INFORMATION EXCHANGE AND CONFLICT RESOLUTION IN PARTICLE SWARM OPTIMIZATION VARIANTS

by
Stephyn G. W. Butcher

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland
April, 2018

Abstract

Single population, biologically-inspired algorithms such as Genetic Algorithm and Particle Swarm Optimization are effective tools for solving a variety of optimization problems. Like many such algorithms, however, they fall victim to the curse of dimensionality. Additionally, these algorithms often suffer from a phenomenon known as *hitchhiking* where improved solutions are not unequivocally better for all variables. Insofar as individuals within these populations are deemed to be competitive, one solution to both the curse of dimensionality and the problem of hitchhiking has been to introduce more cooperation. These multi-population algorithms cooperate by decomposing a problem into parts and assigning a population to each part.

Factored Evolutionary Algorithms (FEA) generalize this decomposition and cooperation to any evolutionary algorithm. A key element of FEA is a global solution that provides missing information to individual populations and coordinates them. This dissertation extends FEA to the distributed case by having individual populations maintain and coordinate local solutions that maintain consensus. This Distributed

ABSTRACT

FEA (DFEA) is demonstrated to perform well on a variety of problems and, sometimes, even if consensus is lost. However, DFEA fails to maintain the same semantics as FEA.

To address this issue, we develop an alternative framework to the "cooperation versus competition" dichotomy. In this framework, information flows are modeled as a blackboard architecture. Changes in the blackboard are modeled as merge operations that require conflict resolution between existing and candidate values. Conflict resolution is handled using Pareto efficiency, which avoids hitchhiking. We apply this framework to FEA and DFEA and develop revised DFEA, which performs identically to FEA.

We then apply our framework to a *single* population algorithm, Particle Swarm Optimization (PSO), to create Pareto Improving PSO (PI-PSO). We demonstrate that PI-PSO outperforms PSO and sometimes FEA-PSO, often with fewer individuals.

ABSTRACT

Finally, we extend our information based approach by implementing parallel, distributed versions of FEA and DFEA using the Actor model. The Actor model is based on message passing, which accords well with our information-centric framework. We use validation experiments to verify that we have successfully implemented the semantics of the serial versions of FEA and DFEA.

Primary Reader: John Sheppard

Secondary Readers: Scott Smith, Brian Haberman

Acknowledgments

At this juncture, I am not sure I have seen further but I have certainly seen different. A work of this magnitude is not possible without the help of many people and fortuitous circumstances. As we all well know, there are many fits and starts along the way and perhaps more so for me than for most.

My first round of thanks go to my advisor, John Sheppard. A dozen or so years ago now, John asked if I would be interested in writing a paper with him and thus started a multi-year adventure that continues to this day. I thank him for all he has done. Hopefully we have at least another dozen or so years of teaching and research in us!

I would like to thank Scott Smith who has also been with me on this lengthy journey as well. I imagine he is rather happy to see me on my way at this point.

I would like to thank Brian Haberman for being on my committee and for starting this research a decade ago.

I would like to thank the members of the Numerical Intelligent Systems Laboratory

(NISL). The cast of characters has changed over the years but all have been great lab mates. I am the latest to build on the research that has come out from or near NISL and so I thank those who went before me: Brian Haberman, Karthik Ganesan Pillai, and Nathan Fortier. And I would especially like to give a special shout out to my immediate predecessor in this line of research, Shane Strasser, for being a great friend and colleague over the years.

I would like to thank my parents: Judy & Stephen Stapleton and Bill & Jeannie Butcher. Thanks go to my siblings: Benjamen, Ali, Christopher, Kimberly and Laurie, as well. Thank you for the love and support over the years. And a special shout out to Ed Tennant who at one point just started calling me "Doctor" because he had more or less lost patience.

The work was made sweeter with the love and support of my husband, Michael Kingan, who has had to put up with both a Master's degree and then a Ph. D. This endeavor has now spanned about 15 years of the 17 we have been together. My final thanks go to our dog, Kieran, who has literally been by my side or at my feet the entire time: man's best friend indeed.

Dedication

In honor of Alan Mathison Turing, for his wondrous but sometimes infuriating machines.

For Michael John Kingan, because discovering how the world works has yet to remove the mystery, humor and love.

Contents

List of Tables

LIST OF TABLES

List of Figures

LIST OF FIGURES

List of Algorithms

Chapter 1

Introduction

There are many complex optimization problems that cannot be solved using exact methods. Inference in Bayesian networks, learning the weights of artificial neural networks [1], determining efficient power usage in a sensor network [2] or finding the best configuration for a satellite antenna [3] are all examples of such problems. Because we cannot use exact methods, we turn to approximation methods and must make do with approximate answers.

Nevertheless, we are always looking for ways to improve the performance of these algorithms so that they find better approximations. Additionally, the No Free Lunch Theorem (NFLT) [4], proves no single algorithm will outperform random search across all optimization problems. So we will need many good (and sometimes just "good enough") algorithms.

There are whole host of these approximation methods including gradient descent, methods inspired by physics (Simulated Annealing), and methods inspired by biology (Genetic Algorithm, GA; [5]. Particle Swarm Optimization, PSO; [6]). Most of the biologically-based algorithms are based on single populations of competing individuals representing full candidate solutions. As the algorithms manipulate these individuals to search the solution-space, the best of them emerges as the approximate solution to our problem.

Research has shown that one way to improve our approximations is to decompose the problem into sub-problems. These algorithms, such as Cooperative Coevolutionary Genetic Algorithm (CCGA) [7] and Cooperative Particle Swarm Optimization (CPSO) [8], decompose a problem into *disjoint* subproblems and assign a GA or PSO to each subproblem. The partial solutions are then recombined into a solution to the full problem. This cooperative approach helps fight against the *curse of dimensionality.*

The curse of dimensionality describes the phenomenon where, as the dimensionality of a problem increases, we must increase the number of particles we use exponentially, if we are to search the space with the same density. Breaking a problem into sub-problems helps tackle the curse of dimensionality without completely solving it. Unfortunately, breaking a problem into sub-problems creates issues of its own, including the issue of pseudo-optima. A pseudo-minimum, for example, exists if the global

minimum in the sub-problem is not also the global minimum in the full problem.

Factored Evolutionary Algorithms (FEA) [9] are a multi-population variant of single population-based algorithms such as Particle Swarm Optimization and Genetic Algorithm. FEA is also very much like CCGA and CPSO but expands on those algorithms in important ways. First, FEA can use any many different optimization algorithms as the actual sub-problem optimizer. Second, FEA decomposes the problem into *factors* of possibly differing sizes with overlap—they are no longer disjoint but can have variables in common. And like CCGA and CPSO, this permits FEA tackle the curse of dimensionality. However, with the proper overlap, FEA can also prevent pseudo-optima.

Another issue arises in many of these single population algorithms such as GA and PSO called *hitchhiking*. These algorithms work by manipulating the individuals that represent full solutions towards better and better values. Hitchhiking occurs when the replacement solution is better overall than the current solution but some individual variables end up with worse values than those in the solution that was replaced. Because CCGA, CPSO and FEA decompose the problem, they all mitigate against hitchhiking although in slightly different ways.

If FEA decomposes a problem into subproblems and a "subpopulation" is assigned to each subproblem, which contains a subset of variables needed for a full solution, how are individuals in these subpopulations evaluated? FEA maintains a global

context that can be used by an subpopulation to fill in the missing values—values in the problem but not in the subpopulation. The challenge for this approach is that the global context is not effective in a distributed setting. Many optimization problems are computationally intensive. In an age of multi-core, networked machines, a distributed version of the algorithm would allow us to harness those machines. The first problem this dissertation seeks to address is FEA's centralized context. We solve this problem by introducing Distributed Factored Evolutionary Algorithms (DFEA), which assigns a local context to each subpopulation. We will show that DFEA often performs nearly as well as FEA and still better than the corresponding single population EA. This success is limited, however.

The problem is that FEA and DFEA should, theoretically, perform equally as well given the same starting conditions and we can demonstrate that they do not. In order to determine why the performance of FEA and DFEA diverge, we look at the dichotomy of cooperation versus competition that is often invoked as the reason multi-population algorithms are more successful than their single-population counterparts. Although this framework is evocative, it does not help us when both the algorithms are multi-population algorithms that appear to be cooperating to the same degree. The problem is that we need a different framework for analyzing these algorithms.

To solve that problem, we develop a new framework that describes the information flows and conflict resolution mechanism that are central to these algorithms. We

model information flows as a blackboard architecture [10] where subpopulations are communicating through the blackboard to suggest their best values for the parameters in the problem. The subpopulations also read the blackboard to obtain values that they need in order to optimize the subset of parameters assigned to them. Because subproblems overlap, there must be a conflict resolution mechanism when more than one subpopulation is suggesting a new value for a parameter. Looking at FEA and the algorithms that preceded it, we identified that the conflict resolution mechanism is guided by Pareto efficiency [11]. The conflict resolution process only accepts values for individual variables that are Pareto improvements; the new value replaces an existing value in the global context only if it improves the overall solution. This variable-by-variable approach to determining better solutions is what eliminates hitchhiking. By using the framework on FEA, we are better able to understand how the algorithm works. The new framework enables us to better understand these algorithms whereas the framework of cooperation and competition did not.

However, we still have the problem of dissimilar performance for FEA and DFEA. We are able to further validate the usefulness of our framework by applying it to DFEA and determining where the information exchange differs from FEA. After determining these differences, we develop a revised DFEA that solves the problem of divergent performance between the two algorithms.

Based on the insights gained from applying the new blackboard and Pareto-based

framework to FEA and DFEA, we return to the original problems that FEA and DFEA were meant to solve: curse of dimensionality and hitchhiking. We model the information exchange and conflict resolution in PSO exactly the same way we did in FEA and DFEA. The result is a new algorithm, Pareto Improving Particle Swarm Optimization (PI-PSO), that does not exhibit hitchhiking. As "solving' the curse of dimensionality is relative, PI-PSO solves that problem by outperforming PSO on most experiments and performing as well as FEA.

Finally, we return to our original problem of creating a distributed version of FEA. To solve this problem we implement both FEA and DFEA using the Actor model [12]. We are able to validate that the Actor model implementations preserve the information exchange and conflict resolution semantics as the original algorithms.

1.1 Contributions

Science is itself subject to a kind of Linnaean classification system with its own domains, kingdoms, phyla, classes, orders, families, genera, and species. Although our results ultimately reside in the domain of Computer Science and, within Computer Science, Artificial Intelligence, the problems we address lie on the outer limbs of that family tree, among the genera and species as do our contributions.

In this dissertation, we make several significant contributions to the families of

algorithms generally classified as Evolutionary Computation and Swarm Intelligence. These algorithms are used to solve complex optimization problems. The contributions are:

- **Distributed Factored Evolutionary Algorithms:** We develop the Distributed Factored Evolutionary Algorithms. DFEA is a extension of Factored Evolutionary Algorithms [9] in the same way that Distributed Overlapping Swarm Intelligence (DOSI) [13] extended Overlapping Swarm Intelligence (OSI) [14] to the distributed case. Like FEA, DFEA can be used with any "evolutionary algorithm" (for example, Genetic Algorithm and Particle Swarm Optimization).

- **Information Exchange and Conflict Resolution Framework:** FEA and DFEA are both the latest in a long line of multi-population algorithms that have emphasized the conflicting roles of cooperation and competition in biologically-inspired algorithms. As an alternative we develop a framework based on information exchange via a blackboard architecture and conflict resolution using Pareto efficiency.

- **Revised DFEA:** FEA and DFEA (as well as OSI and DOSI) have always had inconsistent performance when, at least on the surface, it had seemed like the distributed versions should perform equally as well as the centralized versions.

By applying the Information Exchange and Conflict Resolution Framework to DFEA, we identify differences in information flows between FEA and DFEA. This enables us to revise DFEA to match the information semantics of FEA. As a result, FEA and DFEA perform identically under identical initial conditions.

- **Pareto Improving Particle Swarm Optimization:** We apply our Information Exchange and Conflict Resolution framework to the selection of the *gbest* in the *gbest* Particle Swarm Optimization algorithm. By making the *gbest* a blackboard architecture rather than a simple cache, and extending variable by variable conflict resolution to particles, we create a single population algorithm that performs on a par with FEA. We also examine the comparative performance and scaling characteristics of the this PI-PSO as compared to PSO.

- **Actor-Based DFEA:** As developed, DFEA is distributed only in terms of state but leaves open questions of concurrency, parallelism, and distributed execution. We provide an implementation based on the Actor model that explores the implications of parallelism and asynchrony for our blackboard architecture.

1.2 Overview

One of the primary contributions of this dissertation focuses on the development of a framework for thinking about and analyzing a certain class of multi-population

evolutionary algorithms used for optimization.

In Chapter 2 we discuss the relevant background in stochastic local optimization. This includes the problems these algorithms encounter such as the curse of dimensionality and "hitchhiking" as well as unsurmountable obstacles such as the conclusions of the No Free Lunch Theorem. We review the major stochastic local search algorithms such as Hill Climbing and Simulated Annealing, as well as the biologically inspired, population-based algorithms such as the Genetic Algorithm and Particle Swarm Optimization. We conclude with a detailed review of the multi-population Factored Evolutionary Algorithms which are the starting point for the dissertation.

In Chapter 3, we develop a distributed version of FEA called Distributed Factored Evolutionary Algorithms. Just as FEA generalized OSI [2] from swarm intelligence to any evolutionary algorithm, DFEA generalizes DOSI [13] to the distributed case. In FEA, the various populations must share a global context. In DFEA, each population has its own context that must be coordinated with the other populations. For best performance, they must maintain identical values for those local solutions or *full consensus*. Our hypothesis is that, under full consensus, DFEA will perform equivalently to FEA and better than the single population version of the particular evolutionary algorithm. For these experiments we concentrate on Particle Swarm Optimization as the evolutionary algorithm in F*EA*.

We also perform experiments where full consensus between the individual contexts

is relaxed. Our hypothesis here is that the performance of DFEA will degrade when we relax consensus but only to the extent that variables in the optimization problem are interdependent.

In Chapter 4, we develop a different framework for thinking about multi-population algorithms such as FEA and DFEA. The line of research of which FEA and DFEA are a part often puts these algorithms within a spectrum of cooperation and competition. As an alternative, we develop a framework based on information exchange via a blackboard architecture and conflict resolution based on Pareto efficiency. We apply this framework to FEA to better understand how FEA is able to improve over single population algorithms and the evolution of FEA's blackboard over time.

In Chapter 5 we use the framework developed previously to examine the DFEA version developed in Chapter 3. By applying the framework to DFEA, we are able to determine how the information flows for DFEA and FEA differ over time, which explains the divergent performance of the two algorithms. We then use the framework to revise DFEA. We argue that previously observed divergence in performance of DFEA and FEA will be eliminated.

We also examine relaxed consensus with this revised DFEA. As before, we hypothesize that as consensus is relaxed, the DFEA's performance will degrade.

In Chapter 6, we use our framework from Chapter 4 again and apply it to Particle Swarm Optimization. We consider a single population PSO as having the same

kinds of information flows as FEA and apply a blackboard architecture and conflict resolution to the maintenance of the *gbest* rather than the selection of the *gbest*. We hypothesize that this Pareto Improving Particle Swarm Optimization (PI-PSO) algorithm will perform better than the standard *gbest* PSO and equivalently with FEA-PSO.

In Chapter 7 we examine the relative performance and scaling characteristics of PI-PSO as compared to PSO. Most experiments in evolutionary computation are carried out on a variety of problems of a single dimension with the same number of candidate solutions. We hypothesize, however, that many algorithms might have a certain amount of overhead; a more complicated or larger problem is required before their performance exceeds the performance of simpler algorithms. Additionally, while it is generally fair to keep as many parameters the same when comparing algorithms, once it has been demonstrated that one algorithm is better than another, it is instructive to see just how much better it is.

In order to test these hypotheses we perform a number of experiments between PI-PSO and PSO with varying number of candidates and varying dimensions. The general hypothesis is that when PI-PSO performs worse than PSO, it will perform better on a problem of higher dimension. Additionally, we hypothesize that when PI-PSO does perform better than PSO, it will do so with fewer particles.

DFEA is distributed in the sense of having a distributed state that must be kept in

sync to some degree. In Chapter 8 we implement a parallel, distributed version of FEA and DFEA using the Actor model. Using validation experiments, we demonstrate that the Actor implementations preserve the performance of FEA and DFEA.

Chapter 2

Background

In this chapter we discuss Factored Evolutionary Algorithms (FEA) [9] and their origins in the larger context of stochastic local search. This background informs discussions in the chapters that follow where we will extend FEAs to the distributed case (DFEAs), develop an alternative framework for analyzing these multi-population algorithms, revise DFEA, and devise variant of the canonical *gbest* Particle Swarm Optimization Algorithm.

2.1 Stochastic Local Search

The No Free Lunch Theorem for Optimization (NFLT) [4] proves that no algorithm can outperform random search averaged across *all* optimization problems.

Aside from the obvious implication that we need more than one search algorithm, the emphasis on random search is interesting because we can view random search as pure *exploration*. If we look at search algorithms in terms of balancing exploration and exploitation, we can interpret at least part of the NFLT result to mean that there will always be some problem for which our exploitation mechanism is a poor match. For example, our algorithm might rely on exploring and exploiting a local gradient in a continuously valued function that simply does not exist in an problem with variables that take on categorical values. So a broad array of algorithms and techniques will be required to solve all of our potential optimization problems. This may be analogous to inductive and representational bias in Machine Learning [15]. Additionally, the "solutions" in many cases will be approximate and even then we may have to accept solutions that are good enough.

One such category of algorithms is called *local* search. While not all such algorithms have a well-developed stochastic component, enough of them do that we will refer to them collectively as *stochastic local* search [16]. In stochastic local search this randomness is the central engine of both exploration and exploitation. The algorithms mainly differ in how they harness the information they obtain as they face the Multi-Armed Bandit problem [17]. At least for optimization, the Multi-Armed Bandit problem presents itself as a dilemma between exploiting a current solution which may turn out to be a dead-end or exploring new vistas that may not pay off

better than the current solution. As we will see, not all algorithms neatly separate their exploration and exploitation so they can be difficult to identify and separate out the various ways balance they balance these competing aims.

2.1.1 Notation

Throughout this dissertation, unless otherwise specified, lowercase and uppercase Latin characters like x, x_i, c refer to scalar values. They may also refer to functions as in $g()$ and $f()$ and records or objects such as $p.x$ and $S.best$. The only exceptions are in the case of X and R which refer to *variables*. Thus X_i is the variable X for the i-th dimension and x_i is the *value* of that variable.

In mathematics, we often only have vectors and matrices. In algorithms, we have collections: lists, vectors, arrays, sets, and hashmaps, to name but a few. In general, $\mathbf{X}$ refers to a collection. The type of collection may not matter although sometimes it does. If we use c_j to refer to a single value in $\mathbf{c}$ then it is an ordered collection and if we use an iterator of some kind as in $x \in \mathbf{x}$, it is an unordered collection. However, we will often use $\mathbf{c}[j]$ in algorithms during assignment as $\mathbf{c}_j \leftarrow 2$ does not quite capture what is meant in programming in this case. Finally, we can have nested collections which are indicated by a bold, script: $\boldsymbol{\mathcal{X}}$ or $\boldsymbol{\mathcal{A}}$. Again, the context will indicate if they are ordered or unordered collections.

No notation is completely airtight so any exceptions will be noted.

2.1.2 Challenges for Optimization

Consider a continuous real-valued function $f()$ of d variables, $\mathbf{X} \in \mathbb{R}^d$. We want to find an optimal value, $\mathbf{x}^*$, for $\mathbf{X}$, either as a global minimum or a global maximum. Without loss of generality, we will consider the case of a global minimum.

As an illustration of such a function, we will take the Eggholder benchmark optimization function [18].

$$f(\mathbf{X}) = \sum_{i=1}^{d-1}[-(X_{i+1} + 47)\sin\sqrt{|X_{i+1} + X_i/2 + 47|} - X_i \sin\sqrt{|X_i - (X_{i+1} + 47)|}$$

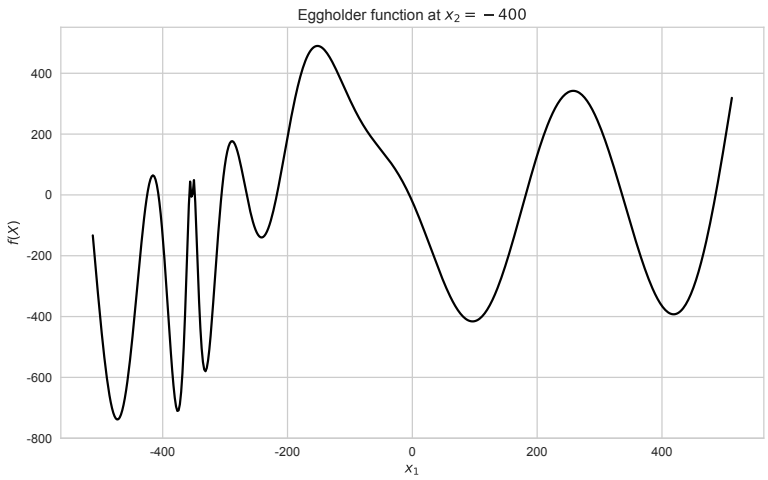
Figure 2.1 shows a cross section of a two dimensional $(2d)$ version of the Eggholder function plotted over the open interval $(-512, 512)$. We have set X_2 to -400 for the purposes of discussion.

The general challenge for any optimization algorithm is to find the global minimum when there are many local minina. The Eggholder function is a good example of this challenge; one that often increases as the number of dimensions in the problem increase. A perhaps less obvious challenge is existence of plateaus.

To illustrate that particular challenge for optimization algorithms, let us consider the Michalewicz benchmark optimization function [8].

$$f(\mathbf{X}) = -\Sigma_{i=1}^{d} \sin(X_i)[\sin(\frac{iX_i^2}{\pi})]^{2m}$$

Figure 2.1: *Example Optimization Problem with Multiple Optima*



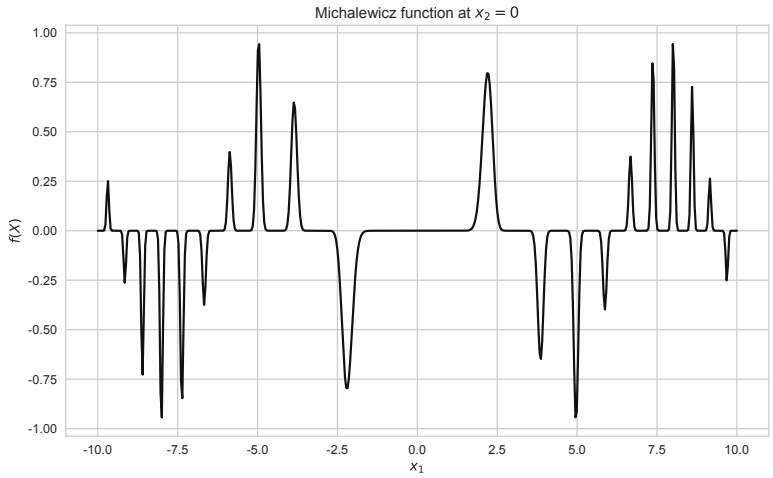
with $m = 10$. We have plotted a cross-section of the $2d$ version of the Michalewicz function on the open interval $(-10, 10)$ with $X_2 = 0$ (Figure 2.2). As we can see, there is a large plateau around $X_1 = 0$ and the smaller plateaus throughout the entire interval.

2.1.3 Hill climbing

As with all such algorithms, there are many variants of Hill Climbing (HC) [19]. Because we are only interested in the broad conceptual themes at this juncture, we will concentrate on the simplest one.

We begin with a candidate solution, $\mathbf{x}$, generated at random. We generate potentially better candidates by using the notion of a *neighborhood* whereby we take $\mathbf{x}$

Figure 2.2: *Optimization Problem with Multiple Optima and Plateaus*



and use the neighborhood function to calculate possibly better candidate solutions "near" $\mathbf{x}$. One way to generate $\mathbf{x}'$ is by looking in some Δ-neighborhood of $\mathbf{x}$ so that $\mathbf{x}' = \mathbf{x} + \Delta\mathbf{x}$.

If $f(\mathbf{x}') \leq f(\mathbf{x})$ then we will take $\mathbf{x}'$ as the new candidate solution; otherwise, we keep $\mathbf{x}$. The "less than or equals" allows the algorithm to traverse plateaus to some degree. More generally, for $d > 1$, we will examine each x_i in turn and pick the first change with an improvement ("Simple Hill Climbing"). Alternatives include picking the change with the *most* improvement ("Steepest Ascent Hill Climbing") and generating x_i at random ("Stochastic Hill Climbing"). The algorithm continues until one more more stopping criteria are met as shown in Algorithm 2.1 and the candidate solution is returned. Throughout this dissertation we refer to the result as

Algorithm 2.1 Simple Hill Climbing

Input: Objective function f
Output: Candidate Solution $\mathbf{x}$

1: $\mathbf{x} \leftarrow initialize()$
2: **repeat**
3: $\quad \mathbf{x}' \leftarrow neighbor(\mathbf{x})$
4: $\quad$ **if** $f(\mathbf{x}') \leq f(\mathbf{x})$ **then**
5: $\quad\quad \mathbf{x} \leftarrow \mathbf{x}'$
6: $\quad$ **end if**
7: **until** stopping criteria
8: **return x**

the "candidate solution" because we are never *guaranteed* that the solution that was found was actually the global minimum.

As a side note, we may sometimes use multiple stopping criteria (Line 7) because we may want to stop if one of any number of criteria are met. For example, we could stop if we have run some fixed number of iterations *or* the candidate has stopped improving. This will be true of all algorithms in this dissertation where *stopping criteria* are referenced even though we will always use a single criterion, a fixed number of iterations, for our experiments.

Considering Algorithm 2.1 applied to the function in Figure 2.2, the most obvious problem is that Simple Hill Climbing will get stuck in local minima. Although this depends a bit on the definition of the *neighbor* function, if the neighbood *delta* is too large, Hill Climbing becomes random search. For example, if we start at $x_1 = 0$, we are likely to move in the direction of the local minimum at about $x_1 = -2.4$ never

getting to either $x_1 = -8.0$ or $x_1 = 5.0$. There are several alternatives to Simple Hill Climbing that attempt to solve this problem.

One alternative is simply to run the algorithm multiple times. With *Random Restart Hill Climbing*, a Simple Hill Climbing algorithm is executed n times from a new starting point and the best X found is kept.

2.1.4 Simulated Annealing

Simulated Annealing (SA) [20] is an algorithm related to Hill Climbing with a critical difference. It sometimes accepts an inferior successor candidate. The basic algorithm is shown in Algorithm 2.2.

The main difference between the Hill Climbing and Simulated Annealing is in Line 7. Unlike Simple Hill Climbing, Simulated Annealing adds an *else* branch that entertains the possibility of accepting an $\mathbf{x}'$ that is actually inferior to $\mathbf{x}$. The probability of inferior exchanges is controlled by the *annealing schedule* for τ (Line 13). Based on this schedule, p slowly decreases over time making inferior changes less likely (Line 8).

This particular version of Simulated Annealing is called *Boltzmann annealing* and has been proven to converge to the global optimum if τ is decreased logarithmically with time, t [21]. In practical applications, many more iterations may be required because of the stochastic nature of the algorithm. Both Hill Climbing and Simulated

Algorithm 2.2 Simulated Annealing

Input: Objective function f
Output: Candidate solution $\mathbf{x}$

```
1:  τ ← large value
2:  x ← initialize()
3:  repeat
4:      x' ← neighbor(x)
5:      if f(x') ≤ f(x) then
6:          x ← x'
7:      else
8:          p ← e^{-(f(X')-f(X))/τ}
9:          if rand() < p then
10:             x ← x'
11:         end if
12:     end if
13:     τ ← τ − Δτ
14: until stopping criteria
15: return x
```

Annealing harness a random component for exploration. In Hill Climbing, there is a strong exploitation strategy in accepting only successors that do not make the objective function worse. In Simulated Annealing, this same strategy exists along with an augmenting exploration strategy that sometimes accepts transitions to inferior solutions. This is what permits SA to escape local minima. However, as we move to more complicated algorithms, the "split" between exploration and exploitation can get less clear.

