{U} = \mathbf{X}\mathbf{X}^\mathsf{T}$ is positive definite if and only if all diagonal elements $x_{ii} > 0$. Since $\mathbf{X}$ is lower triangular, $x_{ij} = 0$ for all $i < j$, and the value of x_{ij} ranges from $-\infty$ to ∞ for $i > j$.

The transformation $\mathbf{U} \to \mathbf{X}$ has Jacobian $2^p \prod_{i=1}^p x_{ii}^{p-i+1}$.

$$\begin{aligned}
\int_{\mathbf{U}>0} \operatorname{etr}(-\mathbf{U}) d\mathbf{U} &= 2^p \int_{x_{ii}>0} \left(\operatorname{etr}(-\mathbf{X}\mathbf{X}^\mathsf{T}) \prod_{i=1}^{p} x_{ii}^{p-i+1} \right) \prod_{i\geq j} dx_{ij} \\
&= 2^p \int_{x_{ii}>0} \left(e^{-\sum_{i\geq j} x_{ij}^2} \prod_{i=1}^{p} x_{ii}^{p-i+1} \right) \prod_{i\geq j} dx_{ij} \\
&= \left[\prod_{i>j} \int_{-\infty}^{\infty} e^{-x_{ij}^2} dx_{ij} \right] \prod_{i=1}^{p} 2 \int_{0}^{\infty} e^{-x_{ii}^2} x_{ii}^{p-i+1} dx_{ii} \\
&= \left[\Gamma\left(\frac{1}{2}\right) \right]^{\frac{p(p-1)}{2}} \prod_{i=1}^{p} \Gamma\left(\frac{p+1}{2} + \frac{1-i}{2} \right) \\
&= \Gamma_p\left(\frac{p+1}{2}\right) && (8.32)
\end{aligned}$$

where $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt = 2 \int_0^\infty x^{2z-1} e^{-x^2} dx$.

Combining Equations 8.30, 8.31, and 8.32 gives:

$$\mathcal{C}(\mathbf{V}, n) < 2^n \pi^{\frac{pn}{2}} \Gamma\left(\frac{p}{2}\right)^{-n} \Gamma_p\left(\frac{p+1}{2}\right) \det(\mathbf{V})^{-\frac{p+1}{2}}. \qquad (8.33)$$

Therefore, $\mathcal{C}(\mathbf{V}, n)$ is finite. □

8.8 Appendix: Bingham Algorithms

We review known algorithms that are essential for working with the Bingham and Bingham-Conjugate distributions. We include these algorithms to fix minor errors and to increase the clarity and level of detail relative to their original presentations.

8.8.1 BinghamEigendecomposition Algorithm

The behavior of the Bingham distribution is determined by the eigenvalues of its parameter matrix $\mathbf{A}$, so it is standard to perform eigenvalue decomposition on $-\mathbf{A}$. Furthermore, the distribution is invariant to adding a constant to each eigenvalue, which only changes the normalizing constant. It is common for the sake of convenience to assume without loss of generality that the eigenvalues are sorted from largest to smallest and normalized with the smallest equal to zero. In practice it is only necessary to normalize the eigenvalues and swap the smallest eigenvalue with the last eigenvalue.[2]

BinghamEigendecomposition takes a symmetric $p \times p$ parameter matrix $\mathbf{A}$ and returns an array $\boldsymbol{\lambda}$ of normalized eigenvalues of $-\mathbf{A}$, a corresponding orthogonal matrix $\mathbf{Q}$ of eigenvectors, and the smallest eigenvalue $\lambda^\star$ of $-\mathbf{A}$.

1. Apply a standard eigendecomposition routine to the symmetric matrix $-\mathbf{A}$ to find $\tilde{\boldsymbol{\Lambda}}$ and $\tilde{\mathbf{Q}}$ such that: $-\mathbf{A} = \tilde{\mathbf{Q}}\tilde{\boldsymbol{\Lambda}}\tilde{\mathbf{Q}}^{-1}$. Let the array $\tilde{\boldsymbol{\lambda}}$ be the diagonal of $\tilde{\boldsymbol{\Lambda}}$.

2. Let $i^\star = \arg\min_i \left(\tilde{\lambda}_i\right)$ and $\tilde{\lambda}^\star = \tilde{\lambda}_{i^\star}$.

3. Define the length-p array $\boldsymbol{\lambda}$ by the elements: $\lambda_p = 0$, $\lambda_{i^\star} = \tilde{\lambda}_p - \tilde{\lambda}^\star$, and for all $i \notin \{p,\, i^\star\}$: $\lambda_i = \tilde{\lambda}_i - \tilde{\lambda}^\star$.

4. Define the $p \times p$ matrix $\mathbf{Q}$ by the columns: $\mathbf{Q}_p = \tilde{\mathbf{Q}}_{i^\star}$, $\mathbf{Q}_{i^\star} = \tilde{\mathbf{Q}}_p$, and for all $i \notin \{p,\, i^\star\}$: $\mathbf{Q}_i = \tilde{\mathbf{Q}}_i$.

5. Thus, $\lambda_p = 0$, $\lambda_i \geq 0$, and the columns of $\mathbf{Q}$ are ordered to maintain the correspondence between eigenvectors and eigenvalues.

6. Return the triple $\left(\boldsymbol{\lambda}, \mathbf{Q}, \tilde{\lambda}^\star\right)$.

8.8.2 BinghamSampler Algorithm

Sampling for the Bingham distribution uses the technique of Gibbs sampling with auxiliary variables [92] as applied to the general Liouville family of distributions [83]. In light of non-trivial omissions and an error, we describe the sampling algorithm in detail.

BinghamSampler [83] takes a symmetric $p \times p$ parameter matrix $\mathbf{A}$ and produces a sequence of samples from the associated Bingham distribution with density $f(\mathbf{x}) \propto$

[2] It is possible to avoid the swap and normalization step entirely by storing the index and value of the smallest eigenvalue and adapting the formulas in the algorithms, but this would make the formulas harder to follow.

$\exp(-\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}), \mathbf{x} \in S^{p-1}$. Because this is a Gibbs sampler, the samples are autocorrelated and should be thinned, though this autocorrelation decays quickly in practice [83]. Note that this algorithm refers to the **BinghamEigendecomposition** algorithm described in Section 8.8.

1. Let $\left(\boldsymbol{\lambda}, \mathbf{Q}, \tilde{\lambda}^\star\right) = \mathbf{BinghamEigendecomposition}(\mathbf{A})$.

2. Initialize the array $\mathbf{s}$ of length $p-1$ such that $s_i = 0$.

3. Gibbs sampling loop:

   (a) Sample v from the $\mathcal{U}\left(0, \exp\left(-\sum_{i=1}^{p-1} \lambda_i s_i\right)\right)$ uniform distribution.

   (b) Sample w from $\mathcal{U}\left(0, \left(1 - \sum_{i=1}^{p-1} s_i\right)^{-1/2}\right)$.

   (c) For i from 1 to $p-1$:

      i. Let $t = 1 - \sum_{1 \le j \le p-1, \, j \ne i} s_j$.

      ii. Let $c = \max\left(0, \, t - w^{-2}\right)$.

      iii. Let $d = \min\left(\lambda_i^{-1}\left(-\ln v - \sum_{1 \le j \le p-1, \, j \ne i} \lambda_j s_j\right), \, t\right)$.

      iv. Sample u from $\mathcal{U}\left(c^{1/2}, \, d^{1/2}\right)$.[3]

      v. Set $s_i = u^2$.

   (d) Set $s_p = 1 - \sum_{i=1}^{p-1} s_i$.

   (e) Set $\mathbf{x} = \mathbf{Q}\mathbf{s}^{1/2}$ (matrix product and element-wise exponentiation).

   (f) Yield the sample $\mathbf{x}$.

8.8.3 BinghamConstant Algorithm

BinghamConstant [86] takes a symmetric $p \times p$ parameter matrix $\mathbf{A}$ and returns an estimate $\hat{c}$ of the normalizing constant $\mathcal{B}(\mathbf{A})$ such that the Bingham density $f(\mathbf{x}) \approx \hat{c}^{-1} \exp(\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x}), \mathbf{x} \in S^{p-1}$.