In many ways, some of the key differences between the stochastic local search algorithms reside exactly in how they balance exploration and exploitation. Another key difference is how many candidate solutions they work with at once.

Both Hill Climbing and Simulated Annealing have variants that keep track of multiple candidate solutions and thus can be thought of running many restarts in parallel. If we think of these multiple candidates as individuals in a population, then it becomes easy to entertain the idea of using drawing inspiration from biological processes to design new, different and, hopefully, better algorithms.

2.1.5 Biologically Inspired Algorithms

There are many search and optimization algorithms inspired by nature and natural processes [22]. Simulated Annealing itself is inspired by the controlled cooling of metals and the properties of the resulting crystalline structures. Other algorithms are inspired by biological processes. In many of the biologically-inspired algorithms, individuals in *populations* interact in more direct ways as part of the combined exploration and exploitation strategy. While there are many such algorithms, we will focus on two of them: *Genetic Algorithm* and *Particle Swarm Optimization*.

2.1.5.1 Genetic Algorithm

The Genetic Algorithm (GA) is attributed to Holland [5]. The canonical version (Algorithm 2.3) follows the general outlines of most population-based algorithms for stochastic local search. We can think of each individual as a record, *individual*, with fields *genes* and $fitness$. The algorithm starts out with a randomly initialized

Algorithm 2.3 Genetic Algorithm

Input: Objective function f, probability of crossover $p_{crossover}$, probability of mutation $p_{mutation}$
Output: Candidate solution $\mathbf{x}$

1: $\mathbf{P} \leftarrow initialize()$
2: $\mathbf{P} \leftarrow evaluate(\mathbf{P})$
3: **repeat**
4: $\quad \mathbf{P}' \leftarrow List()$
5: $\quad$ **for** i in $len(\mathbf{P})/2$ **do**
6: $\quad\quad parent_1, parent_2 \leftarrow$ select($\mathbf{P}$)
7: $\quad\quad$ **if** $rand() < p_{crossover}$ **then**
8: $\quad\quad\quad i \leftarrow randint(len(parent_1.\mathbf{x}))$
9: $\quad\quad\quad child_1.genes \leftarrow parent_1.genes[0:i] + parent_2.genes[i:]$
10: $\quad\quad\quad child_2.genes \leftarrow parent_2.genes[0:i] + parent_1.genes[i:]$
11: $\quad\quad\quad parent_1, parent_2 \leftarrow child_1, child_2$
12: $\quad\quad$ **end if**
13: $\quad\quad parent_1 \leftarrow mutate(p_{mutation}, parent_1)$
14: $\quad\quad parent_2 \leftarrow mutate(p_{mutation}, parent_2)$
15: $\quad\quad P'.append(parent_1)$
16: $\quad\quad \mathbf{P}'.append(parent_2)$
17: $\quad$ **end for**
18: $\quad \mathbf{P} \leftarrow evaluate(\mathbf{P}')$
19: **until** stopping criteria met
20: $\mathbf{x} \leftarrow decode(best(\mathbf{P}).genes)$
21: **return** $\mathbf{x}$

population of candidate solutions (Line 1), which is then evaluated (Line 2), filling in the $fitness$ fields. The algorithm then proceeds to generate a successor population (Lines 5-17).

In the canonical GA, even if the underlying optimization problem is a continuous numerical function, the candidate solutions are represented as strings of bits ("bit-strings"). These bit-strings are interpreted to be a *genotypic* representation of

a candidate solution with the real valued decoding interpreted as a *phenotypic* expression of those genes. The new population is created by manipulating the genomic representation of individuals in each generation to produce a new one.

First, a set of parents is selected to generate offspring. The selection was originally through Weighted Roulette Wheel selection where individuals were chosen (with replacement) probabilistically proportionate to their fitness (Line 6). The pair selected then probabilistically generate offspring according to the probability of crossover, $p_{crossover}$ (Line 7). If the test fails, the pair are passed to the next step. If crossover does occur, a locus is chosen randomly on the parents separating each into two substrings: $A = parent_j.genes[0 : i]$ and $B = parent_j.genes[i :]$. The children are assembled by concatenating the substrings from different parents: $A_{parent_1} + B_{parent_2}$ and $A_{parent_2} + B_{parent_1}$ (Lines 8 - 11).

In the canonical GA, the *mutate* operator does a bit-by-bit test with $p_{mutation}$ to see if the bit is flipped. In other formulations, we can do one test to see if a child is mutated and then pick a random location to flip the bit. After the algorithm runs for a specified number of *generations* (Line 19), the algorithm returns the decoded genes for the best of the final population as the candidate solution (Line 21).

One peculiarity of GA is that it may stumble upon a great solution in Generation 257 and then lose that solution in the next generation, never to recover it. In order to combat this problem, *elitism* is sometimes introduced into the algorithm [23].

The elitism "operator" always copies the best individual of a generation into the next generation. Unfortunately, elitism can exacerbate a different problem in GA known as "premature convergence" [24]. Premature convergence happens when the population has become homogeneous (or mostly so) with respect a particularly fit individual that represents a local minima. Elitism can encourage this genetic homogeneity.

Following on the previous discussion of exploration versus exploitation, we can see that many of these elements are a bit muddled together in the Genetic Algorithm. Selection is probabilistic (exploration), but we are more likely to pick fit individuals (exploitation). Crossover generates new candidates (exploration) but only out of the existing genetic material (exploitation). Mutation may perhaps be the only operator that involves pure exploration.

Perhaps more importantly for the discussion yet to come, one of the most interesting things about the Genetic Algorithm and its accompanying literature is the importance of analogy for the algorithm. The central analogy of the Genetic Algorithm is "Survival of the Fittest" or *competition*. In the GA, the members of the population compete for the chance to spread their genes into the next generation. Fit individuals are selected and, through crossover and mutation, produce hopefully more fit variants as offspring. If an individual is fit enough, it is selected many times to participate, and many variants of its genetic material end up in the successor generation.

Algorithm 2.4 Particle Swarm Optimization

Input: Objective function f, inertia ω, exploration parameters ϕ_1, ϕ_2
Output: Candidate solution $\mathbf{x}$

1: $\mathbf{P} \leftarrow initialize()$
2: **repeat**
3: **for** p in $\mathbf{P}$ **do**
4: $p.\mathbf{v} \leftarrow \omega p.\mathbf{v} + \phi_1 \mathbf{u_1}(gbest.\mathbf{x} - p.\mathbf{x}) + \phi_2 \mathbf{u_2}(p.pbest.\mathbf{x} - p.\mathbf{x})$
5: $p.\mathbf{x} \leftarrow p.\mathbf{x} + p.\mathbf{v}$
6: **if** $f(p.\mathbf{x}) < f(p.pbest.\mathbf{x})$ **then**
7: $p.pbest \leftarrow p$
8: **end if**
9: **end for**
10: $gbest \leftarrow find\text{-}global\text{-}best(gbest, \mathbf{P})$
11: **until** stopping criteria
12: **return** $gbest.\mathbf{x}$

2.1.5.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is another biologically inspired algorithm, although this time the cues are taken from groups or flocks of birds, fish and even people [25]. Here we describe the *gbest* variant of PSO [6].

The PSO algorithm operates on a population (swarm) of candidate solutions (particles), Algorithm 2.4. Each particle has a position, $\mathbf{x}$; velocity, $\mathbf{v}$; fitness, $f(\mathbf{x})$; and the best position it has attained so far or "personal best", *pbest*. The algorithm begins with particles initialized to random positions (Line 1). Each iteration updates every particle's velocity and position and, if warranted, its *pbest* (Lines 3-9). After all particles are updated, the global best, *gbest*, is updated from the swarm's current set of personal bests (Line 10). Because all particles are updated before the *gbest* is

evaluated as opposed to after each particle is updated, this version simulates a *parallel* algorithm [26] instead of an *sequential* one [6].

The velocity update equation combines three components. The first is the ω inertia component applied to the previous velocity. The second is the *public* or social component calculated by taking the difference between the swarm's global best, *gbest*, and the particle's current position, $p.\mathbf{x}$ and then mixing in a randomizing effect calculated by multiplying ϕ_1 times a vector of random numbers on the interval $(0, 1)$, $\mathbf{u_1}$. The third is the *cognitive* or individual component calculated by taking the difference between the particle's personal best's position, $p.pbest.\mathbf{x}$, and the particle's current position, $p.\mathbf{x}$ and then mixing in a randomizing effect calculated by multiplying ϕ_2 times a vector of random numbers on the interval $(0, 1)$, $\mathbf{u_2}$.

We can once again see a mixture of exploration and exploitation. The second and third components include both exploitation by taking the difference between a best position and the current position and exploration by adjusting by an exploration constant, ϕ_i. Because ϕ_i is usually between 1 and 2 and each element of $\mathbf{u_i}$ is between 0 and 1, the exploration factor ranges from 0 to 2.

As each new particle position is calculated, we compare $f(p.pbest.\mathbf{x})$ and $f(p.\mathbf{x})$ to determine if a new personal best has been achieved (Lines 2-4). After all particles' *pbest*s are updated, we pick the best as the new global best (Line 10, Algorithm 2.5). Because each *pbest* is only updated if $p.\mathbf{x}$ is better than the current $pbest.\mathbf{x}$, and *gbest*

Algorithm 2.5 PSO find-global-best

Input: Current global best *gbest*, Current swarm **P**
Output: New global best *gbest*

```
1: for p in P do
2:     if f(p.pbest.x) < f(gbest.x) then
3:         gbest ← p.pbest
4:     end if
5: end for
6: return gbest
```

is only updated if a *pbest* is better than *gbest*, the *gbest* is a non-decreasing function of *pbest*s.

The update process is repeated a fixed number of iterations or until some other stopping criterion is met. The *gbest* (or just the *gbest*'s position) is returned as the candidate solution (Line 6).

2.1.6 Challenges

Although both Genetic Algorithm and Particle Swarm Optimization have been quite successful, they are not without problems. First, like all stochastic local search algorithms, they are subject to the *curse of dimensionality* [27, 28, 29]. As the dimensionality of a problem increases, other things being equal, the number of individuals required in the population to achieve the same level of performance must generally increase exponentially [8]. Second, both algorithms are susceptible to a phenomenon known in the GA literature as *hitchhiking* [30]. In the PSO literature, this has been

Table 2.1: *Hitchhiking in PSO*

$pbest_j$	$\mathbf{X}$	$f(\mathbf{x})$
1	[1.53, 1.84, 5.29, 0.59]	34.06
2 ($gbest_{new}$)	[0.42, 2.01, 4.76, 1.84]	30.26
3	[3.23, 0.72, 4.68, 0.47]	33.07
4	[2.83, 3.83, 2.71, 1.27]	31.64
$gbest_{old}$	[2.39, 1.24, 5.71, 0.34]	39.97

called "Two Steps Forward, One Step Back" [8]. We will use the term *hitchhiking* to describe the phenomena in both GA and PSO.

Hitchhiking is most easily explained with a concrete example. Suppose we are trying to minimize the four-dimensional Sphere function ($\sum_{i=1}^{4} X_i^2$) on the interval $[0, 10]^4$ with four particles, and we find ourselves at the end of an arbitrary iteration ready to call Algorithm 2.5. Although hitchhiking can occur in all functions, we use the Sphere function for this example because it is separable. Separability permits the unambiguous attribution of changes in individual variables to overall fitness. If x_i increases, $f(\mathbf{x})$ increases; if x_i decreases, $f(\mathbf{x})$ decreases.

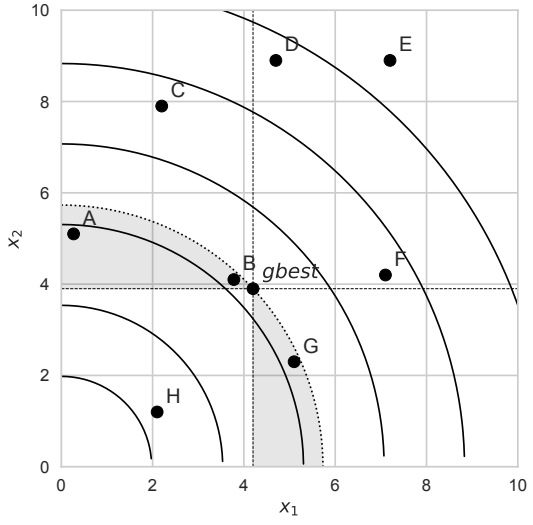
Table 2.1 shows current *pbest*s and fitnesses of the four particles. Particle 1's *pbest* has a fitness of 34.06; Particle 2's has a fitness of 30.26; Particle 3, 33.07; and Particle 4, 31.64. The current global best, $gbest_{old}$, is shown at the bottom of Table 2.1. As previously mentioned, the current *gbest* must always be one of the particles' *pbest* in the version of PSO we are describing. We do not see that here because the *pbest*s have been overwritten in the previous loop. This means that if no *pbest* was better

than $gbest_{old}$, then $gbest_{old}$ would *have* to be one of the *pbest*s in the table. Because Particle 2's personal best has the lowest fitness, 30.26, it will become the new global best, $gbest_{new}$.

However, if we make a pairwise comparisons for each x_i, we can see that while the Particle 2 was a global improvement, it was not an improvement for individual variables. A lower value of X_i is unambiguously better in the Sphere function so we can see that X_1 in $gbest_{old}$ was 2.39 while it is 0.42 in $gbest_{new}$. This is similarly true for X_3. However, X_2 in $gbest_{new}$ is actually larger than its counterpart in $gbest_{old}$, 2.01 versus 1.24. The same is true for X_4. The individually inferior values for X_2 and X_4 (red/italics) are *hitchhikers*.

We can see how this might generally arise in the Sphere function by looking at a

Figure 2.3: *Selecting gbest in PSO (Sphere) and Hitchhiking*



contour or *isoquant* plot of the Sphere function and hypothetical *pbest*s. Figure 2.3 shows this for the case of two variables. In this figure, the arcs represent the contours of the Sphere function for two variables, X_1 and X_2. The current $gbest = (4.2, 3.9)$ also defines a contour (dotted) that is the dividing line between *pbest*s that have a better fitness (a lower contour) or a worse fitness (a higher contour). Additionally, the gray areas denote the set of points where the *pbest* lies on a lower contour than the *gbest* and thus has a better fitness but one or the other of the variables is larger than its value in *gbest*. All points in the gray zones include hitchhiking. We can thus see that *pbest*s C, D, E, F are all inferior to the current *gbest*, and *pbest*s A, B and G involve hitchhikers. Only *pbest* H has both a better fitness and no hitchhiking. Thus if H did not exist, *pbest* A would be chosen as *gbest*, hitchhikers and all. Although throughout our research we concentrate on eliminating hitchhikers, it is not clear that all hitchhiking is bad. Like the acceptance of inferior solutions in Simulated Annealing, at least some hitchhiking could actually help the algorithm find the global solution.

2.2 Factored Evolutionary Algorithms

We previously mentioned the importance placed on competition in biologically inspired algorithms, especially the Genetic Algorithm. One approach researchers have

taken to solving the problem of hitchhiking in both GA and PSO is by introducing *cooperation* via multi-population versions of the algorithms. One such family of multi-population algorithms is *Factored Evolutionary Algorithms*.

Factored Evolutionary Algorithms [9, 31, 32] constitute a family of algorithms that decompose an optimization problem into subsets of variables and apply individual populations to those *factors*. They are considered to be a family of algorithms because any evolutionary algorithm can be used for optimization of a factor. This means there is an FEA-GA, FEA-PSO, FEA-HC, FEA-SA, etc. All of these share some general characteristics by virtue of the FEA part but have specific performance characteristics by virtue of the specific evolutionary algorithm used. In order to better understand FEA, its use of multiple populations, and the rationale for factoring an optimization problem, we first discuss the history of the algorithm.

2.2.1 History

As previously discussed, stochastic local search algorithms such as the Genetic Algorithm and Particle Swarm Optimization are susceptible to the curse of dimensionality. As the size of a problem increases, in general, the resources required for the same level of performance increase exponentially because the problem space increases exponentially. Additionally, algorithms such as the GA and PSO suffer from hitchhiking, which appears to be an inherent characteristic of the algorithms. It is

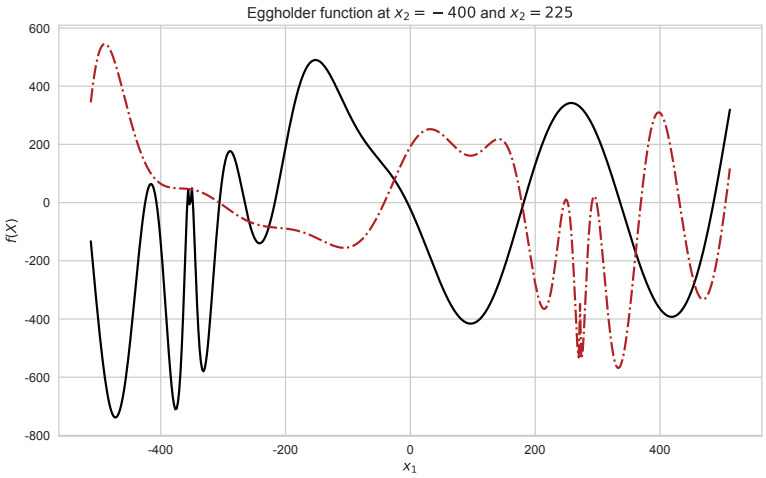
worth noting that these problems are related. As the size of solutions increase, the probability of hitchhiking increases as well. We appear to be doubly cursed.

Potter and de Jong developed one of the original approaches to addressing this issue for the Genetic Algorithm [7]. Their solution was to decompose the problem down to the individual variables and apply a GA to each variable. At any given moment, the candidate solution to the problem was the concatenation of the best results found in each population. For example, if we take a simple $4d$ problem, we might have each of X_1, X_2, X_3 and X_4 optimized by its own GA. The candidate solution is the concatenation of the best individuals from the variable-specific GA populations. The populations thus appeared to be collaborating subspecies each working on a different section of the problem. This *cooperative* approach was contrasted with the competitive nature of the canonical Genetic Algorithm and was called the Cooperative Coevolutionary Genetic Algorithm (CCGA).

Van den Bergh and Engelbrecht [33] applied a CCGA-like version of PSO to training neural networks. They later generalized their algorithm creating the Cooperative PSO (CPSO) [8] as an approach to addressing the "Two Steps Forward, One Step Back" problem, as they characterize hitchhiking in PSO. However, they went a step further by recognizing that the CCGA approach introduces problems of its own.

Decomposing an optimization problem into its constituent variables and solving these individually implies strong assumptions about the independence of the vari-

Figure 2.4: *Example Optimization Problem under Different x_2 values*



ables' values. This assumption is true for some problems like Sphere—a fact we have exploited. However, it is unlikely to hold true for all optimization problems.

In Figure 2.4, we have plotted two cross sections of the Eggholder function from Section 2.1.2 with different values for X_2. The solid (black) line is the same line as before with $X_2 = -400$. The dotted (red) line is plotted with a value of $X_2 = 225$. If we compare these two lines, we see that the minimizing values of X_1 will sometimes be at odds with each other. In fact, looking at $X_1 = -500$, under $X_2 = -400$ we are near a global minimum but under $X_2 = 225$ we are near a global maximum.

In general, if the optimal values of variables are related to each other, then they must be discovered jointly. In keeping with the genetic metaphor, in the GA literature, this phenomenon is called *epistasis*. Potter and de Jong recognized this was a problem.

In their own research, Van der Bergh and Engelbrecht recognized this was a problem and that the partitioning of the variables mattered. They labeled this phenomenon *pseudo-optima*. Again, concentrating on the case of pseudo-*minima*, one way that CPSO sought to avoid such problems was by partitioning the problem into larger groups of variables than CCGA had done. For example, our $4d$ problem might be partitioned into (X_1, X_2) and (X_3, X_4). Of course, the larger these groups are, the more likely the individual groups will begin to experience hitchhiking themselves. Therefore Van der Bergh and Englebrecht introduced the idea of the Hybrid CPSO that would alternate between a CPSO optimizing smaller groups of variables and a PSO optimizing all the variables. In keeping with the established metaphor, they added more competition back into the algorithm.

In a different chain of research starting with Haberman *et al.* [2] and culminating with Fortier *et al.* [34, 35], an alternative solution was developed to address the potential for pseudo-minima in PSO called Overlapping Swarm Intelligence (OSI). The OSI algorithm differs from the basic PSO in that it subsets the variables of a problem into overlapping groups, or *factors*, that are optimized by individual PSOs. This factorization of the optimization problem is similar to how factorization in mathematics decomposes a polynomial into a product of factors. The important innovation was to extend the decomposition of a problem into possibly *overlapping* factors instead of the CPSO's *disjoint* factors.

OSI has been applied successfully to a wide range of problems, such as energy aware routing in sensor networks [2], training deep neural networks [14], performing abductive inference in Bayesian networks [36], and learning Bayesian networks [34, 35].

2.2.2 Algorithm

Factored Evolutionary Algorithms (FEA) [9] generalize and improve upon OSI in several important ways. First, FEA abstracts out the actual optimization of factors into an *Optimize Step* into which any evolutionary algorithm can be inserted. Second, while OSI requires the factors to overlap with one another [36], FEA does not. If we continue our example from above, this means that we could optimize (X_1, X_2), (X_2, X_3) and (X_2, X_3, X_4) as individual factors, and we can use a GA, PSO, or some other algorithm to do so. FEA is thus more general than both OSI or CPSO because factors can overlap and because other optimization algorithms can be used. Additionally, if we wished, we could always include a factor that covered all the variables (X_1, X_2, X_3, X_4). This makes FEA more general than Hybrid CPSO as well.

This enabled FEA to avoid hitchhiking and avoid pseudo-optima while also being generally applicable to a wide range of evolutionary algorithms, including the GA, PSO and others. Strasser also demonstrated that CPSO was a special case of FEA-PSO combination and that FEA-PSO generally performed better than CPSO

Algorithm 2.6 Factored Evolutionary Algorithms

Input: Function f, Evolutionary Algorithm ea
Output: Context $\mathbf{c}$ as candidate solution $\mathbf{x}$

1: $\boldsymbol{\mathcal{X}} \leftarrow factorize(\mathbf{X})$
2: $\mathbf{S} \leftarrow ea.initialize(f, \boldsymbol{\mathcal{X}})$
3: $\mathbf{c} \leftarrow initialize\text{-}context(\mathbf{S})$
4: $\boldsymbol{\mathcal{O}} \leftarrow identify\text{-}optimizers(\boldsymbol{\mathcal{X}})$
5: **repeat**
6: **repeat**
7: **for** S in $\mathbf{S}$ **do**
8: $S \leftarrow ea.update(S)$
9: **end for**
10: **until** stopping criteria
11: $\mathbf{c} \leftarrow compete(f, \mathbf{S}, \boldsymbol{\mathcal{O}}, \mathbf{c})$
12: $share(f, \mathbf{S}, ea, \mathbf{c})$
13: **until** stopping criteria
14: **return** $\mathbf{c}$

or Hybrid CPSO [9].

The FEA algorithm is shown as Algorithms 2.6–2.8. The main FEA algorithm is Algorithm 2.6 and basically glues three steps together: *Update*, *Compete*, and *Share*.

The algorithm starts by decomposing $\mathbf{X}$ into factors (Line 1). Line 2 uses the constituent Evolutionary Algorithm, A, to initialize the individual populations assigned to each factor (which we will call "sub" populations), $\mathcal{S}$. Because factors may be of different sizes and overlap, we must discover the set of optimizers, $\mathcal{O}$. Each element of $\boldsymbol{\mathcal{O}}$, $\mathbf{O}_i$, is the set of optimizers for X_i. This is accomplished in Line 4. We then construct an initial context, $\mathbf{c}$, that represents the candidate solution, $\mathbf{x}$ (Line 3). The context could begin as a concatenation of the best individuals of the sub-populations

as in CCGA or as an algorithm-appropriate individual initialized with values for all of $\mathbf{X}$. We will use the latter approach in the discussions that follow.

We then begin the main loop (Line 5) that will alternate between the three steps identified earlier. The Update Step occurs in Lines 6–10. Each population is updated for one iteration in Line 8 where an iteration might be a single generation in a GA or single swarm update in a PSO. The number of updates is controlled by the loop. After the Update Step, FEA applies the Compete Step and Share Step.

The Compete Step is described in Algorithm 2.7. Here the algorithm loops through each variable, x_i. At Line 3, the current context $\mathbf{c}$ is evaluated and the value for the current x_i stored (Line 4). The context must be re-evaluated each time through the loop because it is being updated dynamically, variable by variable. The algorithm then loops through all the identified optimizers of X_i (Lines 4-11) and compares each one within the context of $\mathbf{c}$ in order to pick the best one. If a better x_i is found, then the context is updated in Line 12. If no better value was found, this is a "no-op."

The Share Step is described in Algorithm 2.8 and mostly involves bookkeeping for the individual subpopulations as a result of identifying a new context. In Line 2, we take the set difference of the global context/candidate solution, $\mathbf{c}$, and the values of $\mathbf{X}$ that this particular swarm is optimizing (the actual factor) as the values of the residuals $\mathbf{r}$. In order for $\mathbf{S}$ to use $\mathbf{r}$ to evaluate its individuals, we create a partially applied version of f using $\mathbf{r}$. Next, we identify the worst member of the

Algorithm 2.7 FEA Compete

Input: Objective function f, Subpopulations $\boldsymbol{\mathcal{S}}$, Optimizers $\boldsymbol{\mathcal{O}}$, Global context $\mathbf{c}$
Output: Global context $\mathbf{c}$

```
1:  for j = 1 to d do
2:      fitness ← f(c)
3:      value ← c[j]
4:      for i in O_j do
5:          candidate ← S[i].best
6:          c[i] ← candidate.x[i]
7:          if f(c) ≤ fitness then
8:              value ← candidate.x[i]
9:              fitness ← f(c)
10:         end if
11:     end for
12:     c[i] ← value
13: end for
14: return c
```

subpopulation (Line 4) and replace it with the proper values from the context (Line 5), encoding those values if required (for example, for the GA). Finally, we set the subpopulation's new, subpopulation specific objective function and re-evaluate the entire subpopulation (Lines 6 and 7).

2.2.3 Avoiding Hitchhiking

In the previous section, we described FEAs and how they worked. Now we will give an example of how it avoids hitchhiking. This example covers FEA-PSO and matches the example given previously in Table 2.1. We will again take up the Sphere function because its separability property makes changes in the values easy to interpret.

Algorithm 2.8 FEA Share

Input: Objective function f, Subpopulations $\mathbf{S}$, Evolutionary Algorithm ea, Context $\mathbf{c}$
Output: Subpopulations $\mathbf{S}$

1: **for** S in $\mathbf{S}$ **do**
2: $\quad \mathbf{r} \leftarrow \mathbf{c} \setminus S.X$
3: $\quad f_r \leftarrow partial(f, \mathbf{r})$
4: $\quad p \leftarrow ae.worse(S)$
5: $\quad p.\mathbf{x} \leftarrow \mathbf{c} \setminus \mathbf{r}$
6: $\quad S.f \leftarrow f_r$
7: $\quad ae.reevaluate(S)$
8: **end for**

Table 2.2: *FEA-PSO Determination of C_{new} with Overlapping Factors*

$gbest_j.\mathbf{x}$	$\mathbf{X}$	$f(\mathbf{x})$
$\mathbf{c}$	[2.39, 1.24, 5.71, 0.34]	39.97
S_1	[1.53, 1.84, ----, ----]	38.45
S_2	[----, 2.01, 4.76, ----]	32.53
S_3	[----, ----, 4.68, 0.47]	29.37
S_4	[----, ----, ----, 1.27]	41.47
$\mathbf{c}_{new}$	[1.53, 1.24, 4.68, 0.34]	25.90

In Table 2.2, we see four subswarms, S_i, instead of four particles (previously we thought of populations as vectors, $\mathbf{S}_i$, but here we think of them as records carrying around additional information). Each swarm optimizes a factor of $\mathbf{X}$ that we previously referred to as $S.\mathbf{X}$. For example, S_1 optimizes (X_1, X_2). The dashes represent values that are filled in from the context, $\mathbf{c}$. The current context, $\mathbf{c}$, is shown at the top of the table. The new context is shown at the bottom of the table.