[3] Note that the bounds were mistakenly reported as: $(c^{1/\alpha_1}, \, d^{1/\alpha_1})$ [83]. The bounds from the change of variables should have instead read: $(c^{\alpha_1}, \, d^{\alpha_1})$.

1. Let $\left(\boldsymbol{\lambda}, \mathbf{Q}, \tilde{\lambda}^\star\right) = \textbf{BinghamEigendecomposition}(\mathbf{A})$.

2. Let $K_\theta^{(1)}(t) = \sum_{i=1}^p \frac{1}{2(\lambda_i - t)}$,
   $K_\theta^{(2)}(t) = \sum_{i=1}^p \frac{1}{2(\lambda_i - t)^2}$,
   $K_\theta^{(3)}(t) = \sum_{i=1}^p \frac{1}{(\lambda_i - t)^3}$,
   and $K_\theta^{(4)}(t) = \sum_{i=1}^p \frac{3}{(\lambda_i - t)^4}$.

3. Apply a standard root-finding routine (such as Newton's method) to find the unique root $\hat{t}$ in the interval $(-p/2, -1/2)$ of the function $K_\theta^{(1)}(t) - 1$ with derivative $K_\theta^{(2)}(t)$ and initial value $t_0 = -1/2$.

4. Let $\hat{K}_\theta^{(2)} = K_\theta^{(2)}\left(\hat{t}\right)$, $\hat{K}_\theta^{(3)} = K_\theta^{(3)}\left(\hat{t}\right)$, and $\hat{K}_\theta^{(4)} = K_\theta^{(4)}\left(\hat{t}\right)$.

5. Let $T = \frac{\hat{K}_\theta^{(4)}}{8\left(\hat{K}_\theta^{(2)}\right)^2} - \frac{5\left(\hat{K}_\theta^{(3)}\right)^2}{24\left(\hat{K}_\theta^{(2)}\right)^3}$.

6. Return $\hat{c} = \left(2\pi^{p-1}\right)^{1/2}\left(\hat{K}_\theta^{(2)}\prod_{i=1}^p\left(\lambda_i - \hat{t}\right)\right)^{-1/2} e^{T - \hat{t} - \hat{\lambda}^\star}$.

Note that the Bingham distribution with $p = 3$ and $\lambda_1 = \lambda_2 = \lambda_3 = 0$ is simply the uniform distribution on the ordinary sphere. In this case, the true value of the normalizing constant is 4π, and this algorithm estimates the constant as 3.997π.

Conclusion

Chapter 9

Conclusion

We set out to improve parallel optimization by beginning with a parallel perspective and treating coordination and communication as primary concerns. In Section 9.1 we review how each of the contributions of this work contribute to this goal. In Section 9.2 we look forward to how future work can further develop parallel optimization algorithms based on these contributions.

9.1 Contributions

This work is organized in three parts. The first part describes a platform for conveniently developing parallel optimization algorithms with communication and centralized coordination explicitly called out. The second part explores various ways to improve the performance of Particle Swarm Optimization in a parallel computational environment by reducing communication and unnecessary coordination. The third part reconsiders how the work of optimization should be decomposed for parallel computation and develops a statistical model for coordinating search directions for exploiting separability.

The first part describes a MapReduce framework and changes to the MapReduce model that make it convenient to develop flexible parallel optimization algorithms. Chapter 2 decomposes the operations of the standard PSO algorithm, reformulates them as MapReduce operations, and motivates the need for iterative MapReduce. Note that PSO in MapReduce produces numerically identical results to a standard serial PSO implementation if given the same random seed. Chapter 3 presents the Mrs MapReduce framework. In practice, the

MapReduce-based optimization algorithms have proven easy to develop, adapt, and use. The programming model's explicitness in handling the state of data and communication, along with the framework's random number facilities, make it easy to ensure that numerically identical computation is performed in both standard serial PSO and in various equivalent parallel formulations. For example, the complex interactions in Speculative Evaluation PSO (in Chapter 5) require careful debugging that probably may have proved incurably difficult in a typical low-level implementation. Chapter 4 describes an improved iterative MapReduce programming model with associated performance improvements for programs such as optimization algorithms. For example, the asynchronous model allows a MapReduce-based optimization algorithm to support asynchronous iteration with much improved performance, while requiring little or no change to the map and reduce functions. The first part of the work sets a computational framework in which to explore parallel optimization.

The second part uses the framework to explore a variety of changes to improve the parallel performance of Particle Swarm Optimization by reducing communication and unnecessary coordination. Chapter 5 introduces Speculative Evaluation PSO, which performs iterations twice as quickly as standard parallel PSO, at the cost of additional processors. It produces numerically identical results as standard PSO and gives insight into the behavior of iteration in PSO. Chapter 6 describes the Apiary topology for using subswarms in parallel PSO without requiring centralized coordination. As it achieves this through the topology mechanism of PSO, it is compatible with asynchronous PSO. Chapter 7 reviews all of these, and many other, approaches to parallel PSO, describing their demands on communication and interaction between tasks and clarifying the types of objective functions and computational environments in which each approach is well suited. The second part of the work sheds light on the fundamental concerns of parallel optimization.

The third part develops a mathematical model for performing inference on search directions for exploiting loose separability of an objective function, without requiring centralized coordination or receiving all sampled points from other processors. Chapter 8 introduces the BinghamConjugate distribution with a proof that its constant of integration is finite and an efficient sampling algorithm. Furthermore, it explores how this distribution can be practically applied and incorporated into a mathematical model that can detect promising directions in non-overlapping portions of the hypersphere. The third part of the work lays a foundation for exploiting separability which can be further developed into a wide variety of parallel optimization algorithms founded on sound mathematical principles.

9.2 Future Work

This work makes it possible to develop a variety of algorithms that perform decentralized coordination while exploiting loose separability. This opens up many areas for additional research, including:

- developing line search algorithm to use on each processor;
- pruning past observed directions to save space and accommodate changes in the state of other processors;
- developing tests for detecting separability;
- determining which evaluated points to share with other processors;
- using additional processors beyond the number of dimensions; and
- enriching the statistical model (note that the normalizing constant algorithm [86] discussed in Appendix 8.8 may make certain types of mixture models practical).

Each of these issues, and many others not listed here, are fascinating starting points for further work.

We also note that the area of function optimization needs new theoretical tools for describing and classifying objective functions. Functions are described in terms of whether they are convex or non-convex, unimodal or multimodal, continuous or discontinuous, and separable or non-separable. Although functions can be described separable or non-separable, there may be many different types of local or weak separability. While the existing classifications are useful, a more complete set of useful classifications would encourage the development of algorithms designed for specific classes of functions.

References

[1] A. Cauchy. Méthode générale pour la résolution des systemes déquations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536, 1847.

[2] M. Powell. An Efficient Method for Finding the Minimum of a Function of Several Variables Without Calculating Derivatives. *The computer journal*, 7(2), 1964.

[3] J. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4), 1965.

[4] J. Matyas. Random Optimization. *Automation and Remote Control*, 26(2), 1965.

[5] S. Kirkpatrick, C. Gelatt Jr, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), 1983.

[6] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21, 1953.

[7] I. Rechenberg. *Evolutionsstrategie—Optimierung technisher Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, 1973.

[8] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A Survey of Evolution Strategies. In *Proc. International Conference on Genetic Algorithms*. 1991.

[9] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[10] K. De Jong. *Analysis of the Behavior of a Class of Genetic Adaptive Systems*. Ph.D. thesis, University of Michigan, 1975.

[11] N. Hansen and A. Ostermeier. Adapting Arbitrary Normal Mutation Distributions in Evolution Strategies: The Covariance Matrix Adaptation. In *Proc. IEEE International Conference on Evolutionary Computation*. 1996.

[12] R. Storn and K. Price. Differential Evolution—a Simple and Efficient Heuristic for Global Optimization Over Continuous Spaces. *Journal of global optimization*, 11(4):341, 1997.

[13] P. Bosman and D. Thierens. Expanding from Discrete to Continuous Estimation of Distribution Algorithms: the IDEA. In *Proc. Parallel Problem Solving from Nature*. 2000.