We assume we have just completed an arbitrary Optimize Step and have entered the Compete Step shown in Algorithm 2.7. Looking down the columns for X_1, X_2,

etc., we see that X_1 only has the one optimizer, S_1. Following Algorithm 2.7, we try $\mathbf{S}_1.gbest.\mathbf{x}_1 = 1.53$ in $\mathbf{c}$ and discover that it is better than the existing value $c_1 = 2.39$. The new context, $\mathbf{c}_{new}$, is updated with $c_1 = 1.53$ (blue/bold).

In contrast, if we look at the column for X_2, we will see that both $\mathbf{S}_1$ and $\mathbf{S}_2$ are optimizers for X_2. Additionally, neither of the values $\mathbf{S}_1.gbest.\mathbf{x}_2 = 1.84$ or $\mathbf{S}_2.gbest.\mathbf{x}_2 = 2.01$ are better than the existing value of $c_2 = 1.24$ (blue/bold). Therefore c_2 remains unchanged in the new context, $\mathbf{c}_{new}$. Repeating this procedure for X_3 sees the context updated with $\mathbf{S}_3.gbest.\mathbf{x}_3$ and c_4 remaining unchanged. Because the successor candidate is constructed using variable by variable comparisons, there is no hitchhiking. Additionally, with overlapping swarms, we have more values for each X_i to chose from and we may avoid pseudo-minima. These are themes that will recur throughout this dissertation.

2.3 Summary

There are many categories of optimization algorithms not all of which are applicable to all problems. For example, it is difficult if not impossible to conceive of how we might apply an analytical approach to optimization to the parameters of a simulation. One category of optimization algorithms that has met with success in these and other situations is stochastic local search. Starting with Hill Climbing and

Simulated Annealing, these algorithms have also developed in biologically inspired population-based algorithms such as the Genetic Algorithm and Particle Swarm Optimization. Both GA and PSO, however, suffer from two curses. First, as the number of dimensions in the problem grow, the number of individuals in the population must theoretically increase exponentially. This is the curse of dimensionality. The second curse results as the dimensions of individuals increase. As individuals become larger, they begin to experience more and more hitchhiking.

A particular strand of research starting with CCGA and including CPSO, Hybrid CPSO, OSI and culminating in FEA, has sought to solve the hitchhiking problem. The general theme of these algorithms has been to introduce cooperation by decomposing the variables into factors and assigning populations to each factor. However, this solution introduces problems of its own, namely, pseudo-optima. Both Hybrid CPSO and FEA seek to eliminate pseudo-minima. FEA accomplishes this through overlapping factors.

In the following chapters we will build on this background bringing in additional related work as needed. In Chapter 3, we will develop a distributed version of FEAs, Distributed FEAs. In Chapter 4, we will re-examine the cooperation versus competition dichotomy and focus instead on information sharing and conflict resolution and present a revised version of DFEAs (Chapter 5). In Chapter 6, we will apply the insights of Chapter 4 on a single population PSO. Finally, in Chapter 8, we will

implement a PSO version of FEAs and DFEAs using the Actor model of concurrency [12].

Chapter 3

Distributed Factored Evolutionary Algorithms

In this chapter, we build on the previous research on FEA and develop a distributed version by generalizing the Distributed Overlapping Swarm Intelligence (DOSI) [13] algorithm. This Distributed Factored Evolutionary Algorithm (DFEA) replaces a shared, centralized context, C, for a distributed one. In DFEA, every subpopulation has its own context which is kept in sync with all the others. However, maintaining this *consensus* is expensive so we will also investigate what happens when we permit consensus to be relaxed.

3.1 Generalizing DOSI to DFEA

One of the downsides to FEA is that making the algorithm completely distributed is difficult because FEA relies on a single full global solution to be communicated between all of the factors (subpopulations). There has been work on developing distributed versions of OSI, called Distributed Overlapping Swarm Intelligence (DOSI), in which the algorithm no longer requires a single full global solution to be maintained between all factors [36, 13]. Instead, each factor in DOSI maintains its own full solution that is updated during a sharing step. While this allows for DOSI to be completely distributed, it can increase the runtime. This is because all previous work on DOSI required factors' full solutions to reach full consensus during sharing [36]. Depending on the problem and factor architecture, this can be computationally expensive.

In this chapter, we provide a generalization of DOSI, called Distributed Factored Evolutionary Algorithm (DFEA), that is similar to FEA's generalization of OSI. This allows for DFEA to use any optimization algorithm for the Optimize Step as with FEA. Our hypothesis is that DFEA will perform equally as well as FEA. We will test this hypothesis on a variety of optimization problems including abductive inference in Bayesian networks, maximizing NK landscapes, and minimizing benchmark test functions. We will use PSO as the "EA" in both algorithms and compare results to the global best PSO presented in the previous chapter.

We also investigate the effect of relaxing the degree of consensus between factors has on DFEA's performance using the same optimization problems and constituent algorithms. We hypothesize that there is a relationship between the amount of consensus required during the Sharing Step in DFEA and the degree of epitasis in the problem and that this relationship affects the solution quality. In these experiments, we will reduce the exchange steps between factors by varying amounts including full consensus, half consensus and a single exchange step.

3.2 DOSI

As previously discussed, FEA itself is related to previous research on OSI. The first version of OSI was introduced in 2012 by Haberman and Sheppard [2] as Particle-based Routing with Overlapping Swarms for Energy Efficiency (PROSE). PROSE was then adapted by Ganesan Pillai and Sheppard to learn the weights of deep artificial neural networks [14].

Fortier *et al.* used OSI for inference tasks in Bayesian networks, such as abductive inference, where the task is to find the most probable set of states for some nodes in the network given a set of observations [37, 36] Additionally, Fortier *et al.* used OSI for structural learning of Bayesian networks [34] and learning latent variables [35]. As discussed at length in the previous chapter, FEA was first introduced by Strasser

et al., which generalizes OSI so that any evolutionary algorithm can be used as the underlying optimization technique [9].

DOSI was first developed by Fortier *et al.* to learn weights on deep neural networks [13]. The key distinction from OSI is that a full global solution is not used for fitness evaluation. Instead, each subswarm maintains its own full personal solution, which allows for the algorithm to be distributed more effectively. A communication and sharing algorithm was defined so that subswarms could share values while also competing with one another. The authors were able to show that DOSI's performance was close to that of OSI's on several different networks, but there were several instances when OSI outperformed DOSI.

Similar to OSI, DOSI has been adapted to perform full and partial abductive inference in Bayesian networks. DOSI was found to be comparable to OSI on most problems and was only outperformed on large Bayesian networks or when the explanation sets are greater than four [36]. The authors also demonstrated that DOSI required more fitness evaluations than OSI [36].

3.3 Background: NK Landscapes, Bayesian Networks, and DMVPSO

Our experiments with DFEA require a set of test problems and appropriate component optimization algorithms. For the test problems we chose NK landscapes, abductive inference in Bayesian Networks, and some common benchmark optimization problems. NK landscapes were included because they represent commonly used functions for evaluating the performance of evolutionary and swarm algorithms. We included abductive inference in Bayesian Networks because they are a practical application of optimization. Additionally, Fortier *et al.* showed that OSI outperforms domain specific algorithms like approximate mini-bucket elimination on complex networks [36].

Like FEA, DFEA can use any optimization algorithm for the Update Step. Because this dissertation focuses on PSO variants, we will use PSO for our experiments. However, the version of PSO presented in the previous chapter is only suitable for continuous optimization problems. This means we can use that version of PSO for the benchmark optimization problems. However, the NK landscapes and Bayesian Networks are not continuously valued real functions but instead discrete (categorical) optimization problems. For these problem we will use a version of PSO proposed by Veeramchaneni *et al.* called Discrete Multi-Value Particle Swarm Optimization

(DMVPSO) for those problems [38].

The following sections go into more detail on both the test problems (NK Landscapes, Abductive Inference in Bayesian Networks) and DMVPSO. We will defer describing the benchmark functions until the experiment design section as they are familiar continuous real valued functions. The continuous version PSO was covered in the previous chapter.

3.3.1 NK Landscapes

The NK landscape is a mathematical framework that generates tunable fitness landscapes that are often used as test functions for evaluating EAs [39]. An NK landscape model contains two parameters, N and K, that control the overall size of the landscape and the structure or amount of interaction between each dimension, respectively [40].

An NK landscape is a function $f : \mathcal{B}^N \rightarrow \mathbb{R}^+$ where $\mathcal{B}^N$ is a bit string of length N. K specifies the number of other bits in the string on which a bit is dependent. This interaction is often referred to as epistasis. Given a landscape, the fitness value is calculated as

$$f(\mathbf{X}) = \frac{1}{N} \sum_{i=1}^{N} f_i(X_i, nb_K(X_i))$$

where $nb_K(X_i)$ returns the K bits that are located within X_i's neighborhood. The

individual factors f_i are then defined as $f_i : \mathcal{B}^K \rightarrow \mathbb{R}^+$ and the values of f_i are generally created randomly.

There are multiple ways to define the neighborhood function. The simplest way is to return the next K contiguous bits of the string starting at X_i. If the end of the string is reached, then the neighborhood wraps back around to the beginning of the string. In other cases, the neighborhood of each bit is created randomly.

3.3.2 Bayesian Networks

A Bayesian network is a directed acyclic graph $G = (\mathbf{V}, \mathbf{E})$ that encodes a joint probability distribution over a set of random variables, where each variable can assume one of an arbitrary number of mutually exclusive values [41, 42]. In a Bayesian network, each random variable X_i is represented by a node, and edges between nodes in the network represent probabilistic relationships between the random variables. Each root node contains a prior probability distribution while each non-root node contains a probability distribution conditioned on the node's parents.

For any set of random variables in the network, the joint probability distribution can be represented using the local distributions in the network

$$P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i | \mathrm{Pa}(X_i)).$$

where $\text{Pa}(X_i)$ corresponds to the parents of X_i.

A node is conditionally independent of all other nodes in the network given its Markov Blanket. In a Bayesian network, the Markov blanket of a node consists of the node's parents, children, and children's parents.

A common type of query for Bayesian networks is the probability distribution of a variable given a set of evidence. Another type of query is called abductive inference, which finds the most probable state assignment $\mathbf{x}$ to the variables in $\mathbf{X}_U$ given the evidence $\mathbf{X}_O = \mathbf{x}_O$. This is also known as the Maximum *A Posteriori* (MAP) probability state of the variables of a network. In addition, users often ask for the top k hypotheses. When $k > 1$, this is often referred to as the k-Most Probable Explanation (k-MPE) problem.

3.3.3 Particle Swarm Optimization

The canonical global best PSO was described in detail in the previous chapter. We would only emphasize at this point that that version of PSO, developed by Kennedy and Eberhart, is geared to optimize real valued functions, $f : \mathbb{R}^n \rightarrow \mathbb{R}$ [6]. So while the PSO velocity update (Algorithm 2.4, Line 4) and position update (Algorithm 2.4, Line 5) have been shown to work well on optimization problems involving continuous variables, many real-world problems operate over a set of discrete variables.

Veeramachaneni *et al.* presented an algorithm that allows PSO to optimize dis-

crete multi-valued functions called Discrete Multi-Valued PSO (DMVPSO). In this algorithm, the velocity update equations remain mostly unchanged. However, the semantics of the velocity vector are changed to denote the probability of a particle's position term having a value $[0, M-1]$. The update to the position vector is also modified to take advantage of the new velocity vector semantics. Each dimension in the velocity vector is restricted to values in $[0, M-1]$, where M is the cardinality of the dimension. After the velocity is updated, it is mapped into a $[0, M-1]$ interval using the sigmoid function

$$S_{i,j} = \frac{M-1}{1+\exp(-V_{i,j})}.$$

Next, each particle's position is updated by generating a random number according to the Gaussian distribution, $X_{i,j} \sim N(S_{i,j}, \sigma \times (M-1))$ and rounding the result. Finally, the result is passed through the piecewise function

$$X_{i,j} = \begin{cases} M-1 & X_{i,j} > M-1 \\ 0 & X_{i,j} < 0 \\ X_{i,j} & \text{otherwise} \end{cases}$$

to ensure the values remains in the range $[0, M-1]$.

3.4 DFEA: Distributed FEA

DFEA is an extension of FEA that allows the algorithm to be distributed completely. In FEA, all of the subpopulations require access to the full central context $\mathbf{c}$ to evaluate candidate solutions. DFEA breaks this dependency by having each subpopulation maintain its own full local context. However, this change requires certain portions of FEA to be adapted to support competition and sharing between the distributed, local contexts instead of a single centralized one.

FEA takes a function $f : \mathbb{R}^n \to \mathbb{R}$ with parameters $\mathbf{X} = \langle X_1, X_2, \ldots, X_n \rangle$ and creates a set $\boldsymbol{\mathcal{X}}$ of factors, $\mathbf{X}_i$. Each factor is assigned to a "sub"-population, S_i, that will optimize that factor using the chosen evolutionary algorithm (as a shorthand we will often refer to S or S_i as a "factor" rather than the more verbose "the factor $\mathbf{X_i}$ of the subpopulation S_i"). Note that f can still be optimized over the factor S_i if we supply the remaining values $\mathbf{R}_i = \mathbf{X} \setminus \mathbf{S}_i$. When $|\boldsymbol{\mathcal{S}}| = s = 1$ and $S_1 = \mathbf{X}$, then $\boldsymbol{\mathcal{S}}$ will have just a single population that results in a traditional application of the population-based algorithm, such as PSO, Differential Evolution (DE) [43], or GA. However, when $s > 1$, $S_i = \mathbf{X}_i \in \boldsymbol{\mathcal{X}}$ for all factors, and $\bigcup \mathbf{S}_i = \mathbf{X}$ for all populations, the algorithm becomes a multi-population algorithm.

FEA is the case where there are factors that are proper subsets of $\mathbf{X}$ that may or may not overlap with one another. In this work, we look at problems where every factor overlaps with some other factor. Should there be a disjoint factor, we have a

family of FEA and DFEAs.

In FEA, all of the factors' remainder variables $\boldsymbol{R}_i$ are guaranteed to have identical state assignments because they were provided by the same context $\mathbf{c}$. However, in DFEA, the remainder variables $\mathbf{R}_i$ are not set by some global $\mathbf{c}$ but are, instead, ultimately assigned set by using S_i's neighbors' local contexts, $\mathbf{c}_j$.

This multitude of contexts presents an interesting challenge for DFEA. Because DFEA does not have a full global context $\mathbf{c}$ and more than one factor may be optimizing any given X_i, some factor has to be "in charge" and arbitrate the possibly conflicting values of X_i and communicate the selected one back out to the other factors. We must, therefore, designate some subpopulation, S, as the *arbiter* for variable X_i that performs the competition for the variable X_i. The arbiter's full local context is then used to evaluate other values during competition. Each arbiter node for X_i communicates directly with any subpopulation S_j that contains X_i, inducing a communication topology between the factors. We define this induced graph H as the DFEA's *communication graph*, where the nodes represent factors. An edge connects two nodes in H if and only if one of the nodes is an arbiter for a value that the other node also optimizes over. Note that two subpopulations S_i and S_j can overlap with one another but not communicate directly with one another. This occurs if S_i is the arbiter for variable X_i and $X_i \notin S_j$.

The DFEA algorithm is presented in Algorithm 3.9. Although this is a distributed

Algorithm 3.9 Distributed Factored Evolutionary Algorithm

Input: Function f, Evolutionary Algorithm ae
Output: Best context $\mathbf{c}$ as candidate solution $\mathbf{x}$

```
1:  𝒳 ← factorize(X)
2:  S ← ae.initialize(f, 𝒳)
3:  𝒞 ← initialize-contexts(S)
4:  𝒪 ← identify-optimizers(𝒳)
5:  𝒜 ← identify-arbiters(𝒳)
6:  repeat
7:      repeat
8:          for S in S do
9:              S ← ae.update(S)
10:         end for
11:     until stopping criteria
12:     𝒞 ← compete(f, S, 𝒪, 𝒜, 𝒞)
13:     share(f, S, A, C)
14: until stopping criteria
15: c ← select-best-context(f, 𝒞)
16: return c
```

algorithm in theory, the pseudocode shows a high level specification that does not include an actual parallel implementation. We will actually present such an implementation in Chapter **??**.

The differences between Algorithm 2.6 and Algorithm 3.9 are small but important. First, we are generating a context for each factor (Line 6). Second, we must select arbiters for each X_i (Line 5). The arbiters must be passed to the Compete Step (Line 12). And finally, as we will see later when we investigate the relaxing of consensus between contexts, we should pick the best context to return from the function (Line 15).

We now describe the Compete and Share Steps for DFEA that update the factors' full local contexts $\mathbf{R}_i \cup \mathbf{S}_i$ or $\mathbf{C}_i$.

3.4.1 Compete Step

The purpose of the Compete Step in DFEA is basically the same as that in FEA. We must determine which factor has the best value for every dimension and resolve any conflicts all the optimizers of X_i (neighbors in the current scheme) may have about a variable's value. The wrinkle again is that for FEA, competition is held by the full global context $\boldsymbol{C}$. For DFEA, the competition is held by each variable's *arbiter*. Here we present a general DFEA competition algorithm based on the work done by Fortier *et al.* [36] in Algorithm 3.10.

The DFEA Compete Step works as follows. First, in Line 2 the arbiter for X_i is selected and then used to obtain the context corresponding to that factor. After initializing the comparison variables in lines 3-4, the algorithm iterates over the variables of the problem. As before, we loop over the optimizers of X_i (Lines 5-12) substituting X_i from each factor's best candidate into the context. The only difference in this version—as compared to the FEA version—is that this is the arbiter's context rather than a single, global one. After X_i is arbitrated, the new value is communicated to all of the optimizers of X_i (Lines 14-16).

DFEA's Compete Step only relies on the arbiter factor for X_i and the factors that

Algorithm 3.10 DFEA Compete

Input: Objective function $f(x)$, subpopulations $\mathbf{S}$, optimizers $\boldsymbol{\mathcal{O}}$, arbiters $\boldsymbol{\mathcal{A}}$, contexts $\mathbf{C}$
Output: New contexts $\boldsymbol{\mathcal{C}}$

```
1: for i = 1 to d do
2:     c ← C[A[i]]
3:     fitness ← f(c)
4:     value ← c[i]
5:     for j in O[i] do
6:         candidate ← S[j].best
7:         c[i] ← candidate.x[i]
8:         if f(c) ≤ fitness then
9:             value ← candidate.x[i]
10:            fitness ← f(c)
11:        end if
12:    end for
13:    c[i] ← value
14:    for j in O[i] do
15:        C[j].c[i] ← c[i]
16:    end for
17: end for
18: return C
```

also optimize X_i. Note that the competition algorithm is not guaranteed to find the best combination of values from the factors.

3.4.2 Share Step

In DFEA, the Share Step's purpose is to let factors distribute information to one another and is much more important than the mere bookkeeping that occurs in the FEA's version. Information distribution is accomplished by having two neighboring

Algorithm 3.11 DFEA Share

Input: Factors $\mathcal{S}$

```
for k = 1 to W do
    for all S_i ∈ S and S_j ∈ neighbors(S_i) do
        Exchange(S_i, S_j)
    end for
end for
return
```

factors in the communication graph H exchange information about their full personal solutions. However, neighboring factors need to be augmented with additional information in order for the factors to know which values to share with one another. This is accomplished by having each factor maintains a δ-map.

For each dimension i, the δ-map stores the minimum number of steps required to reach a factor learning X_i, where a step can occur only between neighboring factors. For example, if factor S_i learns dimension j, then $S_i.\delta_j = 0$. If S_i does not learn dimension j but neighbors a factor that does, then $S_i.\delta_j = 1$. Let $d_H(S_i, S_j)$ denote the distance or the minimum number of hops between nodes corresponding to S_i and S_j in the graph H. Then $S_i.\delta_k = \min\{d_G(S_i, S_j)|S_j \text{ knows} X_k\}$. We say that the factors reach *consensus* when they all agree on all state assignments.

Initially, $S_i.\delta_j = 0$ if S_i optimizes X_j and $S_i.\delta_j = \infty$ otherwise. We say that factor S_i knows dimension X_k once $S_i.\delta_k < \infty$. The full share and exchange algorithms for DFEA are shown in Algorithm 3.11 and 3.12, respectively.

The Share algorithm operates as follows. For W iterations, the algorithm iterates

Algorithm 3.12 DFEA Exchange

Input: Factors S_i, S_j

1: **for** $k = 1$ to n **do**
2: **if** $S_i.\boldsymbol{\delta}_k > S_j.\boldsymbol{\delta}_k$ **then**
3: $S_i.\boldsymbol{\delta}[k] \leftarrow S_j.\boldsymbol{\delta}[k] + 1$
4: $S_i.\mathbf{c}[k] \leftarrow S_j.\mathbf{c}[k]$
5: **else if** $S_i.\boldsymbol{\delta}_k < S_j.\boldsymbol{\delta}_k$ **then**
6: $S_j.\boldsymbol{\delta}[k] \leftarrow S_i.\boldsymbol{\delta}[k] + 1$
7: $S_j.\mathbf{c}[k] \leftarrow S_i.\mathbf{c}[k]$
8: **end if**
9: **end for**
10: **return**

over all neighboring pairs of factors in H and calls the exchange algorithm for those two factors. During the Exchange algorithm, all of the n dimensions in $\mathbf{X}$ are iterated over. In lines 2-5, the algorithm compares the δ values of the two factors for the current dimension k. If the δ_k value for factors S_j is lower than δ_k from S_i, then the value from S_j is inserted into S_i. In addition, δ_k for S_i is updated according to Line 3. Lines 5–8 do the same thing except that information is shared from S_i to S_j. Note that $S_i.\mathbf{c}[\,]$ is the factor's full local solution (context) and is equal to $\mathbf{x}_i \bigcup \mathbf{r}_i$.

3.4.3 Complexity of DFEA

In order to analyze the complexity of DFEA, we build on the work of Strasser [32], which established the algorithmic complexity of FEA. As the main differences between the FEA and DFEA are the Compete and Share Steps, the general complexity analysis of fitness evaluations and the underlying evolutionary algorithms as

well as the complexity analysis of the Update Step still apply to DFEA. However, in terms of algorithmic complexity, only the Share Step is more complex.

The complexity of fitness evaluation is taken to be a function of the dimensionality of the problem, $d = |\mathbf{X}|$. We define $\Lambda(d)$ to be a problem specific function that takes the dimensionality of the problem and returns the cost of fitness evaluation. Therefore the complexity of a single fitness evaluation is $\mathcal{O}(\Lambda(d))$.

The underlying evolutionary algorithm (EA) has a complexity that depends both on the cost of fitness evaluations and the cost of updating the individuals. We define a function $U(d)$ that returns the cost of updating an individual of dimension d for the particular EA we are using. This includes the fitness evaluation. In general, most algorithms will update each individual and evaluate its fitness so the total cost for an individual is $\mathcal{O}(U(d)) = \mathcal{O}(d + \Lambda(d))$. The total cost for a population of size $p = |S|$ is $\mathcal{O}(pU(d))$.

Strasser established the complexity of FEA's Update, Compete and Share steps as follows:

- **Update Step**: $\mathcal{O}(skpU(d))$

  Rationale: Let $s = |\mathbf{S}|$, the number of factors/subpopulations, and k be the number of Update iterations (specified as "stopping criteria" in the pseudocode, then our previous single iteration complexity of the underlying EA is $\mathcal{O}(pU(d))$ and it is repeated $s \times k$ times.

- **Compete Step**: $\mathcal{O}(nd\Lambda(d))$

  Rationale: Let n be the number of optimizers for X_i, then there are $n \times d$ fitness evaluations required to establish the new value of the global context, $\mathbf{c}$.

- **Share Step**: $\mathcal{O}(sd)$

  Rationale: For each swarm/factor, s, the Share Step replaces the worst individual in the population with values from the global context, $\mathbf{c}$. Replacing the worst individual requires iterating over the d variables in the worst individual's position and setting each value to the corresponding value in $\mathbf{c}$.

If there are m total iterations of the FEA (Compete and Share Step), then the total complexity of FEA is:

$$\mathcal{O}(FEA) = \mathcal{O}(mskpU(d)) + mnd\Lambda(d) + msd)$$

We can do a similar analysis for DFEA. As we previously mentioned, all steps except the Share Step are the same in terms of algorithmic complexity.

- **Share Step**: $\mathcal{O}(dWs^2)$

  Rationale: For the Share step, the algorithm has to iterate over all pairs of neighboring factors and share values between the two factors. Iterating over all pairs has a complexity of $\mathcal{O}(s^2)$. During each pairwise Share Step, the algorithm

iterates over every variable and compares the factors' δ values. Because the algorithm iterates over d dimensions s^2 times, the complexity of each share iteration is $\mathcal{O}(ds^2)$. This step must is repeated W times such that all factors reach consensus, giving a total complexity of $\mathcal{O}(dWs^2)$.

The total algorithmic complexity is thus:

$$\mathcal{O}(DFEA) = \mathcal{O}(mskpU(d)) + mnd\Lambda(d) + mdWs^2)$$

as with FEA, for DFEA, any of the Solve, Compete, and Share Steps can dominate. When the number of update iterations performed on each subpopulation k and the number of individuals p is very large, the dominating step in DFEA will be Solve. When k is small and the number of individuals p for each factor is smaller than d, the dominating step in DFEA will be either Compete or Share. For Compete to be the dominating step, the number of sharing iterations W must be small. However, in several applications, the authors required DFEA to reach full consensus. To do so, W must be equal to the diameter of the factor communication network H. In the worst case this requires $W = s$, meaning the computation complexity of Share is $\mathcal{O}(ms^4)$; therefore, in many applications, the dominating step will be the Share Step.

3.5 Comparison of DFEA to FEA with Full and Relaxed Consensus

One of the defining differences between FEA and DFEA is the necessity of the Share Step. Each subpopulation (factor) in DFEA has a local rather than central view of a current best-so-far solution in the form of the context, $\mathbf{c}_i$. The Share Step is thus necessary to exchange information between subpopulations in order for them to reach consensus. Depending on the size and complexity of the communication topology, this can increase the runtime of DFEA significantly. In order to reduce this runtime, we investigate the effect that relaxing the amount of consensus between factors has on DFEA's performance. To test this, we relax the number of sharing iterations W that are used in DFEA, which in turn causes the subpopulations to reach a lower level of consensus. We also applied DFEA and consensus relaxation to several general optimization problems which had not been done before.

3.5.1 Design

To test our hypothesis, we created three different versions of DFEA: DFEA-1, DFEA-1/2, and DFEA-Full. DFEA-1 used only 1 sharing iteration during the Share algorithm while DFEA-1/2 used $Round(D/2)$ sharing iterations, where D is the diam-

eter of the graph induced by the communication topology. DFEA-Full ran D sharing iterations. The diameter of the graph represents the maximum distance between an arbiter and any subpopulation. D is thus the most number of exchanges we need to make to move a new value from an arbiter to any subpopulation. We also ran FEA for an additional comparison. All FEA and DFEA versions ran for 10 inter-swarm optimization iterations since this value was found by Strasser *et al.* [9] to allow FEA to converge.

For our experiments we used three sets of problems: maximizing NK landscapes, performing abductive inference on Bayesian networks, and optimizing several standard benchmark functions.

For the NK landscapes and abductive inference, FEA and DFEA used DMVPSO as the underlying optimization algorithm. On the benchmark problems, we used canonical PSO. For both PSOs, the ω parameter was set to 0.729, and ϕ_1 and ϕ_2 were both set to 1.49618.

We applied these same versions of PSO, FEA, and DFEA, as well as the relaxed versions of DFEA to the benchmark optimization problems. The individual and component PSO parameters were the same. The individual PSO was run for 100 iterations with population sizes of 10 times the dimensions. The FEA and DFEA were run for 20 Compete-and-Share iterations with the component PSOs running for 5 iterations. This gives single swarm algorithms the same total number of iterations

as the factors in DFEA. Each factor for FEA and DFEA had a population size of 10, which gives FEA and DFEA the same number of individuals as the single swarm algorithms.

3.5.1.1 NK Landscapes

We generated NK landscapes with parameters $N = 25$ and 40 and $K = 2, 5$ and 10. For each set of parameters, we created 30 random landscapes.

In applying DFEA to NK landscapes, we used the Neighborhood architecture proposed by Strasser *et al.* [9] since it outperformed competing factor architecture approaches. In general, the Neighborhood architecture controls how factors communicate during the Share Step. This particular Neighborhood architecture creates a factor for each variable X_i and adds to the factor variable X_i and all variables in the set $nb_K(X_i)$. This results in factors of size $K + 1$.

3.5.1.2 Bayesian Networks

For abductive inference on Bayesian networks, we used the Hailfinder, Hepar2, Insurance, and Win95pts Bayesian networks from the Bayesian Network Repository [44] (Table 3.1). These networks were chosen to be consistent with [36]. To evaluate the fitness of a state assignment we used the log likelihood ℓ, which is calculated

Table 3.1: *Bayesian Network Characteristics*

Network	Nodes	Arcs	Parameters	Avg. MB Size
Hailfinder (Ha)	56	66	2656	3.54
Hepar2 (He)	70	123	1453	4.51
Insurance (I)	27	52	984	5.19
Win95pts (W)	76	112	574	5.92

$$\ell(\mathbf{x}) = \sum_{i=1}^{n} \log P(x_i|\mathrm{Pa}(x_i))$$

where $\mathbf{x} = \{x_1, x_2...x_n\}$ is a complete state assignment and $\mathrm{Pa}(x_i)$ corresponds to the assignments for the parents of X_i.

The factor architecture chosen was the Markov architecture proposed by Fortier *et al.* since this was shown to outperform all other architectures on Bayesian networks [36, 9]. This uses the Markov blanket of every node to create subpopulations, because it offers one of the most natural ways to subdivide a Bayesian network and provide overlap. Additionally, it gives the algorithm an advantage because every node in the network is conditionally independent of all other nodes when conditioned on its Markov blanket. For our experiments we used an empty evidence set to keep results comparable.

3.5.1.3 Benchmark Optimization Problems

We picked a variety of benchmark optimization problems: Sphere, Exponential, Schwefel 1.2, Dixon-Price, Ackley's, Rosenbrock, and Griewank [18] (See Appendix A more information the benchmark functions). All of the problems are minimization problems with global minima at 0.0 except for the Exponential which has a minimum at –1.0. All of the problems are scalable, meaning they can be optimized for versions of any dimension. The Sphere function is separable. The remaining functions are non-separable with most functions depending on adjacent, overlapping dimensions such as X_i and X_{i+1}. Because of this, we used a factor size of two for all of the benchmark optimization problems.

3.5.2 Results

Table 3.2 shows the results comparing FEA and all versions of DFEA on performing abductive inference on Bayesian networks and maximizing NK landscapes. Note that these are maximization problems. Results comparing PSO, FEA, and DFEA on minimizing the benchmark functions are in Table 3.3 while the results comparing the different versions of DFEA on the benchmark functions are in Table 3.4. All results are expressed as means over 30 trials with standard errors in parentheses.

In the Bayesian network problems, there are only small differences between all

versions of DFEA. In most cases, DFEA-Full performs the best. DFEA-1/2 performs better than DFEA-1 and DFEA-Full only on the Win95pts network. In the Insurance network, DFEA-1 performs better than DFEA-Full, but only by a small margin. On all networks, all DFEA algorithms are competitive with FEA.

For the NK-landscape results, DFEA-1 almost always performed worse than the other DFEA algorithms. In some landscapes, such as $N = 25$ and $K = 2$, DFEA-1/2 performs better than DFEA-Full but for when $N = 25$ and $K = 10$, DFEA-1/2 performs slightly worse than DFEA-Full. When $N = 40$ and $K = 2, 10$, DFEA-1/2 outperforms DFEA-Full, but when $N = 40$ and $K = 5$, DFEA-Full outperforms DFEA-1/2.

On the benchmark optimization problems, DFEA-Full outperformed PSO except for the Schwefel 1.2 function. Overall, DFEA was slightly worse than FEA on all the benchmark problems. When looking at the consensus results in Table 3.4 relaxation results are presented. DFEA-Full performed better that DFEA-1 and DFEA-1/2 on Sphere, Exponential, Dixon-Price, and Ackley's. However, DFEA-1 performed the best on Rosenbrock while DFEA-1/2 performed the best on Griewank. DFEA-1/2 outperformed DFEA-1 on all functions except for Schwefel and Rosenbrock, but was outperformed by DFEA-Full except on Schwefel and Griewank.

Table 3.2: *Results from varying the amount of consensus between factors.*

			Diameter	PSO	FEA	DFEA-1	DFEA-1/2	DFEA-Full
Bayesian	Hailfinder		8	−86.94 (23.70)	−33.85 (0.46)	−38.70 (0.49)	−38.58 (0.58)	−36.86 (0.66)
	Hepar2		5	−49.47 (0.45)	−16.85 (0.34)	−20.05 (1.13)	−21.03 (1.01)	−19.62 (1.16)
	Insurance		5	−23.83 (0.24)	−11.22 (0.29)	−12.76 (0.59)	−13.75 (0.48)	−12.78 (0.44)
	Win95pts		5	−90.41 (1.65)	−16.41 (1.34)	−29.86 (1.28)	−28.50 (2.00)	−31.16 (2.07)
NKs	$N = 25$	$K = 2$	12	17.96 (0.02)	18.56 (0.09)	17.69 (0.09)	18.06 (0.10)	18.00 (0.09)
		$K = 5$	5	18.23 (0.02)	19.23 (0.05)	18.38 (0.06)	18.46 (0.06)	18.23 (0.07)
		$K = 10$	3	18.21 (0.01)	18.99 (0.04)	18.27 (0.04)	18.38 (0.06)	18.40 (0.05)
	$N = 40$	$K = 2$	20	26.88 (0.03)	29.55 (0.10)	26.58 (0.11)	28.91 (0.11)	28.73 (0.11)
		$K = 5$	8	27.43 (0.05)	30.85 (0.07)	28.37 (0.09)	29.21 (0.11)	29.55 (0.10)
		$K = 10$	4	27.54 (0.05)	30.53 (0.06)	27.47 (0.06)	28.17 (0.08)	28.75 (0.09)

Table 3.3: *Benchmark Problem results for PSO, FEA, and DFEA*

		PSO	FEA	DFEA-Full
Benchmarks	Sphere	$3.8E+00$ $(1.7E-01)$	$1.7E-09$ $(1.5E-10)$	$1.5E-09$ $(1.4E-10)$
	Exponential	$-1.00E+00$ $(3.7E-05)$	$-1.0E+00$ $(1.7E-09)$	$-1.0E+00$ $(0.0E+00)$
	Schwefel	$4.4E+03$ $(9.6E+01)$	$1.8E+04$ $(1.2E+03)$	$2.5E+05$ $(6.3E+03)$
	Dixon-Price	$2.1E+01$ $(7.4E-01)$	$5.6E-01$ $(1.8E-01)$	$1.2E+00$ $(1.7E-01)$
	Ackley's	$2.2E+00$ $(4.5E-02)$	$1.9E-05$ $(8.4E-07)$	$2.3E-05$ $(1.9E-06)$
	Rosenbrock	$1.7E+02$ $(6.5E+00)$	$5.6E+00$ $(1.1E+00)$	$6.8E+00$ $(2.5E+00)$
	Griewank	$6.3E-01$ $(2.3E-02)$	$2.6E-03$ $(1.1E-03)$	$8.6E-02$ $(2.5E-02)$

3.6 Discussion of Experimental Results

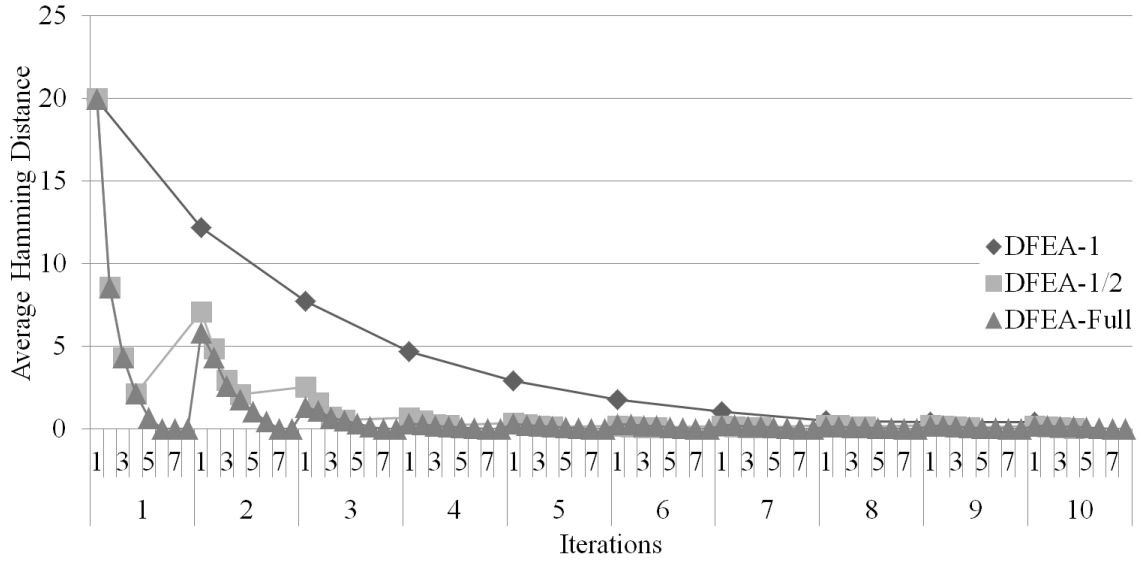
Based on the Bayesian network results, we can see that DFEA does not need to reach full consensus during each Share Step in order to find quality solutions. To investigate this, we looked at the the average Hamming distance between DFEA's factors on the Hailfinder Bayesian network (Figure 3.1). We also looked at fitness curves (Figure 3.2). In Figure 3.1, the major X-axis on the chart is the inter-factor optimization iterations while the minor X-axis is the number of sharing iterations for the different DFEA versions. The X-axis in Figure 3.2 is the inter-factor iteration.

Based on the charts, one can see that in DFEA-1 and DFEA-1/2, the factors

Table 3.4: *Benchmark problem results with varying degrees of DFEA consensus*

		DFEA-1	DFEA-1/2	DFEA-Full
Benchmarks	Sphere	$2.2E-09$ $(2.4E-10)$	$3.1E-09$ $(4.5E-10)$	$1.5E-09$ $(1.4E-10)$
	Exponential	$-4.7E-05$ $(3.6E-06)$	$-1.0E+00$ $(2.1E-09)$	$-1.0E+00$ $(0.0E+00)$
	Schwefel	$2.2E+05$ $(4.9E+03)$	$2.4E+05$ $(5.6E+03)$	$2.5E+05$ $(6.3E+03)$
	Dixon-Price	$9.6E+00$ $(1.8E+00)$	$1.2E+00$ $(1.9E-01)$	$1.2E+00$ $(1.7E-01)$
	Ackley's	$6.9E+00$ $(1.2E-01)$	$3.1E-05$ $(1.6E-06)$	$2.3E-05$ $(1.9E-06)$
	Rosenbrock	$4.8E+00$ $(2.7E-01)$	$7.5E+00$ $(2.6E+00)$	$6.8E+00$ $(2.5E+00)$
	Griewank	$4.7E-02$ $(5.4E-03)$	$4.1E-02$ $(1.7E-02)$	$8.6E-02$ $(2.5E-02)$

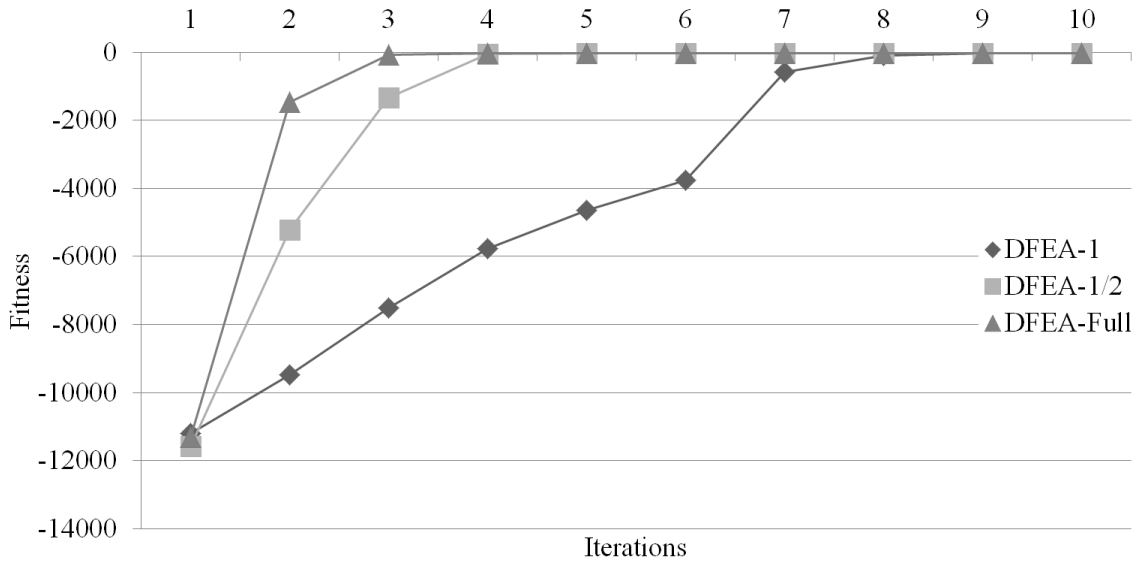
Figure 3.1: *Average consensus between factors over time of DFEA performing abductive inference on the Hailfinder Network.*



are still able to reach consensus over the lifetime of the algorithms because they all eventually reach a Hamming distance of zero. We believe this is because when optimizers start converging in their search spaces, the number of values changed during the exchange step decreases and therefore, the factors are able to reach consensus over several DFEA iterations.

We performed a similar analysis for NK landscapes $N = 25$ and $K = 10$. However,

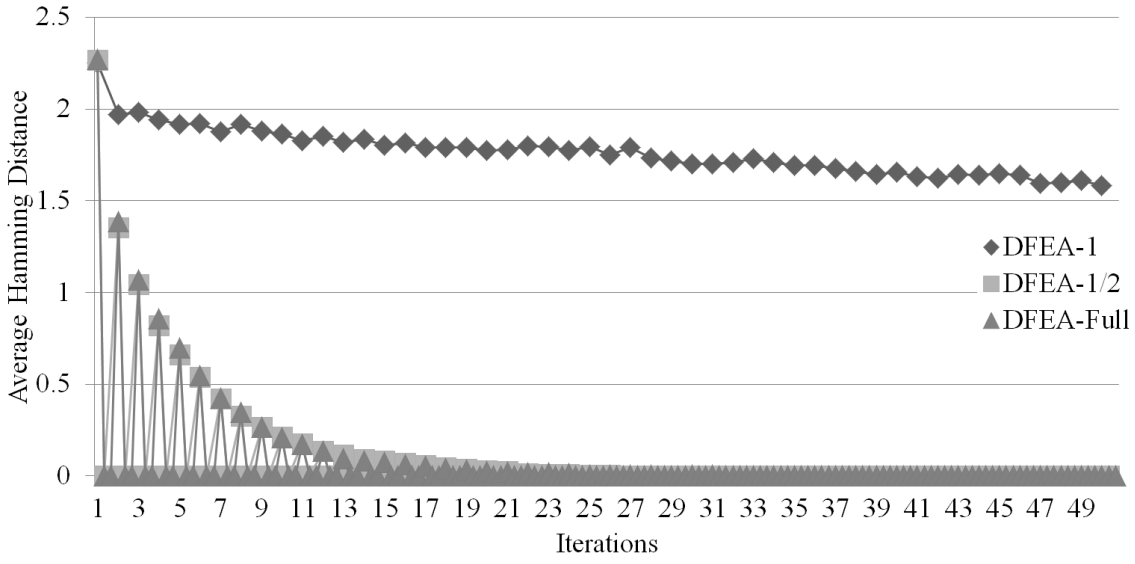
Figure 3.2: *Fitness over time of DFEA performing abductive inference on Hailfinder Network.*



we set the number of inter-factor iterations to 50. Figures 3.3 and 3.4 show the consensus and fitness graphs for NK landscapes $N = 25$ and $K = 10$.

For NK landscapes $N = 25$ and $K = 10$, DFEA-1 reaches consensus at a much slower rate than DFEA-1/2 and DFEA-Full. Meanwhile, DFEA-1/2 reaches consensus and fitness at about the same rate as DFEA-Full. DFEA-1 may be able to reach the same fitness as DFEA-1/2 and DFEA-Full in more iterations, but the cost of needing more iterations greatly outweighs the reduction in runtime by only having 1 sharing step. This appears to be the case where there is high epistasis in the problems, like on NK landscapes when $K = 5, 10$. When there is high epistasis, the solve and competition steps increase the differences between factors. This necessitates the need for more sharing iterations in order to reduce the difference. When there is

Figure 3.3: *Average consensus between factors in DFEA on maximizing NK Landscapes* $N = 25$ *and* $K = 10$.

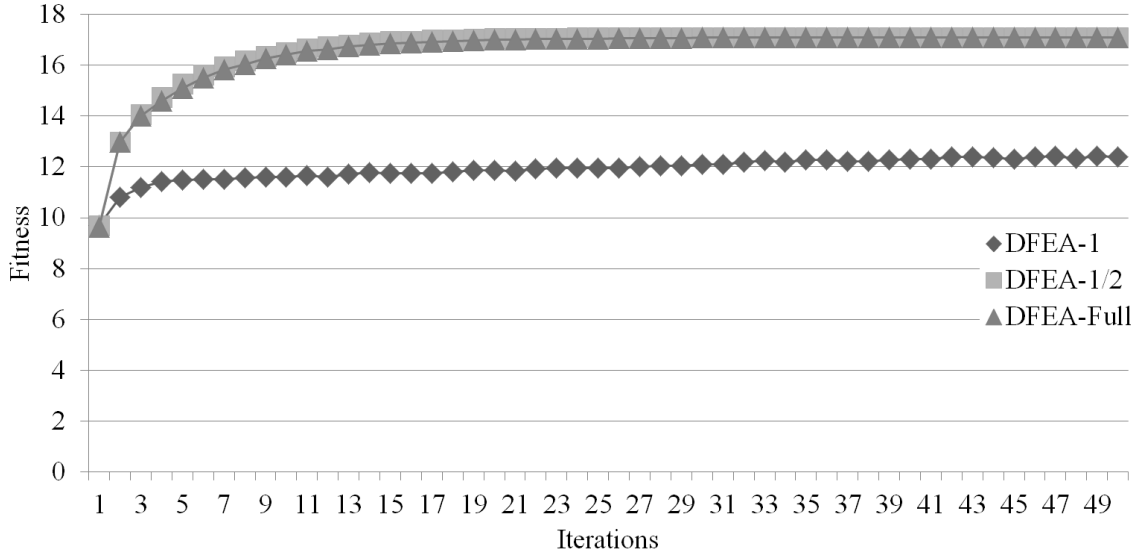


low epistasis, the increase in the factors' difference during the solve and competition steps in DFEA is small enough that only 1 sharing iteration is enough to reduce the difference between factors.

The results for the benchmark optimization problems are interesting. DFEA outperformed PSO on all of the problems with results that are often several orders of magnitude better than PSO. For example, on Ackley's function, PSO achieved a mean minimum of 2.2E+00 whereas DFEA achieved a mean minimum of 2.3E-05. The one exception is the Schwefel 1.2 function where all three algorithms performed poorly. In general, DFEA results were the same or slightly worse than the FEA results.

As previously mentioned, DFEA-Full performed better on Sphere, Exponential, Dixon-Price, and Ackley's while DFEA-1 performed better on Rosenbrock and Griewank.

Figure 3.4: *Fitness of DFEA on maximizing NK Landscapes* $N = 25$ *and* $K = 10$.



It is not altogether clear why this would be the case because we would generally expect consensus to be more important on harder problems. Rosenbrock and Griewank are generally considered to be harder problems than Sphere or Exponential. The pattern for DFEA-1/2 is harder to summarize. It lies outside the DFEA-1 to DFEA-Full range on several problems (Sphere and Griewank), is sometimes closer to the winner (Exponential, Dixon-Price, Ackley's, Griewank) but is also sometimes closer to the loser (Rosenbrock).

3.7 Conclusions

Just as FEA generalized OSI, we have generalized DOSI into the Distributed Factored Evolutionary Algorithms, DFEA, which like FEA, can use any constituent optimization algorithm and, which like DOSI, does not rely on a global context. We have shown that DFEA performs similarly to FEA but not always identically.

We also examined the possibility of relaxing consensus in DFEA. We have demonstrated that in problems with low epistasis, DFEA is able to perform well when using a reduced number of sharing iterations. However, for problems with high epistasis, DFEA performs worse with less sharing iterations. This drop in performance can be combated by performing more inter-factor iterations, but may negate the complexity reduction gained when the sharing iterations are reduced.

In this version of DFEA, we explored one possible interpretation of *distributed* by emphasizing distributed, local state instead of centralized global state. In a later chapter we will investigate both parallelism and asynchrony as other possible interpretations of "distributed." However, we first turn towards the observation that FEA and DFEA did not always perform similarly and note that this was often true of OSI and DOSI as well.

Chapter 4

Information Exchange and Conflict Resolution

In this chapter, we re-examine a recurring theme in multi-population approaches to solving the problem of hitchhiking: cooperation versus competition. In its place, we develop an alternative framework based on the information exchange via a *blackboard* architecture and Pareto efficiency as a standard for conflict resolution. We then apply this framework to Factored Evolutionary Algorithms. In later chapters we will explore the implications of this framework for DFEA and PSO.

4.1 Cooperation and Competition

As we saw in Chapter 2, competition and cooperation are among the big themes in many biologically inspired optimization algorithms. This is especially true for the Genetic Algorithm (GA), based as it is on "Survival of the Fittest." However, it often is just as true for kindred population-based algorithms such as Particle Swarm Optimization (PSO). As previously discussed, improvements in these algorithms have often come from varying the degrees of cooperation and competition. Potter and de Jong [7] developed the Cooperative Coevolutionary Genetic Algorithm (CCGA) to combat hitchhiking in the GA by using multiple populations or "subspecies" that cooperated by focusing on the individual variables of an optimization problem. This line of research also includes van den Bergh and Engelbrecht's CPSO [8], Fortier *et al.*'s OSI and DOSI [36], [13], Strasser *et al.*'s Factored Evolutionary Algorithms (FEA) [9], discussed in Chapter 2, and our own DFEA introduced in the last chapter.

The imagery is very powerful; van den Bergh and Engelbrecht state, "Although competition among individual humans usually improves their performance, much greater improvements can be obtained through cooperation." [8]. Even in their Hybrid CPSO they emphasize the cooperation between the CPSO and PSO steps rather than the necessity of re-introducing competition, in the form of a PSO step, in order to escape pseudo-optima. Strasser *et al.* [9] emphasizes competition without mentioning cooperation at all. In contrast, Strasser [32] includes "Cooperative" in

the title. So while the trade off between competition and cooperation has been a major theme of this research, we believe a different perspective could increase our understanding of these algorithms.

Van den Burgh and Engelbrecht refer to Clearwater *et al.*'s blackboard architecture and cooperating agents [45] as informing their thinking about their own algorithms, CPSO and Hybrid CPSO [8]. For Clearwater *et al.*, any agents that exchange information through a blackboard structure are deemed to be cooperative. Those that do not are not-cooperative.

For example, even the *population-based* Hill Climbing algorithm that we alluded to in Chapter 2 would be considered non-cooperative by Clearwater because the individual Hill Climbers do not share information with each other. The mere existence of a population of agents is not sufficient to make them cooperative. In contrast, the multi-population algorithms, such as FEA, that we have been discussing in this dissertation are clearly cooperative in Clearwater's sense. What we suggest here is that looking at how the blackboard is used, either implicitly or explicitly, might provide a better framework for thinking about these multi-population algorithms, analyzing their execution, making improvements and designing new algorithms.

4.2 Blackboard Architecture

In Clearwater *et al.* [45] the focus is on agents that are trying to solve cryptarithmetic problems such as $SEND + MORE = MONEY$. As agents search the solution space they can read hints from the blackboard and write hints based on the new states explored. For our purposes, there are several interesting characteristics of how the blackboard is used.

First, a blackboard uses an explicit structure or shared state to store information. Second, there is no sense of conflicting information; there can be multiple hints that cover the assignment of, say, S, D, and M. A hint might be something like $\{S = 3, D = 4, M = 7\}$ Second, multiple hints may contain different and contradictory information about the solution, but any hint an agent selects (usually at random) is internally consistent. Again, there may be a hint like the one above and another hint $\{S = 2, D = 5, M = 1, N = 3\}$ on the blackboard. These hints are not consistent for S but this is acceptable as long as there is no hint that is internally inconsistent as in $\{S = 2, S = 7\}$.

Corkill *et al.* [46] take a more general look at blackboard architectures and how they can be designed for flexibility, efficiency, and generality. They define four operations that blackboard architectures usually implement: insertion, merging, retrieval, and deletion. Insertion covers the placing of new information about an entity on the blackboard. Merging involves the reconciliation of information about identical

entities. Retrieval is about obtaining information about an entity. Deletion involves removing information about an entity from the blackboard which is no longer required. Of course, the nature of these operations can differ widely depending on the actual blackboard implementation, whether it is a relational database or an in-memory HashMap. The actual operators will often be problem specific.

4.2.1 Information Exchange

For the algorithms we are discussing, the central concept is information exchange via a blackboard between multiple populations and the definition of the various operations (as applicable). We start with the nature of the blackboard structure itself.

Interestingly, in CCGA [7] the global solution, $\mathbf{c}$ (or *context* as we have been calling it in FEA), is *implicit*. Using FEA notation, $\mathbf{c}$ is the concatenation of the $S_i.best$ of all the subpopulations where $S_i.best$ is the best individual of the subpopulation. If we wish to evaluate some $\mathbf{x}_i$ in subpopulation S_i, we must assemble the remaining values $\mathbf{r}_i$ from all the other subpopulations. For the canonical GA, this can present some problems. First, the fitness of the best individual in the canonical GA can go up or down because the actual best individual is just a member of the population subject to selection, crossover and mutation. Second, while we can always preserve the best individual at every generation using *elitism*, elitism itself is not without problems as it often causes premature convergence [24]. Overall, this means that the global

solution is not guaranteed to improve throughout the course of the algorithm.

Although van den Burgh and Engelbrecht cite Clearwater *et al.* [45], their global solution, $\mathbf{c}$, is actually implicit as well. Defined as a function, $\mathbf{b}()$, the global solution is a concatenation of the global bests of all the subpopulations (swarms). As with CCGA, for any particular swarm S_i optimizing $\mathbf{x}_i$, $\mathbf{b}(\mathbf{x}_i, \mathbf{r}_i)$ returns a vector concatenating the relevant swarm's $\mathbf{x}_i$ and the remaining values that S_i requires. This represents a full solution that can then be evaluated by f.

The main difference between CCGA and CPSO in this regard is that the canonical *gbest* PSO is monotonically non-increasing (for minimization) in the fitness of the *gbest*. That is, the current *gbest* really is the best position observed so far throughout the run of the algorithm and may not match the position of any particle in the current swarm. And if a better candidate is never found, this *gbest* does not change and does not get lost. In a sense, a form of elitism is built into PSO but does not appear to wreak the same havoc as in GA.

It is not until OSI [14] that the global solution appears as a recognizable *state* which subpopulations use to read and write information, a blackboard architecture for information exchange. This is further developed by Fortier *et al.* in OSI and DOSI (for example, [37], [35]) and Strasser *et al.* [9]. In the previous chapter we saw a distributed version of the blackboard developed for DFEA. In the following discussion we will concentrate on FEA. The application of this framework to DFEA

will be covered in the next chapter.

In FEA, the insertion operation is straight forward. At the start of the algorithm, the blackboard as $\mathbf{c}$ is initialized into a random state. Retrieval is equally straight forward. At least in the current implementations and algorithms, $\mathbf{c}$ is an in-memory array or List which can be accessed as needed to provide R_i to any given subpopulation S_i. As mentioned previously, FEA does not use the delete operation on $\mathbf{c}$ because we always require a value in $\mathbf{c}$ for each X_i in $\mathbf{X}$.