[14] P. Larrañaga, R. Etxeberria, J. Lozano, and J. Peña. Optimization in Continuous Domains by Learning and Simulation of Gaussian Networks. In *Proc. Optimization by Building and Using Probabilistic Models Workshop at the Genetic and Evolutionary Computation Conference*. 2000.

[15] J. Kennedy and R. C. Eberhart. Particle Swarm Optimization. In *Proc. International Conference on Neural Networks IV*. 1995.

[16] W. Zhao, H. Ma, and Q. He. Parallel k-means Clustering Based on MapReduce. *Cloud Computing*, 2009.

[17] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *Proc. Advances in Neural Information Processing Systems*. 2007.

[18] C. Jin, C. Vecchiola, and R. Buyya. MRPGA: an Extension of MapReduce for Parallelizing Genetic Algorithms. In *IEEE International Conference on eScience*. 2008.

[19] J. Liang and P. Suganthan. Dynamic Multi-Swarm Particle Swarm Optimizer. In *Proc. IEEE Swarm Intelligence Symposium*. 2005.

[20] J. Romero and C. Cotta. Optimization by Island-Structured Decentralized Particle Swarms. In *Proc. Fuzzy Days: Computational Intelligence, Theory and Applications*. 2005.

[21] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1), 1997.

[22] A. McNabb, C. Monson, and K. Seppi. Parallel PSO using MapReduce. In *Proc. IEEE Congress on Evolutionary Computation*. 2007.

[23] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. Operating System Design and Implementation*. 2004.

[24] M. Clerc and J. Kennedy. The Particle Swarm—Explosion, Stability, and Convergence in a Multidimensional Complex Space. *IEEE Transactions on Evolutionary Computation*, 6(1), 2002.

[25] J. Schutte, J. Reinbolt, B. Fregly, R. Haftka, and A. George. Parallel Global Optimization with the Particle Swarm Algorithm. *International Journal for Numerical Methods in Engineering*, 61(13), 2004.

[26] B.-I. Koh, A. George, R. Haftka, and B. Fregly. Parallel Asynchronous Particle Swarm Optimization. *International Journal of Numerical Methods in Engineering*, 67, 2006.

[27] G. Venter and J. Sobieszczanski-Sobieski. A Parallel Particle Swarm Optimization Algorithm Accelerated by Asynchronous Evaluations. In *Proc. World Congress on Structural and Multidisciplinary Optimization*. 2005.

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

[29] S. Mostaghim, J. Branke, and H. Schmeck. Multi-Objective Particle Swarm Optimization on Computer Grids. Technical Report 502, AIFB Institute, Karlsruhe, Germany, 2006.

[30] N. Jin and Y. 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), 2005.

[31] K. E. Parsopoulos, D. K. Tasoulis, and M. N. Vrahatis. Multiobjective Optimization Using Parallel Vector Evaluated Particle Swarm Optimization. In *Proc. IASTED International Conference on Artificial Intelligence and Applications*. IASTED/ACTA Press, Calgary, AB, Canada, 2004.

[32] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, Harlow, England, second edition, 2003.

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

[34] J. Kennedy. Small Worlds and Mega-Minds: Effects of Neighborhood Topology on Particle Swarm Performance. In *Proc. IEEE Congress on Evolutionary Computation*, volume 3. 1999.

[35] A. McNabb, J. Lund, and K. Seppi. Mrs: MapReduce for Scientific Computing in Python. In *Proc. Python for High Performance and Scientific Computing*. 2012.

[36] A. McNabb, C. Monson, and K. Seppi. MRPSO: MapReduce Particle Swarm Optimization. In *Proc. Conference on Genetic and Evolutionary Computation*. 2007.

[37] Y. Bu, B. Howe, M. Balazinska, and M. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *Proc. VLDB Endowment*, 3(1-2), 2010.

[38] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox. Twister: a Runtime for Iterative MapReduce. In *Proc: High Performance Distributed Computing*. 2010.