For our particular use case, the most important operation of the four for FEA is, arguably, the merge operation. Using Corkill *et al.*'s [46] concept of operators, Clearwater *et al.*'s cooperative agents have a merge operation that basically says "post all hints" [45]. We can contrast that merge operator with Corkill *et al.*'s example where when there are two rules in a knowledge system that cover the same entity, the merge operation is to generalize the two rules to cover both antecedents. In FEA, we only keep a single value, c_i, for each variable on the blackboard, $\mathbf{c}$. Thus, given an existing value c_i and possibly many potentially better values, x_i, we require a merge operation that handles conflict resolution.

We note that one could imagine many different merge schemes for accomplishing conflict resolution. We submit that this is one of the chief benefits to using this framework rather than appealing to a tension between sometimes vague concepts of cooperation and competition.

4.2.2 Conflict Resolution

Looking at FEA, we have a global context $\mathbf{c}$ that operates as a blackboard through which subpopulations exchange information. They are able to read from $\mathbf{c}$ (retrieval) in order to fill in the missing values of $\mathbf{x}_i$ ($\mathbf{r}_i$ or remaining values) that make the optimization of a factored problem by individual subpopulations possible. In FEA, we see the "read" during the Share Step.

After the Optimize Step is completed, we want to update the blackboard, $\mathbf{c}$, with any new and better information that has been discovered by the individual subpopulations. But we do not want to accept arbitrary writes to $\mathbf{c}$ from the subpopulations. This is the essence of the Compete Step in FEA.

Each optimizer of X_i wishes to write its value to the blackboard. But we already have a value for x_i in $\mathbf{c}$, c_i. Before we overwrite c_i with x_i, we need to evaluate it. If $f(c_1, c_2, ..., x_i, ..., c_n)$ is better than $f(c_1, c_2, ..., c_i, ..., c_n)$, then we will accept x_i as the new value of c_i. Otherwise, we keep c_i. Note that this conflict resolution mechanism is required even if the factors do not overlap (as in a CPSO/CCGA equivalent FEA). We must always at least resolve the single x_i and c_i. If factors overlap, there will merely be more optimizers of x_i attempting to write to $\mathbf{c}$ and therefore more values for x_i to reconcile.

The main difference between FEA and CPSO, for example, is that this conflict resolution occurs in the Compete Step for FEA and in the underlying selection of the

gbest for CPSO. In FEA, we avoid hitchhiking in $\mathbf{c}$ because we evaluate writes to the blackboard on a variable by variable basis. In CPSO, we avoid hitchhiking in *gbest* only when the factors are comprised of single variables.

However, in both cases, if the factor size is greater than one, the individual factors themselves are subject to hitchhiking even if the global context is not. We will return to this theme later in Chapter 6 when we apply this framework to PSO directly. For now, we will proceed by applying the idea of information exchange via a blackboard and conflict resolution to PSO, FEA-PSO without overlapping swarms and FEA-PSO with overlapping swarms.

We look first at how we might apply this framework to PSO itself and see how and why hitchhiking arises. In order to do this, we return to our previous example of the minimization of a $4d$ Sphere function. The current *gbest* is shown in the first row of Table 4.1 to emphasize the blackboard nature of recording the *gbest* in PSO. It is worth noting as an aside that PSO contains a second *blackboard* in the form of each particle's *pbest* through which each particle is able to exchange information with its best past performance.

When we consider the *gbest* update equation from Algorithm 2.4, we note that the *gbest* as a blackboard is updated wholesale with the information from an individual particle if any particle has a *pbest* with a better fitness than the current *gbest*. The fitness of the current *gbest* is 39.97. And while in this case every particle's *pbest*

Table 4.1: *Hitchhiking in PSO*

$pbest_j$	**X**	$f(\mathbf{x})$
$gbest_{current}$	[2.39, 1.24, 5.71, 0.34]	39.97
1	[1.53, 1.84, 5.29, 0.59]	34.06
2	[0.42, 2.01, 4.76, 1.84]	30.26
3	[3.23, 0.72, 4.68, 0.47]	33.07
4	[2.83, 3.83, 2.71, 1.27]	31.64
$gbest_{new}$ (#2)	[0.42, 2.01, 4.76, 1.84]	30.26

has a better fitness than the current *gbest*, Particle 2's *pbest* has the best fitness of 30.26. Thinking of the *gbest* as a blackboard, Particle 2 gets to write its *pbest* to the blackboard. During the next iteration, all of the particles will read both from the *gbest* and their individual *pbest*s to update their velocities.

As before, we can identify hitchhiking when examining the Sphere function because variable values should be decreasing in the direction of $[0.0, 0.0, 0.0, 0.0]$ for a global minimum. If we compare the values of X_1 and X_3 in $gbest_{new}$ to their corresponding values in $gbest_{current}$, we see that they have decreased. However, looking at X_2 and X_4 in $gbest_{new}$ and comparing them to their previous values in $gbest_{current}$, we see that they have increased. The merge operation is thus a complete replacing of the contents of the blackboard with the *pbest* with the best fitness. The downside to this definition of merge is that we can lose potentially better information.

Let us compare this to an FEA-PSO example for the same problem but with non-overlapping factors with $size = 1$. With $size = 1$, hitchhiking is impossible in the constituent factors when using PSO for optimization. The example is shown

Table 4.2: *FEA-PSO Determination of C_{new} with Non-overlapping Factors*

gbest	**X**	$f(\mathbf{x})$
C	[2.39, 1.24, 5.71, 0.34]	39.97
s_1	[1.53, ----, ----, ----]	36.59
s_2	[----, 2.01, ----, ----]	42.47
s_3	[----, ----, 4.68, ----]	29.27
s_4	[----, ----, ----, 1.27]	41.47
C_{new}	[1.53, 1.24, 4.68, 0.34]	25.90

in Table 4.2. In this Table, we can see the current $\mathbf{c}$, the *gbest*s of each constituent PSO/factors, and what will be the new $\mathbf{c}$. Thinking of $\mathbf{c}$ as a blackboard and $x_1 = 1.53$ as a new value to be written to the blackboard, we invoke our merge operation. Because we can only have one value for c_1, our merge operation is a conflict resolution mechanism whereby the x_1 or c_1 that leads to the best fitness is the value chosen. In this case, because we are working with the Sphere function (which is separable), we know that $x_1 = 1.53$ is better than $c_1 = 2.39$, so c_1' should be 1.53. This is shown in $\mathbf{c}_{new}$. We now move to x_2, and so on. Eventually, we have evaluated all merges individually to make sure that overall fitness has not worsened. So the main difference between PSO and FEA-PSO is that the merge operation is global in PSO and variable by variable in FEA-PSO.

As we can see with Table 4.3, although we have more conflicts to resolve when the factors overlap, our merge operation is still able to handle a variable by variable resolution of those conflicts.

Interestingly enough, this test of variable by variable information exchange for an

Table 4.3: *FEA-PSO Determination of C_{new} with Overlapping Factors*

gbest	**X**	$f(\mathbf{x})$
C	[2.39, 1.24, 5.71, 0.34]	39.97
s_1	[1.53, 1.84, ----, ----]	38.45
s_2	[----, 2.01, 4.76, ----]	32.53
s_3	[----, ----, 4.68, 0.47]	29.37
s_4	[----, ----, ----, 1.27]	41.47
C_{new}	[1.53, 1.24, 4.68, 0.34]	25.90

improvement in fitness is related to another form of exchange found in economics. Specifically this floor-like constraint on fitness with respect to the individual values of the variables is reminiscent of the concept, Pareto efficiency.

4.2.3 Pareto Efficiency

Pareto efficiency, named for Italian economist Vilfredo Pareto, is usually applied to changes in the distribution of resources for individuals or groups of individuals subject to some constraint such as a preference function.[1] Although Edgeworth [47] developed the original analysis of beneficial trade between two individuals, Pareto refined the theory and added the concept of Pareto improvements and Pareto efficiency [11]. Nash later applied game theory to these ideas [48] and there is a definite resemblance between strongly dominating strategies and Pareto efficiency.

Consider the case where Jane and Sam each have a basket of apples and oranges

[1]Pareto efficiency is also known as Pareto optimality. We eschew the term *optimality* in this discussion to avoid confusion.

(economics never says where these come from). Given their individual preferences, modeled by *utility functions*, there is a possibility of mutually beneficial exchange between Jane and Sam if they can trade apples for oranges or oranges for apples. If Jane trades three apples for four of Sam's oranges and the utility functions for each are higher, then we have made a Pareto *improvement*. Jane and Sam can keep exchanging apples and oranges until no Pareto improvement can be made, at which point we have a Pareto *efficient* situation.

We can view our information exchange via a blackboard and our conflict resolution mechanism as using the same standard, even if the analogy is not exact. As we showed in the previous section, in a single objective optimization problem, for FEA information exchange occurs if the candidate new value, x_i, for the c_i in the context, C, improves fitness. This is similar to the exchange of apples and oranges between Jane and Sam.

In the case of PSO, however, we have a much blunter instrument. In that algorithm, we must find an entire solution that has a higher fitness than the current *gbest*. This candidate has to come from the swarm's current collection of *pbest*s. And while the fitness of a particular *pbest* may be greater than the current *gbest*, as we have seen, it is possible for the desirability of specific variables to be less than if we could have exchanged information variable by variable. The "lumpiness" of the transaction leads to inefficiencies or hitchhiking.

4.2.4 Pareto Efficiency and PSO

This application of Pareto Efficiency to a PSO-like algorithm appears to be unique. Applications of Pareto efficiency to PSO have generally focused on multi-objective optimization rather than single objective optimization. Pareto efficiency is a natural approach to resolving conflicts in problems with shared inputs and incommensurable outputs. There are many different possible combinations of minimizations of those outputs, and one needs a way to evaluate them. In this sense, the outputs are like the apples and oranges of the example above. Applications to multi-objective optimization in PSO have included magnetostatics [49], job shop scheduling [50] [51], power dispatch [52] [53], and portfolio optimization [54] to name a few.

There has been research on transforming single-objective optimization problems into multi-objective optimization problems via *helper objectives* and then applying Pareto efficiency [55], [56]. For example, in Chapter 2, we discussed how elitism in GA can lead to premature convergence. With a helper objective such as "maintain diversity", the conventionally single objective GA (minimize f) is cast as a multi-objective problem, "minimize f and maintain diversity" [55]. However, our approach is distinct from this research in that it focuses on the characteristics of information exchange and conflict resolution "as is" in PSO and, more generally, FEA without introducing auxiliary objectives. We do not apply helper objectives and in no way transform our problems to be multi-objective. Our approach is still single objective

optimization.

Although our approach involves Pareto improvements, FEA and similar algorithms are not necessarily Pareto *efficient*. The algorithms simply guarantee that if some c_i in $\mathbf{c}$ is replaced, it will be a Pareto improvement under the function being optimized. Pareto efficiency requires that we be unable to make a Pareto improvement. A Pareto improvement in the FEA case involves allocating the information contained in all the subpopulations to get a *better* successor context, $\mathbf{c}_{new}$. We obtain this result by considering $\mathbf{c}$ and a particular ordering of the variables. While any particular ordering does not matter, we cannot know if the ordering we picked leads to the best possible $\mathbf{c}_{new}$. Additionally, we can only change a single variable at a time. A Pareto efficient outcome would use all the information contained in the subpopulations to create the best possible $\mathbf{c}_{new}$. Unfortunately, determining a Pareto efficient outcome would be exponential in both the factor overlap and the dimension of the problem, $O(|O_i|^d)$, since all combinations would need to be tried to find the *best* successor(s).

4.3 The Context over Time

Blackboard architectures generally require four operations to be defined: insertion, merge, retrieval and deletion. When applying this framework to algorithms like FEA, we have generally acknowledged that insertion and retrieval pose no problems and that

deletion is not applicable. This leaves the merge operation. Because our blackboard can only contain one value for c_i in $\mathbf{c}$ at a time, and we want any change in $\mathbf{c}$ to be a Pareto improvement, every time a subpopulation suggests a candidate x_i to replace c_i, we evaluate x_i by replacing c_i in $\mathbf{c}$ and either pick x_i if it is better or stay with c_i if it is not. Thus our conflict resolution mechanism always leads to a Pareto improvement.

While in some cases, differences in the merge operation are a sufficient basis of comparison between two algorithms (such as FEA and PSO), in cases where two algorithms use the same merge operation, we may need to look at the evolution of C over time. In this section, we will look at how $\mathbf{c}$ changes in FEA over time, developing a notation and nomenclature that will be put to use in the next chapter when we examine the evolution of $\mathbf{c}$ under DFEA introduced in the last chapter.

In the FEA Compete Step, the information flows from the swarms to the context. At minimum, at least one swarm will have been optimizing some variable x_i and this new, potentially better, value will need to be evaluated against the existing value, c_i, in the context, $\mathbf{c}$. This is our merge operation which involves conflict resolution. If we have an existing context, $[c_1, c_2, c_3, c_4]$ and a new x_2, we will evaluate both $f([c_1, c_2, c_3, c_4])$ and $f([c_1, x_2, c_3, c_4])$ and either keep c_2 or select x_2 depending on which gives the better fitness. More generally, there may be many swarms optimizing x_j, the set of which we will designate $\mathbf{O}_j$, and there will be $|\mathbf{O}_j| + 1$ conflicting possibilities for x_j, including c_j.

Algorithm 4.13 FEA Compete

Input: Objective function f, Subpopulations $\boldsymbol{\mathcal{S}}$, Optimizers $\boldsymbol{\mathcal{O}}$, Global context $\mathbf{c}$
Output: Global context $\mathbf{c}$

```
1: for j = 1 to d do
2:     fitness ← f(c)
3:     value ← c[j]
4:     for i in O_j do
5:         candidate ← S[i].best
6:         c[i] ← candidate.x[i]
7:         if f(c) ≤ fitness then
8:             value ← candidate.x[i]
9:             fitness ← f(c)
10:        end if
11:    end for
12:    c[i] ← value
13: end for
14: return c
```

The order of variable resolution can be arbitrary, but this does not mean that every order will end up with the same result for C. Let us consider the case where factors have only one variable, thus X_i is the factor, x_i is the value from the factor, and c_i is the corresponding value from the context, $\mathbf{c}$. If we evaluate the factors in the following order, X_1, X_3, X_4, X_2, we will almost assuredly end up with a different value for $\mathbf{c}$ than if we evaluated the factors in this order, X_1, X_2, X_3, X_4. Throughout the following discussion, we will use the order X_1, X_2, X_3, X_4 but the results do not depend on this order. It just makes the bookkeeping and exposition clearer. As a reminder, the FEA Compete Step pseudocode is reproduced here as Algorithm 4.13.

In our previous discussions, we could focus on the *values* that variables could take,

either x_i or c_i, depending. Now we are concerned with differences over time and the context in which conflicting values are resolved. Thus, we need a notation that is more abstract than the value-based notation of Tables 4.1, 4.2 or 4.3.

Instead of values, we will use symbols. First, we use the term *reconciliation* to cover both the sharing of new values of x_j from optimizers of x_j, O_j, and the resolution of the conflicting values against $\mathbf{c}$. A variable, X_i, that has not been reconciled yet will be denoted by $\bigodot$. For a variable that has been reconciled we will use $\bigoplus$. Note that this says nothing about whether c_i was kept or one of $x_i^k, \forall i = 1 \ldots |O_j|$ replaced it.

As an example we return to our $d = 4$ Sphere problem although the actual problem is not as important. After the Optimize Step, just as the Compete Step begins, the context, C, appears as $[\bigodot\bigodot\bigodot\bigodot]$. This means that none of the variables X_1, X_2, X_3, X_4 have been reconciled.

When X_1 is reconciled, the context becomes $[\bigoplus\bigodot\bigodot\bigodot]$. The important thing to note here is that X_1, whether it is c_1 or x_1, is determined in relation to the other variables: X_2, X_3, X_4. If we continue with X_2, we have $[\bigoplus\bigoplus\bigodot\bigodot]$, followed by $[\bigoplus\bigoplus\bigoplus\bigodot]$, and finally followed by the reconciliation of X_4, $[\bigoplus\bigoplus\bigoplus\bigoplus]$. If we look at the evolution of the blackboard context, C, over time then we see the pattern in Table 4.4. The importance of the patterns that emerge will become more apparent in the next chapter.

Table 4.4: *FEA Context C over Time*

t_i	C
1	$[\oplus \odot \odot \odot]$
2	$[\oplus \oplus \odot \odot]$
3	$[\oplus \oplus \oplus \odot]$
4	$[\oplus \oplus \oplus \oplus]$

The important point here is not the values but that X_1 is reconciled in relation to the starting values of X_2, X_3, X_4. X_2 is reconciled in relation to the reconciled value of X_1 and the unreconciled values of X_3 and X_4. X_3 is reconciled in relation to the reconciled values of X_1, X_2 and the unreconciled value of X_4. And, finally, X_4 is reconciled in the presence of reconciled values for X_1, X_2, and X_4.

4.4 Conclusions

In this chapter, we have abandoned the dichotomy of cooperation versus cooperation when thinking about these multi-population algorithms. Instead we introduced the concept of information exchange via a blackboard architecture as a potentially more fruitful way of analyzing and improving these algorithms. Additionally, we identified the Compete Step in FEA as a merge operation involving conflict resolution. We identified the conflict resolution mechanism being based on Pareto improvement. Finally, we developed a symbolic notation for identifying the evolution of the context or blackboard over time. This notation will allow us to easily identify differences in

the evolution of blackboards between different FEA variants.

Chapter 5

Distributed Factored Evolutionary Algorithms Revisited

Armed with a conceptual framework based on information exchange via a blackboard architecture and conflict resolution, in this chapter we revisit the Distributed Factored Evolutionary Algorithm. One of the persistent puzzles for both OSI/DOSI and FEA/DFEA has been that the distributed versions, against expectations, have not always performed as well or at least similarly to their centralized counterparts. Using the framework developed in the previous chapter, we will determine the reason for the differences between FEA and DFEA.

5.1 Discrepancies between FEA and DFEA

At first glance, DFEA and FEA appear to be nearly identical algorithms. The main difference is that, instead of having single context like FEA, $\mathbf{c}$; in DFEA, each subpopulation, S_i, has its own context, $\mathbf{c}_i$. As the algorithm progresses these multiple contexts need to communicate with each other in order to stay synchronized. But there may be many optimizers of X_j and so some S_i must be designated as having the definitive value of c_j in its context. The solution is to designate one of the swarms optimizing X_j to be the *arbiter* of X_j, which we denote $a(X_j)$. During the DFEA Compete Step, all optimizers of X_j communicate their values of X_j to the arbiter and the arbiter compares those values (including its own) with c_j found in its context. This is the DFEA Compete Step discussed in Chapter 3 (Algorithm 5.14).

Because neither the Compete Step nor the Share/Exchange Steps consume random numbers, if the algorithms start with the same initial state (random seed) and use the same evolutionary algorithm, they should get the same results. As we see in Listing 5.1, this is not the case. We can use the framework developed in the previous chapter to begin to unravel why this is so.

The DFEA Compete Step reconciles all the contexts, $\mathbf{c}_i \subset \mathbf{C}$, using the notation developed in the previous chapter we can analyze changes in all the $\mathbf{c}_i$ over time. We assume that the arbiter for X_j is $\mathbf{c}_j$ ($\mathbf{c}$ in Line 2 of Algorithm 5.14), and as before we take the variables in order, X_1, X_2, X_3, X_4. Once again, we start at the end of the

Algorithm 5.14 DFEA Compete

Input: Objective function $f(x)$, subpopulations $\boldsymbol{\mathcal{S}}$, optimizers $\boldsymbol{\mathcal{O}}$, arbiters $\boldsymbol{\mathcal{A}}$, contexts $\boldsymbol{\mathcal{C}}$
Output: New contexts $\boldsymbol{\mathcal{C}}$

```
1: for i = 1 to d do
2:     C ← 𝒞[𝒜[i]]
3:     fitness ← f(C)
4:     value ← C.c_i
5:     for j in 𝒪[i] do
6:         candidate ← 𝒮[j].best
7:         𝒞.c_i ← candidate.x_i
8:         if f(𝒞) is better than fitness then
9:             value ← candidate.x_i
10:            fitness ← f(𝒞)
11:        end if
12:    end for
13:    𝒞.c_i ← value
14:    for j in 𝒪[i] do
15:        𝒞[j].c_i ← 𝒞.c_i
16:    end for
17: end for
18: return C
```

Listing 5.1: *Execution of FEA and DFEA with same random seed*

```
$ python test.py --benchmark ackley-1
benchmark ackley-1
seed 1521996906
starting FEA
fitness = 4.648499585258037e-08
starting DFEA
fitness = 0.1089753263218296
```

Update Step and assume all the contexts are identical. Using our example from Table 4.4, when we reconcile X_1, we will have $\mathbf{c}_1 = [\bigoplus \odot \odot \odot]$, and when we reconcile X_2, we will have $\mathbf{c}_2 = [\odot \bigoplus \odot \odot]$. Similarly, $[\odot \odot \bigoplus \odot]$ and $[\odot \odot \odot \bigoplus]$ follow

Table 5.1: *Evolution of Context(s) in FEA and DFEA*

t	**FEA**	**DFEA**
1	$[\oplus\ \odot\ \odot\ \odot]$	$[\oplus\ \odot\ \odot\ \odot]_1$ $[\odot\ \odot\ \odot\ \odot]_2$ $[\odot\ \odot\ \odot\ \odot]_3$ $[\odot\ \odot\ \odot\ \odot]_4$
2	$[\oplus\ \oplus\ \odot\ \odot]$	$[\oplus\ \odot\ \odot\ \odot]_1$ $[\odot\ \oplus\ \odot\ \odot]_2$ $[\odot\ \odot\ \odot\ \odot]_3$ $[\odot\ \odot\ \odot\ \odot]_4$
3	$[\oplus\ \oplus\ \oplus\ \odot]$	$[\oplus\ \odot\ \odot\ \odot]_1$ $[\odot\ \oplus\ \odot\ \odot]_2$ $[\odot\ \odot\ \oplus\ \odot]_3$ $[\odot\ \odot\ \odot\ \odot]_4$
4	$[\oplus\ \oplus\ \oplus\ \oplus]$	$[\oplus\ \odot\ \odot\ \odot]_1$ $[\odot\ \oplus\ \odot\ \odot]_2$ $[\odot\ \odot\ \oplus\ \odot]_3$ $[\odot\ \odot\ \odot\ \oplus]_4$

for X_3 and X_4.

We would now like to compare the state of the two algorithms after their respective Compete Steps. To do this we observe, that since our examples have used the same order of reconciliation, we can line up the single FEA context at time t_i with the corresponding DFEA context $\mathbf{c}_i$. This is shown in Table 5.1.

Aligned this way, the difference between the algorithms becomes immediately apparent. For FEA, X_2 is reconciled at t_2 in the presence of the reconciled value of c_1 from t_1 ($\oplus$). For DFEA, X_2 is reconciled in $\mathbf{c}_2$ with the *unreconciled* value of c_1 ($\odot$). While it is possible that c_2 will be the same value in both cases, it is not guaranteed. Additionally, it seems unlikely that every c_j would reconcile exactly the

same way for both algorithms over all variables and iterations.

This analysis so far has been only for the Compete Steps. Both FEA and DFEA have an additional step, the Share Step, which has a greater importance for DFEA. In FEA, the Share Step is mostly a bookkeeping step where the new context is communicated back to the individual populations and fitness values are re-evaluated. Additionally, a form of elitism introduces the context as an actual individual in the population. In DFEA, the Share Step includes this same bookkeeping but also moves information about the arbitrated values about the induced network we discussed in Chapter 3. We will now look at how this influences information sharing. As a reminder, $\mathbf{c}_i$ is a vector of values assigned to $\mathbf{X}$ for swarm S_i whereas c_i is the value of X_i in $\mathbf{c}_i$.

Consider two swarms that are regarded to be *neighbors*. The neighbors relation can be defined in a number of ways, but in Chapter 3 we defined it in the context of the set of optimizers, $\mathcal{O}$. If two swarms, S_j and S_k, are both members of some set of optimizers $\mathbf{O}_i$, then they are considered to be neighbors. During the Share Step, those two swarms will compare how recent the values of all variables in their contexts, $\mathbf{c}_j$ and $\mathbf{c}_k$, are using the δs. If $\mathbf{c}_j$ has a newer value of c_i than $\mathbf{c}_k$, then $\mathbf{c}_k$ will take $\mathbf{c}_j$'s value of c_i. If the reverse is true, $\mathbf{c}_j$ will take $\mathbf{c}_k$'s value of c_i.

The neighbor relation sets up the possibility of swarms being indirect neighbors as well. If S_j and S_k are both members of $\mathbf{O}_i$ and S_k and S_m are both members

of $\mathbf{O}_q$ then S_j and S_m are indirectly neighbors. Information will flow from one to the other, depending on the number of iterations in the Share Step. The neighbor relation induces a topology on the swarms through which information travels, and if the Share Step iterates a sufficient number of times, information will flow from $\mathbf{c}_1$ to $\mathbf{c}_d$. As a result, all $\mathbf{c}_i$ will be identical. This is called *(full) consensus*.

Relative to FEA, however, this sharing takes place too late to affect how reconciliation plays out in the DFEA Compete Step. Consider reconciliation of c_1 and c_2. First c_1 is reconciled with the optimizers $\mathbf{O}_1$, and we have $\mathbf{c}_1 = [\bigoplus \odot \odot \odot]$. Next, c_2 is reconciled with optimizers $\mathbf{O}_2$, and we have $\mathbf{c}_2 = [\odot \bigoplus \odot \odot]$. During the Share Step, c_1 from $\mathbf{c}_1$ will be shared with $\mathbf{c}_2$, but c_1 will not have been determined in the context of $\mathbf{c}_2$. In fact, when c_1 is changed to c_1' in $\mathbf{c}_2$, $\mathbf{c}_2$ is not re-evaluated at all. We thus have no way of knowing if the change is Pareto improving or not. To signify this, we use the $\bigotimes$ symbol: $\mathbf{c}_2 = [\bigotimes \bigoplus \odot \odot]$. With full consensus, $\mathbf{c}_2$ will eventually look like $[\bigotimes \bigoplus \bigotimes \bigotimes]$. Although each of the values c_1, c_2, c_3, and c_4 will have been Pareto improvements when they were evaluated during reconciliation, at no point were they evaluated *collectively*. Even with full consensus, the contexts in DFEA do not collectively preserve the information semantics of the context in FEA.

5.2 Revising DFEA

In order to preserve the information flow and conflict resolution semantics of FEA in DFEA, we will need to revise DFEA. Based on Table 5.1 it would appear that the Share Step and Compete Step need to happen simultaneously. Additionally, if we wish the right-hand side of the table to match the left-hand side of the table, in the ideal case, we need to start by considering all swarms to be neighbors of all other swarms. Later we will investigate what relaxing consensus might mean for the revised algorithm. This new *Reconcile* Step that combines both information sharing and conflict resolution is described as Algorithm 5.16. As before, this is a high level specification that does not describe a specific parallel implementation. We will present an Actor model-based implementation in Chapter **??**.

By adding a *broadcast* loop at Line 14 every time some c_i is reconciled, the reconciliation is communicated to all the other contexts. For any c_j to be reconciled, all $c_k, \forall k < j$ will be their reconciled values, just as in the FEA Compete Step. The revised DFEA relegates the Share Step to performing similar bookkeeping functions as it does in FEA.

Using our previous notation, when X_1 is reconciled to $\mathbf{c}_1 = [\bigoplus \odot \odot \odot]$, all the other contexts $\mathbf{c}_j$ will have X_1 updated as well: $\mathbf{c}_2 = [\bigoplus \odot \odot \odot]$, $\mathbf{C}_3 = [\bigoplus \odot \odot \odot]$, and $\mathbf{c}_4 = [\bigoplus \odot \odot \odot]$. And when X_2 is reconciled it will be in the context of the reconciled value of X_1 just as in FEA: $\mathbf{C}_2 = [\bigoplus \bigoplus \odot \odot]$. Table 5.2

Algorithm 5.15 Distributed Factored Evolutionary Algorithm

Input: Function f, Evolutionary Algorithm ae
Output: Best context $\mathbf{c}$ as candidate solution $\mathbf{x}$

```
1: 𝒳 ← factorize(X)
2: S ← ae.initialize(f, 𝒳)
3: 𝒞 ← initialize-contexts(S)
4: 𝒪 ← identify-optimizers(𝒳)
5: 𝒜 ← identify-arbiters(𝒳)
6: repeat
7:     repeat
8:         for S in S do
9:             S ← ae.update(S)
10:        end for
11:    until stopping criteria
12:    𝒞 ← reconcile(f, S, 𝒪, 𝒜, 𝒞)
13: until stopping criteria
14: c ← select-best-context(f, 𝒞)
15: return c
```

shows the end result as compared to FEA. Now all four DFEA contexts, $\mathbf{c}_i$, are the same as the FEA context at t_4.

This section has demonstrated how looking at these algorithms in terms of information flow and conflict resolution (reconciliation) can reveal a deeper structure and more interesting semantics than invoking cooperation versus competition. We examined how reconciliation works in FEA and original DFEA and showed that the semantics of the two were not identical as previously supposed. Using the same framework, we devised a revised DFEA that does preserve the semantics of FEA. Finally, we encountered something new. In the original DFEA, a value c_j that was a Pareto improvement in $\mathbf{C}_j$ was communicated to $\mathbf{C}_k$ without any evaluation ($\bigotimes$). What

Algorithm 5.16 DFEA Reconcile

Input: Function f, Subpopulations $\mathbf{S}$, optimizers $\boldsymbol{\mathcal{O}}$, arbiters $\boldsymbol{\mathcal{A}}$, Local contexts $\boldsymbol{\mathcal{C}}$
Output: Local contexts $\boldsymbol{\mathcal{C}}$

```
1: for j = 1 to d do
2:     c ← C[A(x_j)]
3:     fitness ← f(c)
4:     value ← c[j]
5:     for k in O_j do
6:         candidate ← S[k].best
7:         c[j] ← candidate.x[j]
8:         if f(c) ≤ fitness then
9:             value ← candidate.x[j]
10:            fitness ← f(c)
11:        end if
12:    end for
13:    c[j] ← value
14:    for k = 1 to d do
15:        C[k].c[j] ← c[j]
16:    end for
17: end for
18: return C
```

impact this might have on the operation of the algorithm and what it means for performance relative to FEA is not entirely clear. For now, we refer to these values as *discordant* because they are injected into a context without any conflict resolution.

Table 5.2: *Final Context(s) in FEA and Revised DFEA*

t	**FEA**	**DFEA**
1	$[\oplus\odot\odot\odot]$	$[\oplus\odot\odot\odot]_1$ $[\oplus\odot\odot\odot]_2$ $[\oplus\odot\odot\odot]_3$ $[\oplus\odot\odot\odot]_4$
2	$[\oplus\oplus\odot\odot]$	$[\oplus\oplus\odot\odot]_1$ $[\oplus\oplus\odot\odot]_2$ $[\oplus\oplus\odot\odot]_3$ $[\oplus\oplus\odot\odot]_4$
3	$[\oplus\oplus\oplus\odot]$	$[\oplus\oplus\oplus\odot]_1$ $[\oplus\oplus\oplus\odot]_2$ $[\oplus\oplus\oplus\odot]_3$ $[\oplus\oplus\oplus\odot]_4$
4	$[\oplus\oplus\oplus\oplus]$	$[\oplus\oplus\oplus\oplus]_1$ $[\oplus\oplus\oplus\oplus]_2$ $[\oplus\oplus\oplus\oplus]_3$ $[\oplus\oplus\oplus\oplus]_4$

5.2.1 Complexity of the Revised DFEA

In Chapter 3, we established the complexity of DFEA to be: The total algorithmic complexity is thus:

$$\mathcal{O}(FEA) = \mathcal{O}(mskpU(d)) + mnd\Lambda(d) + mdWs^2)$$

where m is FEA iterations, s is the number of factors/subpopulations, k is the number of EA or update iterations, p is the number of individuals in each subpopulation, d is the number of variables, and n is the number of optimizers (largest) of any X_i.

With the Revised DFEA, the Share Step is now the same as FEA and the Compete

Step is now different. The complexity of FEA was:

$$\mathcal{O}(DFEA) = \mathcal{O}(mskpU(d)) + mnd\Lambda(d) + msd)$$

There are two inner loops in the Compete Step, the first updates the arbiter's context for a specific variable. It does this by looping over the number of optimizers for X_i which we take to be n. The second broadcasts the arbitrated value of X_i to all of the factors/subpopulations, s. The complexity of the inner loop is thus $\mathcal{O}(n\Lambda(d) + s)$. Because the evaluation of an individual is going to be at least $\mathcal{O}(d)$, we claim that d is always at least as larger and usually larger than s and therefore the complexity of the inner loop is $\mathcal{O}(n\Lambda(d)$. This gives the revised DFEA's Compete Step the same complexity as FEA and thus the same overall complexity as FEA.

5.3 Comparing FEA, Original DFEA and Revised DFEA

Based on the previous discussion and analysis, our hypothesis is that the revised DFEA and FEA will perform the same. The revision that was made ensures that, other things being equal, FEA and the revised DFEA will end up with the same result. As for the original DFEA, it is difficult to say how the discordant values

Listing 5.2: *Execution of FEA, original DFEA, and revised DFEA with same random seed*

```
$ python test.py --benchmark ackley-1
benchmark ackley-1
seed 1521996906
starting FEA
fitness = 4.648499585258037e-08
starting DFEA
fitness = 0.1089753263218296
seed 1521996906
starting Revised DFEA
fitness = 4.648499585258037e-08
```

influence the performance of the algorithm. Because [57] showed that the original DFEA was sometimes better and sometimes worse than FEA, we also hypothesize that the results will be mixed. We will revisit this hypothesis and the results later in the discussion of future work. As a starting point, we re-execute our test script from Listing 5.1 for all three algorithms and show the results in Listing 5.2.

As a broader test of our hypothesis, we ran a large number of experiments on standard benchmark functions from different categories for FEA-PSO, revised DFEA-PSO, and original DFEA-PSO. We include results for the single population *gbest* PSO as a baseline. The following sections describe the design, results, and discussion of those experiments.

Table 5.3: *Benchmark Optimization Functions by Category*

Category	Benchmark Function
Bowl	Exponential, Sargan, Sphere
Many Local Optima	Ackley-1, Eggholder, Griewank, Rastrigin, Salomon, Stretched-V
Plate	Brown, Schwefel-2.23, Whitley, Zakharov
Ridge	Michalewicz, Schaffer-F6, Schwefel-2.22
Valley	Dixon-Price, Rosenbrock, Schwefel-1.2

5.3.1 Design

We selected benchmark optimization problems from [8] and [18] that were scalable to multiple dimensions (See Appendix A more information the benchmark functions). The problems selected are shown in Table 6.3, arranged by categories inspired by [58]. All of the problems are minimization problems, and with the exception of the Exponential and Eggholder functions, they all have a minimizing solution and value of $f([0]^d) = 0$. The Exponential function has a minimum at $[-1]^d$, and the Eggholder function has a dimension-dependent minimum and minimizing vector. None of the functions except the Sphere function are separable in their current forms.

Experiments consisted of 50 runs of each algorithm on each benchmark function with a dimension of 32. Because we noticed that the results were not always normally distributed—hardly a surprise for optimization problems—the confidence intervals were 500 replications of the Bootstrap to estimate 95% confidence intervals [59].

Following [60], each algorithm used the same number of candidate solutions. In this case we chose 10 particles per dimension, that is $d \times 10 = 320$.

PSO, the PSO portion of FEA, and the various DFEA versions all used the same parameters: $\omega = 0.729$ and $\phi_1 = \phi_2 = 1.49618$. While PSO was run for 100 iterations, FEA versions were run for 20 FEA/DFEA iterations separated by 5 PSO iterations for a total of 100 PSO iterations. All FEA/DFEA variants used the "Simple Centered" factor of $i, i+1$, which followed the functional form of most of the benchmark functions—they are functions of adjacent x values—and shown by Strasser *et al.* to perform well [9]. With $d-1$ such factors, and $d = 32$, there were $\lfloor (320/31) \rfloor = 10$ particles per swarm for the FEA/DFEA-PSO variants.

5.3.2 Results

The results are shown in Table 5.4. Independent of which algorithm is best, what we are looking for, in general, is for FEA-PSO and the new DFEA-PSO to have similar performance. This appears to be true for Ackley, Brown, Exponential, Salomon, Schaffer-F6, Schwefel-1.2, Schwefel-2.22, Schwefel-2.23, Sphere, Stretched-V, and Zakharov. There were 19 benchmark functions overall, so our experiments show that the results were similar for FEA-PSO and the new DFEA-PSO for 11 of them (58%).

There were six cases where the original or old DFEA-PSO performed the same as

the new DFEA-PSO, five where they performed better than FEA-PSO: Dixon-Price, Eggholder, Griewank, Michalewicz, Rosenbrock, and one where they were worse then FEA-PSO: Rastrigrin. In three cases, FEA-PSO was better than either version of DFEA-PSO (Rastrigrin, Sargan, Whitley). In three cases, PSO was better than any FEA variant: Salomon, Schwefel-1.2 and Zakharov.

5.3.3 Discussion

Given that FEA and the revised DFEA are demonstrably equivalent, it is surprising that there are eight cases out of 19 where they did not have the same results. All FEA/DFEA variants depend on PSO as the underlying optimizer. The variants all have the same numbers of factors for each problem and thus the same number of swarms. In the code tested, they are all initialized the same way and at the same time. Furthermore, the (D)FEA Compete/Share/Reconcile Steps as presented do not have any stochastic elements that might cause a purely *random* divergence.

Because of this we decided to run a second set of experiments on the same benchmark functions. This time each run, i, of FEA-PSO, old DFEA-PSO and revised/new DFEA-PSO used the same random seed, $seed_i$. As we can see in these results, shown in Table 5.5, FEA-PSO and revised DFEA-PSO had *exactly* the same results as we would expect. Thus it appears that the random seed was the culprit in generating the differences in performance.

Table 5.4: *Comparison of PSO, FEA-PSO and both variants of DFEA-PSO*

Benchmark	PSO	FEA-PSO	Revised DFEA-PSO	Old DFEA-PSO
ackley-1	1.84e+00 (1.77e+00, 1.90e+00)	**2.81e-03** (5.23e-06, 7.00e-03)	**2.78e-03** (3.82e-08, 6.99e-03)	**2.83e-03** (1.39e-08, 6.98e-03)
brown	7.56e+00 (6.69e+00, 8.55e+00)	**1.22e-25** (3.72e-26, 2.42e-25)	**1.21e-25** (5.11e-26, 1.98e-25)	1.02e-23 (1.36e-24, 2.42e-23)
dixon-price	4.23e+01 (2.54e+01, 6.34e+01)	1.18e+02 (1.07e+02, 1.28e+02)	**3.85e+01** (2.93e+01, 4.70e+01)	**3.87e+01** (2.98e+01, 4.92e+01)
eggholder	-1.68e+04 (-1.71e+04, -1.64e+04)	-1.77e+04 (-1.79e+04, -1.74e+04)	**-2.10e+04** (-2.13e+04, -2.07e+04)	**-2.05e+04** (-2.07e+04, -2.02e+04)
exponential	-9.99e-01 (-1.00e+00, -9.99e-01)	-1.00e+00 (-1.00e+00, -1.00e+00)	-1.00e+00 (-1.00e+00, -1.00e+00)	-1.00e+00 (-1.00e+00, -1.00e+00)
griewank	4.96e-01 (4.47e-01, 5.45e-01)	9.49e-01 (8.99e-01, 9.91e-01)	**1.49e-01** (6.92e-02, 2.23e-01)	**1.35e-01** (7.51e-02, 2.02e-01)
michalewicz	-8.65e+00 (-9.02e+00, -8.33e+00)	-2.59e+01 (-2.63e+01, -2.56e+01)	**-3.06e+01** (-3.07e+01, -3.04e+01)	**-3.06e+01** (-3.07e+01, -3.04e+01)
rastrigin	1.03e+02 (9.59e+01, 1.11e+02)	**2.60e-02** (1.97e-05, 6.59e-02)	7.28e-02 (1.36e-02, 1.51e-01)	1.01e-01 (3.98e-02, 1.99e-01)
rosenbrock	1.95e+02 (1.56e+02, 2.51e+02)	2.20e+02 (1.69e+02, 2.88e+02)	**4.60e+01** (1.96e+01, 7.77e+01)	**5.09e+01** (1.31e+01, 1.04e+02)
salomon	**1.41e+00** (1.33e+00, 1.48e+00)	2.29e+00 (2.12e+00, 2.49e+00)	1.96e+00 (1.78e+00, 2.12e+00)	1.93e+00 (1.77e+00, 2.17e+00)
sargan	9.76e+00 (8.55e+00, 1.11e+01)	**2.04e-12** (1.37e-12, 2.91e-12)	5.67e+02 (1.32e+02, 1.15e+03)	3.19e+03 (1.54e+03, 5.07e+03)
schaffer-f6	2.49e+00 (2.31e+00, 2.67e+00)	1.99e+00 (1.78e+00, 2.20e+00)	**9.86e-01** (8.60e-01, 1.09e+00)	**9.34e-01** (8.23e-01, 1.04e+00)
schwefel-1.2	**7.96e+03** (7.19e+03, 8.84e+03)	6.59e+04 (5.72e+04, 7.60e+04)	6.53e+04 (4.59e+04, 8.85e+04)	6.88e+04 (5.55e+04, 8.44e+04)
schwefel-2.22	3.01e+02 (2.91e+02, 3.13e+02)	**1.22e-12** (7.73e-13, 1.75e-12)	**1.35e-12** (5.74e-13, 2.40e-12)	**1.54e-12** (8.63e-13, 2.41e-12)
schwefel-2.23	2.44e-01 (1.18e-01, 4.15e-01)	**8.65e-102** (7.25e-116, 2.31e-101)	**5.07e-102** (2.39e-111, 1.67e-101)	**1.39e-101** (8.93e-103, 3.02e-101)
sphere	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
stretched-v	1.20e+01 (1.14e+01, 1.28e+01)	4.37e+00 (3.78e+00, 5.02e+00)	4.24e+00 (3.91e+00, 4.54e+00)	**3.58e+00** (3.31e+00, 3.86e+00)
whitley	9.82e+02 (9.67e+02, 9.99e+02)	**3.51e+02** (2.97e+02, 3.91e+02)	5.35e+02 (4.84e+02, 5.80e+02)	5.62e+02 (5.20e+02, 6.02e+02)
zakharov	**1.39e+02** (1.28e+02, 1.51e+02)	1.60e+03 (3.09e+02, 3.77e+03)	8.16e+02 (7.80e+02, 8.45e+02)	7.97e+02 (7.67e+02, 8.23e+02)

In this second set of experiments, what is perhaps more interesting is that, for most of the problems, the differences in the DFEA versions did not seem to matter. For 14 of the 19 benchmark problems, all three algorithms performed about the same. For two benchmark problems, the old DFEA-PSO performed better than FEA-PSO/revised DFEA-PSO (Rosenbrock, Stretched-V). For three of the benchmark problems, FEA-PSO/revised DFEA-PSO performed better than old DFEA-PSO (Salomon, Sargan, Zakharov).

5.4 Relaxing Consensus

Not only did we introduced the distributed version of FEA or DFEA in Chapter 3, we also examined the implications of relaxing consensus. We now examine what relaxing consensus might mean for our revised DFEA. In Chapter 3, relaxed consensus was achieved in the DFEA Share Step when newly reconciled values were not communicated throughout the network of contexts induced by the neighbor relation. In the revised DFEA, however, we have replaced the Compete and Share Steps with a Reconcile Step. The Reconcile Step replaces a network model using "hops" with a network model using broadcasted messages. In order to introduce an effect like relaxed consensus in the revised DFEA, we introduce the idea of dropped messages.

Referring back to Line 14 in Algorithm 5.16, we see that after a new c_j is reconciled

Table 5.5: *FEA-PSO, Old and New DFEA-PSO with Same Random Seeds*

Benchmark	FEA-PSO	Revised DFEA-PSO	Old DFEA-PSO
ackley-1	3.37e-03 (5.38e-07, 8.96e-03)	3.37e-03 (5.38e-07, 8.96e-03)	2.01e-03 (5.46e-08, 5.95e-03)
brown	2.90e-21 (4.32e-25, 9.17e-21)	2.90e-21 (4.32e-25, 9.17e-21)	2.96e-21 (1.51e-24, 9.30e-21)
dixon-price	3.73e+01 (2.82e+01, 4.72e+01)	3.73e+01 (2.82e+01, 4.72e+01)	3.20e+01 (2.22e+01, 4.14e+01)
eggholder	-2.12e+04 (-2.15e+04, -2.10e+04)	-2.12e+04 (-2.15e+04, -2.10e+04)	-2.07e+04 (-2.10e+04, -2.03e+04)
exponential	-1.00e+00 (-1.00e+00, -1.00e+00)	-1.00e+00 (-1.00e+00, -1.00e+00)	-1.00e+00 (-1.00e+00, -1.00e+00)
griewank	7.14e-02 (2.57e-02, 1.28e-01)	7.14e-02 (2.57e-02, 1.28e-01)	1.71e-01 (8.22e-02, 2.66e-01)
michalewicz	-3.09e+01 (-3.10e+01, -3.08e+01)	-3.09e+01 (-3.10e+01, -3.08e+01)	-3.09e+01 (-3.10e+01, -3.08e+01)
rastrigin	2.28e-01 (1.12e-01, 3.40e-01)	2.28e-01 (1.12e-01, 3.40e-01)	2.28e-01 (1.12e-01, 3.40e-01)
rosenbrock	1.58e+01 (6.38e+00, 2.70e+01)	1.58e+01 (6.38e+00, 2.70e+01)	**3.16e+00** (2.40e+00, 3.97e+00)
salomon	**1.79e+00** (1.64e+00, 1.93e+00)	**1.79e+00** (1.64e+00, 1.93e+00)	2.82e+00 (2.63e+00, 3.05e+00)
sargan	**1.67e+03** (5.58e+02, 2.84e+03)	**1.67e+03** (5.58e+02, 2.84e+03)	1.35e+05 (1.04e+05, 1.77e+05)
schaffer-f6	9.26e-01 (8.24e-01, 1.02e+00)	9.26e-01 (8.24e-01, 1.02e+00)	8.85e-01 (7.79e-01, 9.71e-01)
schwefel-1.2	8.80e+04 (5.37e+04, 1.24e+05)	8.80e+04 (5.37e+04, 1.24e+05)	1.39e+07 (7.09e+06, 2.02e+07)
schwefel-2.22	4.63e-12 (1.06e-12, 1.11e-11)	4.63e-12 (1.06e-12, 1.11e-11)	4.63e-12 (1.06e-12, 1.11e-11)
schwefel-2.23	2.27e-94 (6.55e-100, 6.92e-94)	2.27e-94 (6.55e-100, 6.92e-94)	1.12e-99 (3.06e-101, 3.28e-99)
sphere	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
stretched-v	4.23e+00 (3.91e+00, 4.53e+00)	4.23e+00 (3.91e+00, 4.53e+00)	**2.83e+00** (2.54e+00, 3.14e+00)
whitley	5.46e+02 (5.05e+02, 5.87e+02)	5.46e+02 (5.05e+02, 5.87e+02)	6.68e+02 (5.82e+02, 7.55e+02)
zakharov	**7.62e+02** (7.34e+02, 7.87e+02)	**7.62e+02** (7.34e+02, 7.87e+02)	3.46e+11 (2.01e+11, 4.81e+11)

by the arbiter of X_j, it is broadcast to all other swarms. By broadcasting c_j, any $c_k, \forall k > j$ that is subsequently reconciled has the benefit of this new information. The main point of the previous discussion about reconciliation in the original DFEA was that this did not happen. In effect, the original DFEA never had full consensus in the sense that FEA does.

Now we introduce the idea that the message containing the new c_j may be dropped. This is accomplished through a success rate, r, which determines if a message is delivered from some arbiter of X_j to each of the remaining swarms. If $r = 0.8$, there is a 20% probability that the message from the arbiter of X_1, for example, to X_2 will go missing, in which case when c_2 is reconciled, $\mathbf{c}_2$ will not have the new value of c_1. This could happen for several iterations of the DFEA Reconcile Step, depending on r. But this also means that for any given iteration of the DFEA Reconcile Step, some arbiters will get c_1, and some will not, with probability $1 - r$.

5.4.1 Experiment

In order to test the effects of dropped messages and their implications for relaxing consensus, we re-ran the benchmark experiments from above. All the parameters are the same. The only difference is that we simulated different success rates of $r = 1.0$ (the baseline), 0.8, 0.6, and 0.4. The results are presented in Table 5.6.

5.4.2 Discussion

Looking at the results, we can see how well the revised DFEA did on the various benchmarks with different success rates. The first column in Table 5.6 is a baseline of 100% success. Comparing the results for a 100% and 80% success rate, we can see that the revised DFEA required a success rate of 100% on 10 of 19 benchmarks. These benchmarks were Ackley, Brown, Griewank, Rosenbrock, Schwefel-1.2, Schwefel-2.22, Schwefel-2.23, Sphere, Whitley, and Zakharov. Perhaps the most surprising appearance in this list is the Sphere function. The Sphere function is fairly simple and separable in its variables. Our *a priori* belief was that missed messages would be more important for *non*-separable functions where the optimizing values of variables would have a high degree of correlation (what we have previously called *epistasis* following the GA literature). One would think this would make missed messages less important relative to some of the other benchmark functions. Evidently, this was not the case.

Conversely, the revised DFEA continued to do well on nine of 19 benchmarks even with a success rate of 80% (drop rate of 20%). The benchmarks were Dixon-Price, Eggholder, Exponential, Michalewicz, Rastrigin, Salomon, Sargan, Schaffer-F6, and Stretched-V. Strangely, on Michalewicz, although the difference was not statistically significant, the lower success rate of 80% actually led to an *increase* in performance.

On a few benchmarks, the revised DFEA did fairly well even with a success rate

Table 5.6: *Relaxing Consensus by Success Rates*

Benchmark	1.0	0.8	0.6	0.4
ackley-1	**2.78e-03** (3.82e-08, 6.99e-03)	3.34e-02 (2.08e-02, 4.52e-02)	1.10e-01 (9.15e-02, 1.28e-01)	4.61e-01 (3.58e-01, 5.86e-01)
brown	**1.21e-25** (5.11e-26, 1.98e-25)	3.53e-20 (3.02e-21, 8.93e-20)	7.02e-15 (9.67e-17, 2.31e-14)	1.05e+00 (1.09e-10, 3.29e+00)
dixon-price	**3.85e+01** (2.93e+01, 4.70e+01)	**1.62e+02** (2.81e+01, 3.57e+02)	1.05e+03 (5.62e+02, 1.54e+03)	1.78e+04 (9.39e+03, 2.88e+04)
eggholder	**-2.10e+04** (-2.13e+04, -2.07e+04)	**-2.05e+04** (-2.09e+04, -2.02e+04)	**-1.98e+04** (-2.03e+04, -1.94e+04)	-1.71e+04 (-1.77e+04, -1.66e+04)
exponential	**-1.00e+00** (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)	**-9.98e-01** (-1.00e+00, -9.93e-01)
griewank	**1.49e-01** (6.92e-02, 2.23e-01)	7.52e-01 (6.98e-01, 8.03e-01)	5.78e-01 (5.23e-01, 6.21e-01)	4.90e-01 (4.45e-01, 5.42e-01)
michalewicz	**-3.06e+01** (-3.07e+01, -3.04e+01)	**-3.08e+01** (-3.10e+01, -3.07e+01)	**-3.06e+01** (-3.08e+01, -3.04e+01)	-2.85e+01 (-2.87e+01, -2.82e+01)
rastrigin	**7.28e-02** (1.36e-02, 1.51e-01)	**1.89e-01** (5.97e-02, 3.45e-01)	**2.77e-01** (1.10e-01, 4.70e-01)	4.14e+00 (2.07e+00, 6.33e+00)
rosenbrock	**4.60e+01** (1.96e+01, 7.77e+01)	1.14e+03 (2.70e+02, 2.29e+03)	4.67e+03 (2.83e+03, 6.75e+03)	1.04e+05 (6.43e+04, 1.44e+05)
salomon	**1.96e+00** (1.78e+00, 2.12e+00)	**1.74e+00** (1.69e+00, 1.79e+00)	**1.90e+00** (1.85e+00, 1.96e+00)	2.44e+00 (2.37e+00, 2.52e+00)
sargan	**5.67e+02** (1.32e+02, 1.15e+03)	**2.80e+03** (9.57e+02, 4.94e+03)	8.00e+03 (4.44e+03, 1.22e+04)	1.23e+04 (9.67e+03, 1.50e+04)
schaffer-f6	**9.86e-01** (8.60e-01, 1.09e+00)	**9.52e-01** (8.56e-01, 1.06e+00)	1.38e+00 (1.27e+00, 1.48e+00)	2.32e+00 (2.09e+00, 2.56e+00)
schwefel-1.2	**6.53e+04** (4.59e+04, 8.85e+04)	3.22e+05 (2.86e+05, 3.55e+05)	2.97e+05 (2.74e+05, 3.24e+05)	2.75e+05 (2.56e+05, 2.95e+05)
schwefel-2.22	**1.35e-12** (5.74e-13, 2.40e-12)	2.94e-06 (9.51e-08, 7.99e-06)	5.25e+00 (1.90e+00, 9.10e+00)	5.72e+01 (4.40e+01, 7.15e+01)
schwefel-2.23	**5.07e-102** (2.39e-111, 1.67e-101)	5.22e+06 (1.27e+06, 1.00e+07)	1.02e+09 (6.05e+08, 1.56e+09)	4.07e+09 (2.89e+09, 5.48e+09)
sphere	**0.00e+00** (0.00e+00, 0.00e+00)	4.46e+01 (3.64e+01, 5.39e+01)	1.54e+02 (1.36e+02, 1.71e+02)	3.75e+02 (3.50e+02, 4.02e+02)
stretched-v	**4.24e+00** (3.91e+00, 4.54e+00)	**4.16e+00** (3.82e+00, 4.46e+00)	5.40e+00 (5.05e+00, 5.76e+00)	9.26e+00 (8.70e+00, 9.82e+00)
whitley	**5.35e+02** (4.84e+02, 5.80e+02)	9.54e+02 (9.02e+02, 1.00e+03)	5.52e+03 (4.28e+03, 7.94e+03)	4.38e+05 (3.31e+05, 5.62e+05)
zakharov	**8.16e+02** (7.80e+02, 8.45e+02)	4.66e+03 (1.29e+03, 1.35e+04)	1.19e+04 (1.47e+03, 2.48e+04)	5.14e+03 (9.58e+02, 9.91e+03)

of 60%, which is fairly low (a drop rate of 40% per variable, per iteration). These benchmarks were Eggholder, Exponential, Michalewicz, Rastrigrin, and Salomon. On a single benchmark, Exponential, the revised DFEA's performance was statistically indistinguishable whether the success rate was 0%, 20%, 40% or 60%.

These results seem to suggest that some tolerance to noise is acceptable for some problems but that full consensus is best during the Reconcile Step.

5.5 Conclusions

In the previous chapter we developed a framework based on information exchange and conflict resolution. Information exchange is modeled as a blackboard architecture and four operations: insert, merge, retrieve and delete. The conflict resolution mechanism is based on Pareto efficiency. We also developed a notation for looking at patterns in the temporal evolution of the blackboard. In Chapter 3 we developed the Distributed Factored Evolutionary Algorithms, DFEA.

Like DOSI before it, DFEA did not always perform as expected when compared to its centralized counterpart, in this case, FEA. In this chapter we applied our framework to DFEA in order to discover the cause of the discrepancy. We were able to determine that the order of reconciliation in the original DFEA differed markedly from that in FEA. We presented a revised version of DFEA that reproduced FEA exactly. We also looked at what relaxing consensus might look like under this revised DFEA.

Chapter 6

Pareto Improving Particle Swarm Optimization

In Chapter 4 we developed a framework around the Blackboard architecture that emphasized information exchange and conflict resolution. We then applied this framework to FEA and, in Chapter 3, to DFEA. In this chapter we apply this framework to *gbest* PSO itself and develop a single-population version of *gbest* PSO that does not exhibit hitchhiking.

6.1 *gbest* and Blackboards

At least for PSO, hitchhiking does not arise in a problem of one dimension. Either $f(x_1')$ is better than $f(x_1)$ or is it is not. For problems of two or more dimensions, hitchhiking becomes possible.