[39] D. Bratton and J. Kennedy. Defining a Standard for Particle Swarm Optimization. In *Proc. IEEE Swarm Intelligence Symposium*. 2007.

[40] A. McNabb and K. Seppi. The Apiary Topology: Emergent Behavior in Communities of Particle Swarms. In *Proc. Parallel Problem Solving from Nature*. 2012.

[41] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proc. USENIX Conference on Hot Topics in Cloud Computing*. 2010.

[42] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.

[43] D. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a Universal Execution Engine for Distributed Data-Flow Computing. In *Proc. Network Systems Design and Implementation*. 2011.

[44] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM Sigplan Notices*. 2010.

[45] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5), 2003.

[46] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. IEEE Symposium on Mass Storage Systems and Technologies*. 2010.

[47] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: a Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proc. European Conference on Computer Systems*. 2010.

[48] E. Elnikety, T. Elsayed, and H. Ramadan. iHadoop: Asynchronous Iterations for MapReduce. In *Proc. IEEE Cloud Computing Technology and Science*. 2011.

[49] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: a Distributed Computing Framework for Iterative Computation. In *IEEE International Parallel and Distributed Processing Symposium Workshops*. 2011.

[50] E. Morenoff and J. McLean. Application of level changing to a multilevel storage organization. *Communications of the ACM*, 10(3):149, 1967.

[51] I. Scriven, D. Ireland, A. Lewis, S. Mostaghim, and J. Branke. Asynchronous Multiple Objective Particle Swarm Optimisation in Unreliable Distributed Environments. In *Proc. IEEE Congress on Evolutionary Computation*. 2008.

[52] K. Tang, X. Li, P. Suganthan, Z. Yang, and T. Weise. Benchmark Functions for the CEC'2010 Special Session and Competition on Large Scale Global Optimization. Technical report, IEEE Congress on Evolutionary Computation, November, 2009.

[53] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1), 1977.

[54] D. Walker and E. Ringger. Model-based document clustering with a collapsed gibbs sampler. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2008.

[55] M. Meila and D. Heckerman. An Experimental Comparison of Model-Based Clustering Methods. *Machine Learning*, 2001.

[56] N. Kumar, S. Satoor, and I. Buck. Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA. In *Proc. High Performance Computing and Communications*. 2009.

[57] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient Large-Scale Distributed Training of Conditional Maximum Entropy Models. *Advances in Neural Information Processing Systems*, 2009.

[58] K. Lang. Newsweeder: Learning to filter netnews. In *Proc. International Conference on Machine Learning*. 1995.

[59] K. Ganchev and M. Dredze. Small statistical models by random feature mixing. In *Proc. Workshop on Mobile NLP at ACL*. 1998.

[60] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *Proc. International Conference on Machine Learning.* 2009.

[61] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3, 2003.

[62] M. Gardner, A. McNabb, and K. Seppi. Speculative Evaluation in Particle Swarm Optimization. In *Proc. Parallel Problem Solving from Nature.* 2010.

[63] A. McNabb, M. Gardner, and K. Seppi. An Exploration of Topologies and Communication in Large Particle Swarms. In *Proc. IEEE Congress on Evolutionary Computation*, pp. 712–719. 2009.

[64] R. Mendes. *Population Topologies and Their Influence in Particle Swarm Performance.* Ph.D. thesis, Universidade do Minho, Guimaraes, Portugal, 2004.

[65] D. Sammataro and A. Avitabile. *Beekeeper's Handbook, 3rd edition.* Cornell University Press, 1998.

[66] I. Sample. Bees Translate Dances of Foreign Species. The Guardian, 2008.

[67] S.-C. Chu and J.-S. Pan. Intelligent Parallel Particle Swarm Optimization Algorithms. *Parallel Evolutionary Computations*, 2006.

[68] Y. Lorion, T. Bogon, I. Timm, and O. Drobnik. An Agent Based Parallel Particle Swarm Optimization—APPSO. In *Swarm Intelligence Symposium.* 2009.

[69] H. Wang and F. Qian. An Improved Particle Swarm Optimizer with Shuffled Sub-Swarms and its Application in Soft-Sensor of Gasoline Endpoint. In *Proc. International Conference on Intelligent Systems and Knowledge Engineering.* 2007.