Let us revisit the example of two dimensions from Chapter 2. When comparing any given particle's *pbest* with the current *gbest*, we have two possibilities for each variable and four for the pairs of variables. We know that x_1 is either better or worse and that x_2 is either better or worse than their previous values. This leads to three zones for potential successor *gbest* values (the current set of *pbest*s). These zones and the implications for Pareto improvements in the selection of a new *gbest* are illustrated in Figure 6.1.

In this figure the arcs represent the contours of the Sphere function for two variables, x_1 and x_2. The current $gbest = (4.2, 3.9)$ also defines a contour (dotted) that is the dividing line between *pbest*s that have a better fitness (a lower contour) or a worse fitness (a higher contour). Additionally, the gray areas denote the set of points where the *pbest* lies on a lower contour than the *gbest* and thus has a better fitness but one or the other of the variables is larger than its value in *gbest*. All points in the gray zones include hitchhiking. We can thus see that *pbest*s C, D, E, F are all inferior to the current *gbest*. Points A, B and G involve hitchhikers. Only Point H has both a better fitness and no hitchhiking.

Figure 6.1: *Selecting gbest in PSO (Sphere) - No Hitchhiking*

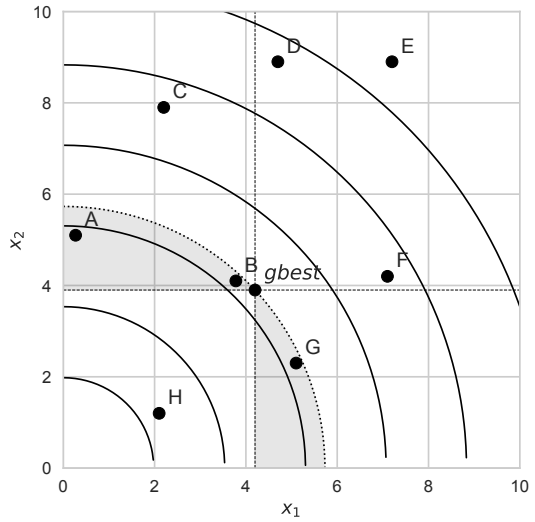


Figure 6.2: *Selecting gbest in PSO (Sphere) - Hitchhiking*

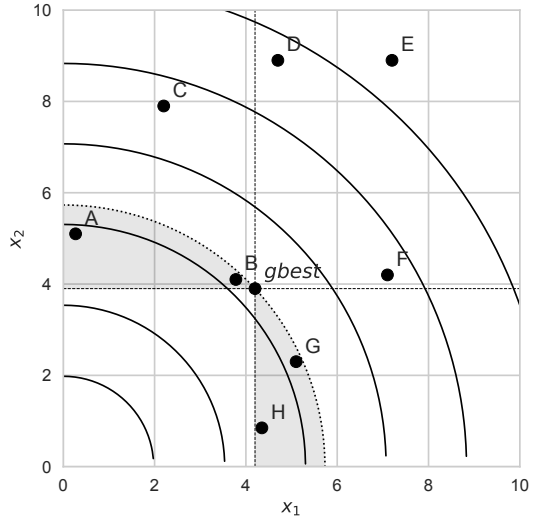


Table 6.1: *Hitchhiking in PSO*

$pbest_j$	$\mathbf{X}$	$f(\mathbf{x})$
$gbest_{current}$	[2.39, 1.24, 5.71, 0.34]	39.97
1	[1.53, 1.84, 5.29, 0.59]	34.06
2	[0.42, 2.01, 4.76, 1.84]	30.26
3	[3.23, 0.72, 4.68, 0.47]	33.07
4	[2.83, 3.83, 2.71, 1.27]	31.64
$gbest_{new}$ (#2)	[0.42, 2.01, 4.76, 1.84]	30.26

Of course, it need not have worked out this way. Figure 6.2 presents an alternate outcome where H has the values $(4.25, 0, 85)$. Although H lies on a lower contour than the current *gbest* and thus has a better fitness, the value for x_1 is worse than the value in the current *gbest* and is thus hitchhiking.

We also looked at a $4d$ example for the Sphere function, reproduced here in Table 6.1. In this table we can see directly that hitchhiking arises because the $gbest_{current}$ is replaced wholesale by a successor that is more fit overall but potentially loses valuable information.

In previous chapters we developed the argument that FEA and similar algorithms mitigate hitchhiking, not because of cooperation, but because the context $\mathbf{c}$ is treated as a blackboard for information exchange and because of how the merge operation resolves conflicting information in the multi-swarm (or, more generally, multi-population) setting. Any conflicting information is reconciled in such a way that $\mathbf{c}_{new}$ is always a Pareto improvement. Our previous example is shown again in Table 6.2. Here we can once again see that the problems of the single swarm version

Table 6.2: *FEA-PSO Determination of C_{new} with Non-overlapping Factors*

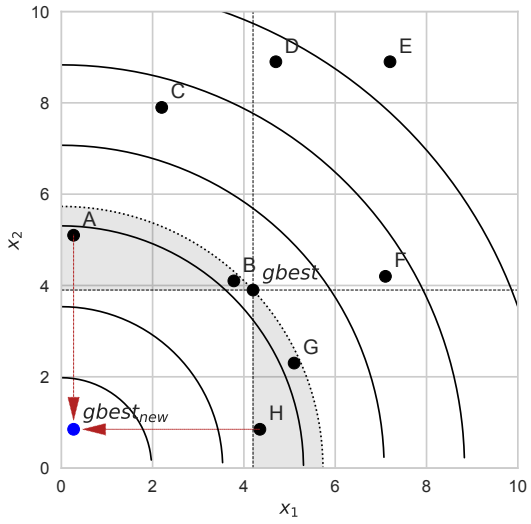
$gbest_j.\mathbf{x}$	$\mathbf{X}$	$f(\mathbf{x})$
C	[2.39, 1.24, 5.71, 0.34]	39.97
S_1	[1.53, ----, ----, ----]	36.59
S_2	[----, 2.01, ----, ----]	42.47
S_3	[----, ----, 4.68, ----]	29.27
S_4	[----, ----, ----, 1.27]	41.47
C_{new}	[1.53, 1.24, 4.68, 0.34]	25.90

of PSO are avoided.

But we also identified a problem with the approach exemplified by Table 6.2. Problems that are inappropriately factored are possibly subject to *pseudo-optima* when the values of variables are highly interrelated (*epistasis*). It has been shown that overlapping factors mitigate this problem [9]. We even went so far as to suggest that one could always include one factor that covered all the variables. The challenge with this solution is that such a factor, covering the entire problem space as it does, would be subject to the hitchhiking we are trying to avoid.

If we look at both Table 6.1 and Table 6.2 together, however, a possible solution arises. The way PSO uses *gbest* certainly has the characteristics of a blackboard discussed in Chapter 4. It is used as a vehicle for information exchange between particles when particle velocities are updated (Algorithm 2.4, Line 4). If we think of how we might interpret the merge operation for this blackboard, it is "winner take all." We could, instead, apply our variable-by-variable Pareto improving merge operation from FEA to the *gbest* in PSO.

Figure 6.3: *Selecting gbest in PSO with new Merge operation*



6.2 PSO to PI-PSO

Although we have put our own interpretation on it, the *gbest* is generally interpreted to be something more like a cache of the best value seen so far in the swarm. We propose changing the role of the *gbest* in PSO from being a cache to being a blackboard. In addition, instead of applying a merge operation based on "winner take all," we apply the merge operation developed for FEA. Our context **c** becomes the *gbest*, and instead of reconciling conflicting information between *swarms*, we reconcile conflicting information between *particles*. Figure 6.3 shows how this leads to a outcome different from the one shown in Figure 6.2. The $gbest_{new}$ is now a combination of *pbest* A's X_2 value and *pbest* H's X_1 value.

We hypothesize that if the global best were constructed in a similar way, the performance would be on a par or better than FEA, and certainly better than PSO. The reason for the first claim is that, by not factoring the variables, we would avoid pseudo-optima. The reason for the second claim is that, by using the global best as a blackboard and resolving information conflicts between particles in a Pareto improving way, we avoid hitchhiking. We call this algorithm Pareto Improving Particle Swarm Optimization (PI-PSO).

The difference between PSO and PI-PSO is fairly minimal, but we claim that the effect is significant. The basic PSO algorithm remains exactly the same except in how the global best is constructed. The pseudocode is shown in Algorithm 6.17. Basically, the algorithm takes the current global best, all the current personal bests and begins by taking x_1 out of each particle's personal best and trying it in the global best. At the end of the loop, X_1 either has the value we started with or it is the best one out of all the particles. This process repeats for all remaining variables, $X_2 ... X_d$. Unlike FEA-PSO, however, we do not practice *elitism* by replacing the worst particle in the swarm with our *gbest*. This means that the *gbest* may not necessarily ever be an actual particle or *pbest* from the swarm.

Unfortunately, as we have previously mentioned, we cannot be guaranteed of a Pareto efficient *gbest*. If we have a swarm of 320 particles with 32 dimensions, we would need to test 320^{32} combinations to find the best use of the information contained

Algorithm 6.17 PI-PSO Select Global Best

Input: Function f to optimize, current global best p_{gbest}, current personal bests, $\mathbf{p_{pbest}}$.
Output: New p_{gbest}.

```
1: for i = 1 to d do
2:     fitness ← f(p_gbest)
3:     value ← p_gbest.x_i
4:     for j = 1 to n do
5:         candidate ← p^j_pbest.x_i
6:         p_gbest.x_i ← candidate
7:         candidate_fitness ← f(p_gbest)
8:         if candidate_fitness is better than fitness then
9:             fitness ← candidate_fitness
10:            value ← candidate
11:        end if
12:        p_gbest.x_i ← value
13:    end for
14: end for
15: return p_gbest
```

on each variable in the swarm. So while we admit the theoretical existence of a Pareto *Efficient* Particle Swarm Optimization (PE-PSO) algorithm, we hope to work towards better approximations of it. We will return to a discussion of this PE-PSO in the conclusion.

As previously discussed in Chapter 3, the complexity of an EA like PSO is:

$$\mathcal{O}(PSO) = \mathcal{O}(p\Lambda(d))$$

The complexity of PI-PSO is now dominated by the more complicated global best update. The algorithm iterates over every individual (p of them) for each variable (d

of them). This puts the algorithmic complexity of PI-PSO at:

$$\mathcal{O}(PSO) = \mathcal{O}(dp\Lambda(d))$$

We will discuss this increased complexity below.

6.3 Comparing PSO, FEA-PSO, and PI-PSO

In order to test our hypothesis that PI-PSO would be at least as good as FEA-PSO and better than basic PSO, we ran a large number of experiments on standard benchmark functions. The following sections describe the design, results, and discussion of those experiments.

6.3.1 Design

We selected Benchmark optimization problems from [8] and [18] (See Appendix A more information the benchmark functions). The problems selected are shown in Table 6.3, arranged by categories inspired by [58]. All of the problems are minimization problems, and almost all have a minimizing solution and value of $f([0]^d) = 0$. The notable exceptions are Exponential, which has a minimum at $[-1]^d$, and Eggholder,

Table 6.3: *Benchmark Optimization Functions by Category*

Category	**Benchmark Function**
Bowl	Exponential, Sargan, Sphere
Many Local Optima	Ackley-1, Eggholder, Griewank, Rastrigin, Salomon, Stretched-V
Plate	Brown, Schwefel-2.23, Whitley, Zakharov
Ridge	Michalewicz, Schaffer-F6, Schwefel-2.22
Valley	Dixon-Price, Rosenbrock, Schefel-1.2

which has a dimension-dependent minimum and minimizing vector. Additionally, Sphere is *separable*, which means each dimension could be optimized individually. All of the other functions are non-separable in their current forms.

Each algorithm was run against each problem 50 times, and the average minimum value discovered was recorded. Each problem was instantiated with 32 dimensions, $d = 32$. The results were then bootstrapped 500 times to estimate 95% confidence intervals/credible intervals [59]. Following [60] each algorithm used the same number of candidate solutions. In this case we chose 10 particles per dimension or $d \times 10 = 320$.

The PSO, PI-PSO, and PSO portion of FEA-PSO all used the same parameters: $\omega = 0.729$ and $\phi_1 = \phi_2 = 1.49618$. Both PSO and PI-PSO were run for 100 iterations. FEA-PSO was run for 20 FEA iterations separated by 5 PSO iterations for a total of 100 PSO iterations. The FEA-PSO used a "Simple Centered" factor of $i, i + 1$ which followed the functional form of most of the benchmark functions—they are functions of adjacent x values—and shown by Strasser *et al.* to perform well [9]. With $d-1$ such factors, and $d = 32$, there were $\lfloor (320/31) \rfloor = 10$ particles per swarm for FEA-PSO.

Table 6.4: *"Bowl" Results - PSO, FEA-PSO, and PI-PSO*

Benchmark	PSO	FEA-PSO	PI-PSO
exponential	-9.99e-01 (-1.00e+00, -9.99e-01)	*-1.00e+00* (-1.00e+00, -1.00e+00)	*-1.00e+00* (-1.00e+00, -1.00e+00)
sargan	9.76e+00 (8.55e+00, 1.11e+01)	**2.04e-12** (1.37e-12, 2.91e-12)	1.55e-06 (1.16e-06, 2.00e-06)
sphere	*0.00e+00* (0.00e+00, 0.00e+00)	*0.00e+00* (0.00e+00, 0.00e+00)	*0.00e+00* (0.00e+00, 0.00e+00)

Table 6.5: *"Many Local Optima" Results - PSO, FEA-PSO, and PI-PSO*

Benchmark	PSO	FEA-PSO	PI-PSO
ackley-1	1.84e+00 (1.77e+00, 1.90e+00)	2.81e-03 (5.23e-06, 7.00e-03)	**4.44e-16** (4.44e-16, 4.44e-16)
eggholder	-1.68e+04 (-1.71e+04, -1.64e+04)	-1.77e+04 (-1.79e+04, -1.74e+04)	**-2.38e+04** (-2.40e+04, -2.35e+04)
griewank	4.96e-01 (4.47e-01, 5.45e-01)	9.49e-01 (8.99e-01, 9.91e-01)	**1.08e-01** (8.07e-02, 1.40e-01)
rastrigin	1.03e+02 (9.59e+01, 1.11e+02)	2.60e-02 (1.97e-05, 6.59e-02)	**0.00e+00** (0.00e+00, 0.00e+00)
salomon	**1.41e+00** (1.33e+00, 1.48e+00)	2.29e+00 (2.12e+00, 2.49e+00)	1.69e+00 (1.57e+00, 1.79e+00)
stretched-v	1.20e+01 (1.14e+01, 1.28e+01)	4.37e+00 (3.78e+00, 5.02e+00)	**2.84e+00** (2.59e+00, 3.04e+00)

Table 6.6: *"Plate" Results - PSO, FEA-PSO, and PI-PSO*

Benchmark	PSO	FEA-PSO	PI-PSO
brown	7.56e+00 (6.69e+00, 8.55e+00)	**1.22e-25** (3.72e-26, 2.42e-25)	1.23e-09 (1.01e-09, 1.45e-09)
schwefel-2.23	2.44e-01 (1.18e-01, 4.15e-01)	*8.65e-102* (7.25e-116, 2.31e-101)	*5.35e-41* (0.00e+00, 1.63e-40)
whitley	9.82e+02 (9.67e+02, 9.99e+02)	3.51e+02 (2.97e+02, 3.91e+02)	**7.51e+00** (6.20e+00, 8.76e+00)
zakharov	*1.39e+02* (1.28e+02, 1.51e+02)	1.60e+03 (3.09e+02, 3.77e+03)	*1.41e+02* (1.28e+02, 1.57e+02)

Table 6.7: *"Ridge" Results - PSO, FEA-PSO, and PI-PSO*

Benchmark	PSO	FEA-PSO	PI-PSO
michalewicz	-8.65e+00 (-9.02e+00, -8.33e+00)	-2.59e+01 (-2.63e+01, -2.56e+01)	**-3.19e+01** (-3.19e+01, -3.19e+01)
schaffer-f6	2.49e+00 (2.31e+00, 2.67e+00)	1.99e+00 (1.78e+00, 2.20e+00)	**4.85e-01** (4.20e-01, 5.40e-01)
schwefel-2.22	3.01e+02 (2.91e+02, 3.13e+02)	1.22e-12 (7.73e-13, 1.75e-12)	**0.00e+00** (0.00e+00, 0.00e+00)

Table 6.8: *"Valley" Results - PSO, FEA-PSO, and PI-PSO*

Benchmark	PSO	FEA-PSO	PI-PSO
dixon-price	4.23e+01 (2.54e+01, 6.34e+01)	1.18e+02 (1.07e+02, 1.28e+02)	**8.65e-05** (7.44e-05, 9.85e-05)
rosenbrock	1.95e+02 (1.56e+02, 2.51e+02)	2.20e+02 (1.69e+02, 2.88e+02)	**9.64e-01** (6.73e-01, 1.26e+00)
schwefel-1.2	7.96e+03 (7.19e+03, 8.84e+03)	6.59e+04 (5.72e+04, 7.60e+04)	**3.28e+02** (2.86e+02, 3.68e+02)

6.3.2 Results

The results of the experiments are reported in Tables 6.4-6.8 for the mean minimum found by each algorithm during each experiment. The 95% confidence/credible intervals are show in parentheses. The algorithms with the best performance are shown in bold, if there is a clear winner, or italics for multiple winners. We now examine the results by Benchmark category.

It is no surprise that all the algorithms did well on the Bowl benchmarks (Exponential, Sargan, and Sphere). All of these functions have a single global optimum without any trapping local optima. All three algorithms tied on the Sphere function, learning it perfectly without variance. The same cannot be said for Sargan

where FEA-PSO did better than the others, or for Exponential where FEA-PSO and PI-PSO were tied for best performance. In all cases, PI-PSO did better than PSO.

In stark contrast to the Bowl benchmarks, the six benchmark functions in the Local Optima category are highly irregular (Ackley-1, Eggholder, Griewank, Rastrigin, Salomon, and Stretched-V). Consistent with previous research [9], FEA-PSO usually did better than PSO (four out of six). However, PI-PSO beat the other algorithms on five of six benchmarks.

The Plate benchmarks (Brown, Schwefel-2.23, Whitley, and Zakharov) have large, flat plateaus over their domains with spiky minima. FEA-PSO and PI-PSO each split these benchmarks with FEA-PSO performing better on Brown and PI-PSO performing better on Whitley. PI-PSO tied on each of the other functions. On Schwefel-2.23, PI-PSO tied with FEA-PSO. On Zakharov, PI-PSO tied with PSO. PI-PSO did better than PSO on all benchmarks except Zakharov.

Michalewicz, Schaffer-F6, and Schwefel-2.22 are all Ridge benchmarks characterized by sharp drop offs at various points in their domain. The PI-PSO did better on all of these benchmarks than the other algorithms.

Finally, we have the Valley benchmark functions that look like a tilted tube sawed in half (Dixon-Price, Rosenbrock, Schwefel-1.2). The PI-PSO again did better than all the other algorithms on these problems.

Overall, PI-PSO was the strongest performing algorithm. It was the best perform-

ing algorithm in 12 out of 19 benchmarks with three ties (Exponential, Sphere, Zakharov) and four losses (Sargan, Salomon, Brown, Schwefel-2.23). Comparing solely to PSO, however, PI-PSO was the best performer for 16 of 19 benchmarks with two ties (Sphere, Zakharov) and one loss (Salomon). PI-PSO performed better than FEA-PSO on 15 of 19 benchmarks, losing twice (Brown, Sargan) and tying twice (Sphere and Schwefel-2.23).

On the other hand, FEA-PSO was better than PSO on 12 of 19 benchmarks, tying twice (Sphere, Rosenbrock) and losing five times (Dixon-Price, Griewank, Salomon, Schwefel-1.2, Zakharov). PSO performed best only on Salomon, tied on three (Sphere, Zakharov, Rosenbrock), and lost the rest.

6.3.3 Discussion

As we hypothesized, PI-PSO was much better than PSO, beating the basic algorithm on 16 of 19 benchmarks. We attribute this both to the elimination of hitchhiking and the lack of pseudo-minima in PI-PSO.

PI-PSO also did well when compared to FEA-PSO, beating it on 15 of 19 benchmarks. Given that FEA-PSO did so well against PSO (12 out of 19 benchmarks), it is difficult to attribute all of this success to pseudo-minima in FEA-PSO that were avoided in PI-PSO, though this may have played a part. What we hypothesize instead is that although both algorithms use the same basic algorithm to construct

their context, **c** for FEA-PSO and *gbest* for PI-PSO, PI-PSO has significantly more information to evaluate because instead of working with relatively few factors representing only partial solutions, PI-PSO works with many particles representing entire solutions.

Although all the algorithms start with the same number of candidate solutions, PI-PSO definitely requires more fitness evaluations. If p is the number of particles, d the number of dimensions, s the number of swarms (factors), m the number of FEA iterations, and i the average width of a factor, then our estimates of fitness evaluations per PSO iteration are:

$$\begin{aligned} \text{PSO} &= p \\ \text{FEA-PSO} &= ps + \frac{di + ps + 1}{m} \\ \text{PI-PSO} &= dp + p + 1 \end{aligned}$$

Given the difficulty of comparing algorithms with different structures, information, and information processing, it is difficult to say that fitness evaluations are a fair means of comparison, which is why we looked at candidate solutions. Different algorithms use the information differently. However, ultimately all factors should be presented so that users of these algorithms can make informed choices.

6.4 Conclusions

In this chapter, we introduced the Pareto Improving Particle Swarm Optimization (PI-PSO) algorithm. The algorithm is built on the ideas of information exchange via a blackboard architecture and conflict resolution taken from FEA. However, instead of applying these concepts to multiple populations in the construction of a shared context, $\mathbf{c}$, we applied them to the particles of a single swarm in the construction of the *gbest*. We hypothesized that this PI-PSO would perform better than regular *gbest* PSO and on a par with FEA-PSO. In order to test our hypothesis we ran multiple experiments on PSO, FEA-PSO and PI-PSO. PI-PSO outperformed PSO in 16 out of 19 functions.

Chapter 7

Comparative Scaling and Performance of PI-PSO

In this chapter, we further explore PI-PSO developed in the previous chapter. Specifically, we look at the comparative scaling and performance characteristics of PI-PSO relative to PSO.

7.1 Introduction

In the previous chapter, we ran experiments on 19 standard benchmark optimization functions for a given dimension, $d = 32$. Additionally, following Engelbrech [60], we evaluated the stochastic optimization algorithms by using the same number

of candidate solutions for each algorithm. Under those conditions, we showed that PI-PSO outperforms PSO. In this chapter, we want to explore the broader scaling and performance characteristics of PI-PSO relative to PSO. In order to do this, we will take advantage of the fact that the benchmark functions we chose are scalable to different dimensionalities. Additionally, we will look at the performance of both algorithms with different numbers of candidate solutions.

7.2 PSO and PI-PSO with Different Population Sizes and Problem Dimensions

In the previous chapter we showed that PI-PSO performed well on standard benchmark problems, besting the *gbest* PSO (or just "PSO" hereafter) on 16 out of 19 of them. However, by looking only at problems with 32 dimensions and using only 10 particles per dimension, we are left unsure of PI-PSO's comparative performance *vis-a-vis* PSO as dimensions increase and as differing numbers of particles are used. The purpose of the experiments in this chapter is to expand on the initial results of the previous chapter and test the comparative performance and scalability of PI-PSO by varying both problem dimension and swarm size.

A very general interpretation of the curse of dimensionality suggests that, for a given level of performance on a 4-dimensional problem with 16 particles, we would

need 256 particles to achieve the same level of performance on an 8-dimensional problem. And to achieve that same level of performance on a 16-dimensional problem, we would need 65,536 particles. But these requirements quickly become untenable with a 32-dimensional problem where we would theoretically require 4,294,967,296 particles. Of course, there are attenuating factors, such as the simplicity of the problem and the nature of our algorithm, but NFLT suggests we cannot expect to surmount these difficulties across all problems.

7.2.1 Design

We use the same benchmark functions as we have used throughout this dissertation from [8, 18] (See Appendix A more information the benchmark functions). The benchmark functions are presented in Table 6.3 by categories suggested by [58]. These are all minimization problems with the same solution, $[0]^d$, except for Exponential, Eggholder, and Michalewicz. Departing from [60], we are specifically interested in the comparative performance of these algorithms with different numbers of candidates. Thus, for these experiments, a "problem" consisted of a benchmark optimization function, a dimensionality, and a number of particles per dimension. The range of particles per dimension was $\{2, 4, 8, 16, 32, 64, 128, 256\}$ and the number of dimensions was $\{4, 8, 16, 32\}$. Each algorithm was run against each problem 50 times, and the average minimum value discovered was recorded. Because of the asymmetries en-

Table 7.1: *Benchmark Optimization Functions by Category*

Category	**Benchmark Function**
Bowl	Exponential, Sargan, Sphere
Many Local Optima	Ackley-1, Eggholder, Griewank, Rastrigin, Salomon, Stretched-V
Plate	Brown, Schwefel-2.23, Whitley, Zakharov
Ridge	Michalewicz, Schaffer-F6, Schwefel-2.22
Valley	Dixon-Price, Rosenbrock, Schefel-1.2

countered in optimization problems, we opted to use bootstrap estimates of the 95% confidence intervals/credible intervals [59]. PSO and PI-PSO both used the same parameters: $\omega = 0.729$ and $\phi_1 = \phi_2 = 1.49618$ with 100 iterations.

Such a large number of experiments gives rise to an even larger matrix of possible outcomes. Although we believe that PI-PSO both eliminates hitchhiking and better exploits the information in the swarm, which should make it better than PSO in general, we do not believe that it will be comparatively better in all cases. We already know from previous results that PI-PSO did not perform as well as PSO on some problems. As a result, we make the following hypotheses:

1. **Hypothesis I** – On a given problem, with a given dimension, we expect PI-PSO to perform better than PSO as particles per dimension increase. As the swarm gets larger, there is more information for PI-PSO to exploit. We are unsure what to expect with PSO. It would seem like, given our discussion on the curse of dimensionality, that as particles per dimension increase for PSO,

performance should increase as well but some have found that more particles can "get in the way" [60].

2. **Hypothesis II** – Because PI-PSO can better exploit the information in the swarm, the 2 particle per dimension (PPD) version of PI-PSO will outperform the 256 PPD version of PSO for a given problem of a given dimension. This comparison is may be seen as a proxy for an approximate run-time comparison of the two algorithms, at least in terms of fitness evaluations that heavily favors PSO. PSO requires $\mathcal{O}(p)$ fitness evaluations to select the *gbest* ($256 \times 32 = 8,192$ fitness evaluations) whereas PI-PSO requires $\mathcal{O}(pd)$ fitness evaluations ($2 \times 32 \times 32 = 2,048$ fitness evaluations).

   If desired, the charts can be used to make a closer comparison by using 4ppd for PI-PSO and 128ppd for PSO.

3. **Hypothesis III** – Because hitchhiking should theoretically be lower at lower dimensions and because there is relatively less information to exploit, the relative performance of PI-PSO will be the same as PSO on 4-dimensional versions of problems and then increase as dimensionality increases.

With 19 benchmarks $\times$ 4 dimensions $\times$ 8 particles per dimension $\times$ 3 metrics, we have 1,824 individual results. To limit the number of tables of results, we present categories of results along with examples of each (Tables for all $32d$ results and charts

for all results are in Appendix B). While our hypotheses were supported overall, the deviations are perhaps the most interesting so we will concentrate more on those.

All tables showing results for a single benchmark function are for the 32-dimensional version of the function. They include results for both PSO and PI-PSO. Each row shows results for a given PPD showing the mean discovered minimum as well as the 95% confidence interval for the mean. The summary table shows results for all benchmark functions (Table 7.3) but only for 2 and 256 PPD.

All figures show results for all dimensions of the problems at all particles per dimension. The blue dots and lines are for PSO. The red dots and lines are for PI-PSO. The dots are the mean discovered minimum. The lines represent the 95% confidence interval. The x-axis is the range of mean discovered minima and are not necessarily the same for all charts within a figure.