[70] M. Dwass. Modified Randomization Tests for Nonparametric Hypotheses. *The Annals of Mathematical Statistics*, 28(1):181, 1957.

[71] A. McNabb and K. Seppi. Serial PSO results are irrelevant in a multi-core parallel world. In *Proc. IEEE Congress on Evolutionary Computation.* 2014.

[72] D. Gies and Y. Rahmat-Samii. Particle Swarm Optimization for Reconfigurable Phase-Differentiated Array Design. *Microwave and Optical Technology Letters*, 38(3), 2003.

[73] Y. Zhou and Y. Tan. GPU-based parallel particle swarm optimization. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on.* 2009.

[74] J. Li, X. Wang, R. He, and Z. Chi. An Efficient Fine-Grained Parallel Genetic Algorithm Based on GPU-Accelerated. In *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*. 2007.

[75] I. Scriven, A. Lewis, D. Ireland, and J. Lu. Decentralised Distributed Multiple Objective Particle Swarm Optimisation Using Peer to Peer Networks. In *Proc. IEEE Congress on Evolutionary Computation*. 2008.

[76] J. Jordan, S. Helwig, and R. Wanka. Social Interaction in Particle Swarm Optimization, the Ranked FIPS, and Adaptive Multi-Swarms. In *Proc. Conference on Genetic and Evolutionary Computation*. ACM, 2008.

[77] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar. An Asynchronous Parallel Stochastic Coordinate Descent Algorithm. In *Proc. International Conference on Machine Learning*. 2014.

[78] I. Loshchilov, M. Schoenauer, and M. Sebag. Adaptive Coordinate Descent. In *Proc. Genetic and Evolutionary Computation Conference*. 2011.

[79] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3), 1960.

[80] P. Jupp and K. Mardia. A Unified View of the Theory of Directional Statistics, 1975–1988. *International Statistical Review*, 57(3), 1989.

[81] C. Bingham. An Antipodally Symmetric Distribution on the Sphere. *The Annals of Statistics*, 2(6), 1974.

[82] J. T. Kent. The Complex Bingham Distribution and Shape Analysis. *Journal of the Royal Statistical Society, Series B*, 1994.

[83] A. Kume and S. G. Walker. Sampling from compositional and directional distributions. *Statistics and Computing*, 16(3), 2006.

[84] J. T. Kent, A. M. Ganeiber, and K. V. Mardia. A New Method to Simulate the Bingham and Related Distributions in Directional Data Analysis with Applications. Technical report, University of Leeds, 2013.

[85] S. G. Walker. Bayesian Estimation of the Bingham Distribution. *Brazilian Journal of Probability and Statistics*, 2013.

[86] A. Kume and A. T. A. Wood. Saddlepoint Approximations for the Bingham and Fisher–Bingham Normalising Constants. *Biometrika*, 92(2), 2005.

[87] G. Nuñez-Antonio and E. Gutiérrez-Peña. A Bayesian Analysis of Directional Data Using the Projected Normal Distribution. *Journal of Applied Statistics*, 32(10), 2005.

[88] J. T. Kent, P. D. Constable, and F. Er. Simulation for the Complex Bingham Distribution. *Statistics and Computing*, 14(1), 2004.

[89] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer, second edition, 2004.

[90] W. Smith and R. Hocking. Algorithm AS 53: Wishart Variate Generator. *Journal of the Royal Statistical Society, Series C*, 21(3), 1972.

[91] R. J. Muirhead. *Aspects of Multivariate Statistical Theory*. Wiley, 1982.

[92] P. Damien, J. Wakefield, and S. Walker. Gibbs Sampling for Bayesian Non-Conjugate and Hierarchical Models by Using Auxiliary Variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(2), 1999.

[93] L. Gårding. The Solution of Cauchy's Problem for Two Totally Hyperbolic Linear Differential Equations by Means of Riesz Integrals. *The Annals of Mathematics*, 48(4), 1947.

[94] A. C. Aitken. On the Wishart Distribution in Statistics. *Biometrika*, 36(1–2), 1949.