7.2.2 Results - Hypotheses I and II

Our findings for the Ackley-1 benchmark function (Table B.1) are typical for benchmark results that support both Hypothesis I and Hypothesis II. For PSO, the $2d$ version had a mean discovered minimum of 2.42 (2.35, 2.49) and the 256d version had a mean discovered minimum of 1.14 (1.08, 1.20). For PI-PSO, the $2d$ version results were 1.54e-04 (1.20e-04, 1.88e-04) and $256d$ version results were 4.44e-16 (4.44e-16, 4.44e-16). Looking at Table B.1 we see that performance progressively improved for

Table 7.2: *Ackley-1 Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	2.42e+00 (2.35e+00, 2.49e+00)	**1.54e-04** (1.20e-04, 1.88e-04)
4 (128)	2.14e+00 (2.08e+00, 2.20e+00)	**1.01e-05** (5.73e-06, 1.54e-05)
8 (256)	1.93e+00 (1.86e+00, 1.98e+00)	**1.11e-07** (4.44e-16, 3.32e-07)
16 (512)	1.85e+00 (1.77e+00, 1.92e+00)	**4.44e-16** (4.44e-16, 4.44e-16)
32 (1024)	1.65e+00 (1.58e+00, 1.72e+00)	**4.44e-16** (4.44e-16, 4.44e-16)
64 (2048)	1.41e+00 (1.33e+00, 1.49e+00)	**4.44e-16** (4.44e-16, 4.44e-16)
128 (4096)	1.25e+00 (1.18e+00, 1.30e+00)	**4.44e-16** (4.44e-16, 4.44e-16)
256 (8192)	1.14e+00 (1.08e+00, 1.20e+00)	**4.44e-16** (4.44e-16, 4.44e-16)

both PSO and PI-PSO as the number of particles per dimension increased. Additionally, the performance of the 2 PPD version of PI-PSO (1.54e-04) was significantly better than the performance of the 256 PPD version of PSO (1.14e+00).

Results for all of the benchmark functions are presented in Table 7.3. Each benchmark function has a row showing the performance of both PSO and PI-PSO for 2 and 256 PPD. These results are for the $32d$ version of the benchmark function. The table is divided into four sections: those results that support both Hypothesis I and II, those that support only Hypothesis I, those that support only Hypothesis II, and those results that support neither hypothesis. Recall that Hypothesis I is that performance increases as PPD increases for both algorithms, and Hypothesis II is that the performance of PI-PSO at 2 PPD will be better than PSO 256 PPD. Thus we can see that the results support Hypotheses I and II in 12 of 19 (63%) cases.

Hypothesis I was supported but not Hypothesis II in four cases: Exponential (Table B.5), Schwefel-1.2 (Table B.14), Sphere (Table B.17, Figure 7.2), and Zakharov

Table 7.3: *Summary of Hypothesis I and II Results*

	PSO		PI-PSO	
Benchmark	**2 ppd**	**256 ppd**	**2 ppd**	**256 ppd**
Hypothesis I and II				
ackley-1	2.42e+00 (2.35e+00, 2.49e+00)	1.14e+00 (1.08e+00, 1.20e+00)	1.54e-04 (1.20e-04, 1.88e-04)	4.44e-16 (4.44e-16, 4.44e-16)
brown	1.95e+01 (1.42e+01, 2.50e+01)	1.57e+00 (1.15e+00, 2.01e+00)	2.16e-07 (1.62e-07, 2.87e-07)	2.90e-13 (2.36e-13, 3.57e-13)
dixon-price	2.34e+02 (1.82e+02, 2.86e+02)	7.07e+00 (1.21e+00, 1.41e+01)	1.03e-01 (6.61e-03, 3.02e-01)	4.32e-08 (3.62e-08, 5.33e-08)
eggholder	-1.52e+04 (-1.56e+04, -1.49e+04)	-1.86e+04 (-1.90e+04, -1.83e+04)	-2.30e+04 (-2.33e+04, -2.27e+04)	-2.37e+04 (-2.40e+04, -2.33e+04)
michalewicz	-7.42e+00 (-7.77e+00, -7.13e+00)	-1.07e+01 (-1.12e+01, -1.03e+01)	-3.18e+01 (-3.19e+01, -3.18e+01)	-3.19e+01 (-3.19e+01, -3.19e+01)
rastrigin	1.56e+02 (1.46e+02, 1.66e+02)	5.54e+01 (5.09e+01, 6.02e+01)	4.54e-06 (1.39e-06, 9.46e-06)	0.00e+00 (0.00e+00, 0.00e+00)
rosenbrock	8.56e+02 (6.13e+02, 1.28e+03)	6.38e+01 (5.03e+01, 7.84e+01)	1.56e+01 (7.57e+00, 2.46e+01)	1.20e+00 (7.09e-01, 1.82e+00)
sargan	9.83e+01 (8.47e+01, 1.12e+02)	1.22e-01 (1.04e-01, 1.40e-01)	6.93e-04 (5.52e-04, 8.65e-04)	2.42e-10 (1.03e-12, 8.31e-10)
schaffer-f6	3.49e+00 (3.26e+00, 3.73e+00)	1.83e+00 (1.69e+00, 1.95e+00)	6.12e-01 (5.42e-01, 6.90e-01)	3.98e-01 (3.44e-01, 4.55e-01)
schwefel-2.22	4.70e+02 (4.52e+02, 4.89e+02)	1.09e+02 (9.45e+01, 1.23e+02)	3.14e-04 (0.00e+00, 9.03e-04)	0.00e+00 (0.00e+00, 0.00e+00)
schwefel-2.23	3.59e+02 (1.53e+02, 6.52e+02)	2.73e-09 (6.32e-10, 5.86e-09)	8.61e-25 (1.89e-25, 1.92e-24)	0.00e+00 (0.00e+00, 0.00e+00)
stretched-v	1.60e+01 (1.52e+01, 1.68e+01)	9.20e+00 (8.69e+00, 9.72e+00)	3.52e+00 (3.21e+00, 3.83e+00)	3.01e+00 (2.67e+00, 3.38e+00)
Hypothesis I only				
exponential	-9.93e-01 (-9.94e-01, -9.92e-01)	-1.00e+00 (-1.00e+00, -1.00e+00)	-1.00e+00 (-1.00e+00, -1.00e+00)	-1.00e+00 (-1.00e+00, -1.00e+00)
schwefel-1.2	1.69e+04 (1.54e+04, 1.84e+04)	1.23e+03 (9.41e+02, 1.50e+03)	1.52e+03 (1.37e+03, 1.66e+03)	2.68e+02 (2.40e+02, 2.96e+02)
sphere	2.73e+01 (1.20e+01, 4.20e+01)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
zakharov	3.74e+02 (3.42e+02, 4.04e+02)	2.29e+01 (1.91e+01, 2.71e+01)	2.29e+02 (2.08e+02, 2.45e+02)	1.37e+02 (1.25e+02, 1.49e+02)
Hypothesis II only				
griewank	1.03e+00 (1.02e+00, 1.04e+00)	2.66e-02 (2.10e-02, 3.32e-02)	1.65e-02 (1.00e-02, 2.33e-02)	2.16e-01 (1.88e-01, 2.44e-01)
whitley	5.52e+03 (4.03e+03, 7.26e+03)	6.80e+02 (6.65e+02, 6.96e+02)	3.59e+01 (1.70e+01, 5.96e+01)	3.93e+02 (3.30e+02, 4.47e+02)
Neither				
salomon	3.18e+00 (2.93e+00, 3.54e+00)	4.14e-01 (3.96e-01, 4.30e-01)	1.79e+00 (1.67e+00, 1.91e+00)	1.91e+00 (1.82e+00, 2.02e+00)

(Table B.20, Figure 7.8). Using the categories from Table 7.1, we find no commonality between these four functions. Both Exponential and Sphere are considered to have a

Table 7.4: *Exponential Benchmark Results 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	-9.93e-01 (-9.94e-01, -9.92e-01)	**-1.00e+00** (-1.00e+00, -1.00e+00)
4 (128)	-9.98e-01 (-9.98e-01, -9.97e-01)	**-1.00e+00** (-1.00e+00, -1.00e+00)
8 (256)	-9.99e-01 (-9.99e-01, -9.99e-01)	**-1.00e+00** (-1.00e+00, -1.00e+00)
16 (512)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)
32 (1024)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)
64 (2048)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)
128 (4096)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)
256 (8192)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)

Table 7.5: *Schwefel-1.2 Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	1.69e+04 (1.54e+04, 1.84e+04)	**1.52e+03** (1.37e+03, 1.66e+03)
4 (128)	1.31e+04 (1.16e+04, 1.44e+04)	**6.73e+02** (6.07e+02, 7.39e+02)
8 (256)	1.07e+04 (9.56e+03, 1.19e+04)	**4.21e+02** (3.84e+02, 4.63e+02)
16 (512)	7.22e+03 (6.16e+03, 8.29e+03)	**3.03e+02** (2.65e+02, 3.38e+02)
32 (1024)	5.48e+03 (4.60e+03, 6.48e+03)	**3.05e+02** (2.80e+02, 3.33e+02)
64 (2048)	3.21e+03 (2.66e+03, 3.79e+03)	**3.22e+02** (2.80e+02, 3.64e+02)
128 (4096)	2.17e+03 (1.67e+03, 2.79e+03)	**2.74e+02** (2.51e+02, 3.00e+02)
256 (8192)	1.23e+03 (9.41e+02, 1.50e+03)	**2.68e+02** (2.40e+02, 2.96e+02)

Table 7.6: *Sphere Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	2.73e+01 (1.20e+01, 4.20e+01)	0.00e+00 (0.00e+00, 0.00e+00)
4 (128)	8.22e+00 (2.00e+00, 1.80e+01)	0.00e+00 (0.00e+00, 0.00e+00)
8 (256)	5.94e+00 (0.00e+00, 1.40e+01)	0.00e+00 (0.00e+00, 0.00e+00)
16 (512)	1.74e+00 (0.00e+00, 6.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
32 (1024)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
64 (2048)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
128 (4096)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
256 (8192)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)

general *Bowl* shape, Schwefel-1.2 is *Valley*-shaped and Zakharov is *Plate*-shaped.

In those cases where only Hypothesis II was supported (Griewank and Whitley),

Table 7.7: *Zakharov Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	3.74e+02 (3.42e+02, 4.04e+02)	**2.29e+02** (2.08e+02, 2.45e+02)
4 (128)	2.46e+02 (2.25e+02, 2.69e+02)	**1.90e+02** (1.69e+02, 2.11e+02)
8 (256)	1.68e+02 (1.54e+02, 1.83e+02)	1.53e+02 (1.42e+02, 1.64e+02)
16 (512)	**1.15e+02** (1.05e+02, 1.24e+02)	1.45e+02 (1.31e+02, 1.57e+02)
32 (1024)	**7.12e+01** (6.50e+01, 7.73e+01)	1.42e+02 (1.28e+02, 1.57e+02)
64 (2048)	**5.09e+01** (4.61e+01, 5.55e+01)	1.39e+02 (1.25e+02, 1.51e+02)
128 (4096)	**3.25e+01** (2.81e+01, 3.65e+01)	1.47e+02 (1.34e+02, 1.65e+02)
256 (8192)	**2.29e+01** (1.91e+01, 2.71e+01)	1.37e+02 (1.25e+02, 1.49e+02)

Table 7.8: *Griewank Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	1.03e+00 (1.02e+00, 1.04e+00)	**1.65e-02** (1.00e-02, 2.33e-02)
4 (128)	8.65e-01 (8.31e-01, 8.98e-01)	**1.03e-01** (7.75e-02, 1.30e-01)
8 (256)	5.90e-01 (5.40e-01, 6.42e-01)	**1.44e-01** (1.15e-01, 1.72e-01)
16 (512)	3.22e-01 (2.79e-01, 3.64e-01)	**1.75e-01** (1.46e-01, 2.08e-01)
32 (1024)	**1.49e-01** (1.27e-01, 1.73e-01)	1.97e-01 (1.65e-01, 2.29e-01)
64 (2048)	**9.10e-02** (7.36e-02, 1.09e-01)	1.77e-01 (1.46e-01, 2.05e-01)
128 (4096)	**4.13e-02** (3.16e-02, 5.15e-02)	1.60e-01 (1.25e-01, 1.92e-01)
256 (8192)	**2.66e-02** (2.10e-02, 3.32e-02)	2.16e-01 (1.88e-01, 2.44e-01)

the results for PSO actually improved as PPD increased but they did not for PI-PSO. Neither of these functions belongs to the same general shape class. Griewank has *Many Local Optima* while Whitley is *Plate*-shaped. The results for Griewank are shown in Table B.7.

The only results that supported neither hypothesis were those for Salomon (Table B.11). Like the results for Griewank and Whitley, increasing PPD for PI-PSO decreased performance (although performance for PSO increased as PPD increased). Additionally, the 256 PPD version of PSO outperformed the 2 PPD version of PI-

Table 7.9: *Salomon Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	3.18e+00 (2.93e+00, 3.54e+00)	**1.79e+00** (1.67e+00, 1.91e+00)
4 (128)	2.31e+00 (2.17e+00, 2.43e+00)	**1.74e+00** (1.61e+00, 1.87e+00)
8 (256)	**1.67e+00** (1.60e+00, 1.74e+00)	1.90e+00 (1.77e+00, 2.03e+00)
16 (512)	**1.18e+00** (1.11e+00, 1.23e+00)	1.87e+00 (1.74e+00, 1.97e+00)
32 (1024)	**8.55e-01** (8.18e-01, 8.94e-01)	1.88e+00 (1.79e+00, 1.98e+00)
64 (2048)	**6.39e-01** (6.14e-01, 6.64e-01)	1.85e+00 (1.74e+00, 1.95e+00)
128 (4096)	**5.18e-01** (4.92e-01, 5.40e-01)	1.79e+00 (1.66e+00, 1.91e+00)
256 (8192)	**4.14e-01** (3.96e-01, 4.30e-01)	1.91e+00 (1.82e+00, 2.02e+00)

PSO. However, there were some situations where PI-PSO outperformed PSO. The interesting "problem" is that at lower dimensions, those same PPDs did not outperform PSO.

7.2.3 Results - Hypothesis III

Hypothesis III relates to how the algorithms' relative performance as the dimension of the problem increased. Specifically, we hypothesized that at low dimensions ($4d$), the results of PI-PSO and PSO overall could be indistinguishable as lower dimensional problems are less likely to exhibit hitchhiking, other things being equal. However, we generally believed that at least Hypotheses I would continue to hold across all dimensions. For any given problem of a certain dimension, as one increases particles per dimension, the performance of both algorithms will increase. It is certainly possible that weaker versions of Hypothesis II might be true. For example,

Figure 7.1: *Ackley-1 Benchmark*

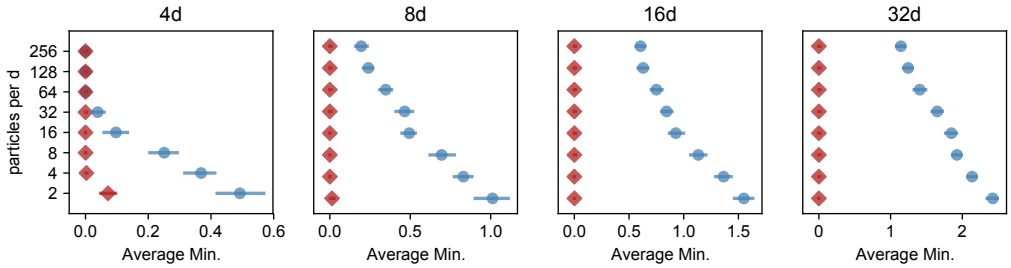
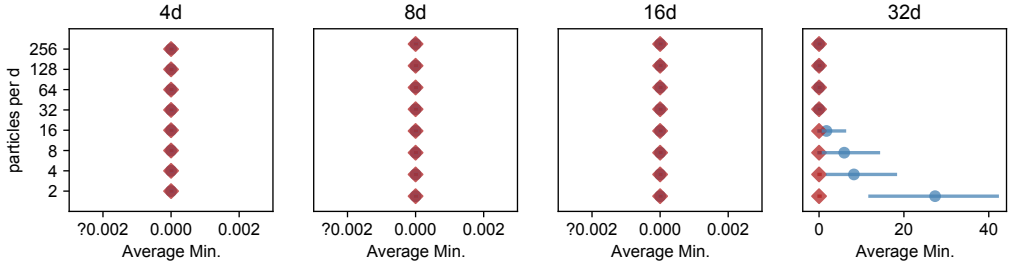


Figure 7.2: *Sphere Benchmark*



PI-PSO with 8 PPD, rather than 2 PPD, might exceed the performance of PSO at 256 PPD.

Figure 7.1 presents the results for the Ackley-1 benchmark function. With only minor variation, these results represent those cases where the results support Hypothesis III. The results for 12 of 19 (63%) benchmark functions strongly support Hypothesis III (Ackley-1, Brown, Dixon-Price, Eggholder, Michaelwicz, Rastrigin, Rosenbrock, Sargan, Schaffer-F6, Schwefel-1.2, Schwefel-2.22, and Stretched-V). The results for Stretched-V (Figure 7.5) are probably closer to those we envisioned. At $4d$, the performance of both algorithms are indistinguishable, but by the time we reach $32d$, there is a clear separation in favor of PI-PSO.

Another interesting case is the Sphere function (Figure 7.2). Much maligned

Figure 7.3: *Schwefel-2.23 Benchmark*

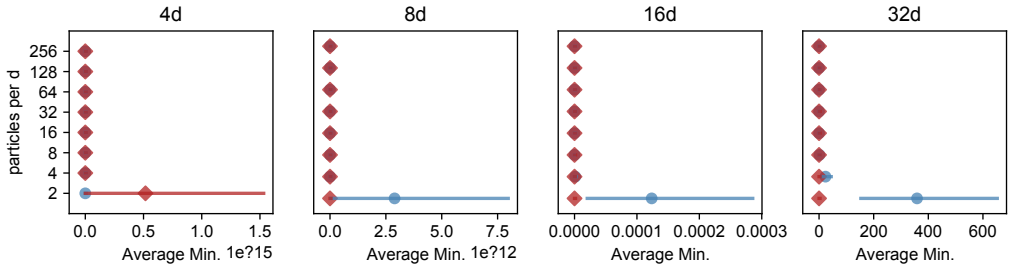
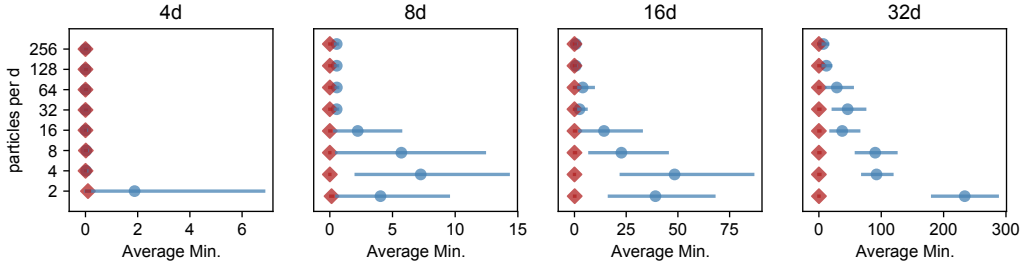


Figure 7.4: *Dixon-Price Benchmark*



for its simplicity, the results show that even this simple function is not immune to hitchhiking and the curse of dimensionality. Although the results for PSO and PI-PSO are nearly identical for problem sizes of $4d$ through $16d$, when we consider $32d$, there are lower particles per dimension where PSO starts to perform worse that PI-PSO. In the case of Exponential, Schwefel 2.23 (Figure 7.3), and Sphere, based on the emerging pattern in the data, it is possible that Hypothesis III is true in dimensions higher than 32.

The Dixon-Price benchmark function, Figure 7.4, exhibited some interesting characteristics as well. Although the overall trend as dimensions increased followed the expected pattern of Hypothesis III, PSO showed a much larger variance relative to PI-PSO on low PPD (2–16 PPD).

Figure 7.5: *Stretched-V Benchmark*

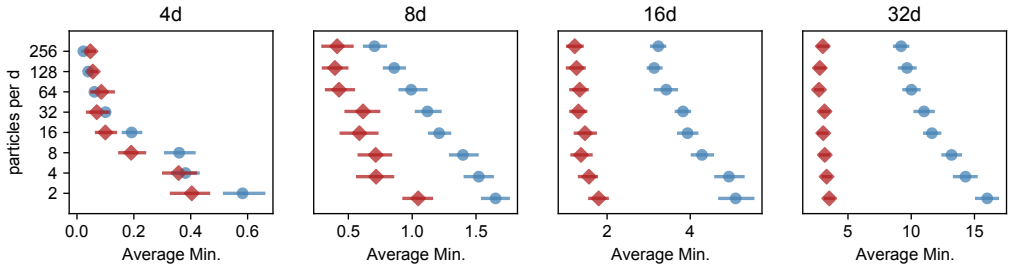
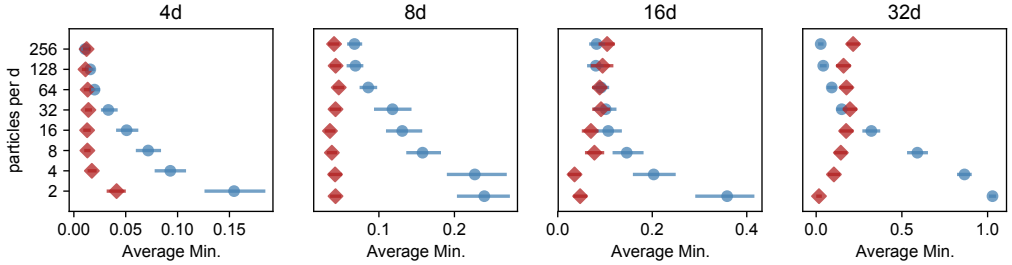


Figure 7.6: *Griewank Benchmark*



There were four benchmark functions whose results did not support Hypothesis III. In Figure 7.6, we see that the hypothesis generally holds for $4d$ and $8d$ versions of the problem but as we cross into $16d$ and certainly by $32d$, PI-PSO seems to exhibit aberrant behavior. Specifically, as the particles per dimension increase, the performance of PI-PSO deteriorates. Note that this is happening in the context of results that support Hypothesis II: PI-PSO with 2 PPD obtains the best results overall for the $32d$ problem.

We previously noted that the results for Salomon were interesting (Figure 7.7). They did not support Hypothesis I or II. They do not support Hypothesis III either. However, if one looks at the pattern in the results, one can imagine that as the dimensionality of the problem increases, all three hypothesis might be true. That is,

Figure 7.7: *Salomon Benchmark*

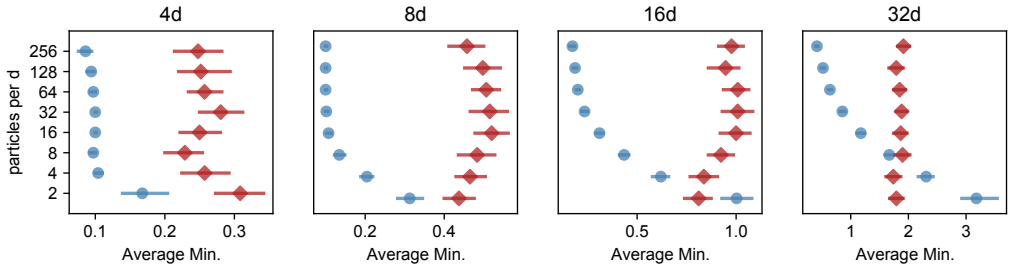


Figure 7.8: *Zakharov Benchmark*

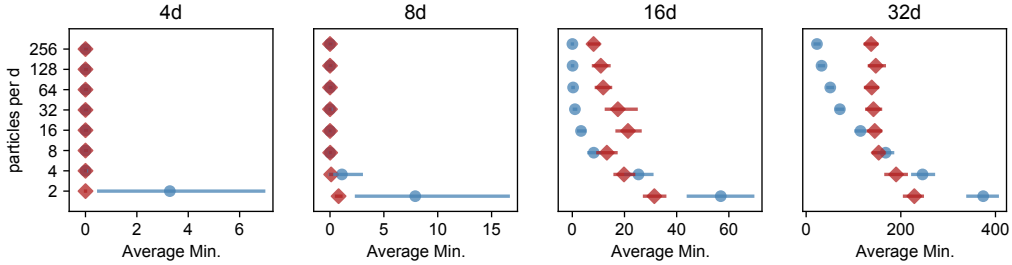
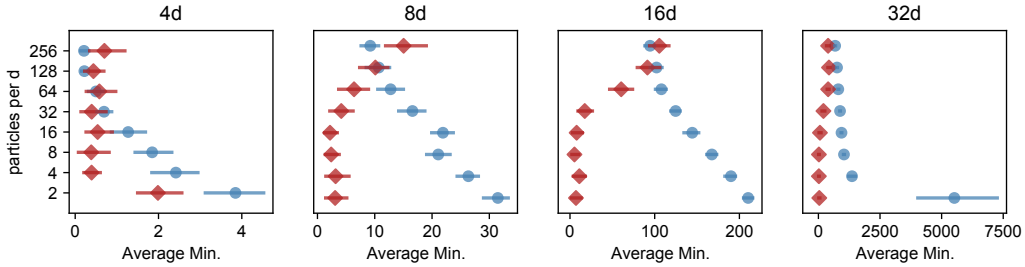


Figure 7.9: *Whitley Benchmark*



one could imagine a 128d problem where PI-PSO is better than PSO in the ways we have been describing.

The results for the Zakharov function might be another case similar to that of Salomon (Figure 7.8). The performance of both algorithms starts out nearly the same at low dimensions, but as the dimensionality increases, the increasing particles appears to help PSO more than PI-PSO to the point where PI-PSO almost appears stuck.

Whitley appears to occupy a strange position with respect to the hypotheses (Figure 7.9). For all dimensions, as the number of particles per dimension increases, the performance of PI-PSO diminishes (which does not support Hypothesis I). And yet the performance at low PPD is almost always better than PSO, which supports Hypothesis II. Yet it does appear that the trend, as the dimensionality of the problem increases, is for PI-PSO to perform better overall.

7.2.4 Discussion

Looking across all the problems and dimensions, there is no single parameterization that is always better than another. With 19 benchmarks and four possible dimensions, there are 76 sets of results. PI-PSO has the same results or better than PSO in 52 of them (68.4%) using 2 particles per dimension. However, this does not represent the best performance achievable by PI-PSO. In most cases, increasing particles per dimension, increases performance, for both PI-PSO and PSO. For PI-PSO, the problem lies in the cases where they do not.

In those cases, the results appear to suggest something like premature convergence is happening. This could be due to similarities in the properties of the benchmark functions studied. Further research is needed to determine if there are *classes* of benchmark functions that suffer from this performance degradation. PI-PSO is greedier than PSO for two reasons. While both algorithms only accept a new *gbest*

if it is better than the last one, PI-PSO eliminates hitchhiking by only permitting Pareto improving **gbest**s. It may be possible that "Two Steps Forward, One Step Back" [8] prevents PSO from climbing the wrong hill. While not exactly like Simulated Annealing, which allows successor states to be inferior to predecessor states, there may be occasional benefits from locally inferior changes in the *gbest*.

7.3 Conclusions

We set out to explore the comparative performance and scalability of the Pareto Improving Particle Swarm Optimization (PI-PSO) algorithm against the standard *gbest* PSO. Across a wide variety of standard benchmark optimization functions, at different dimensions and different particles per dimension, PI-PSO out-performed PSO. There were, however, a few notable exceptions. The success of PI-PSO suggests that the information exchange and conflict resolution mechanism, which works on a variable by variable basis using Pareto efficiency, may also utilize more information in the swarm than the standard algorithm. The exceptions, however, indicate that sometimes this mechanism might be overly greedy.

So far, we have only seen PI-PSO applied to the standard, continuous benchmark problems. In future research, we would like to see the algorithm applied to different kinds of problems such as discrete and combinatorial optimization problems (such

as the NK landscape or Bayesian abductive inference problems discussed in Chapter 3. The exploration of PI-PSO versus PSO performance across other accepted benchmark functions may provide clues to the function attributes that affect behavior with respect to dimensionality and the number of particles per dimension. It might also be interesting to explore the possibility of stochastically chosen *inferior* adjustments to avoid the potential for premature convergence.

Chapter 8

Actor Based (D)FEA

Evolutionary algorithms of all stripes can be computationally intensive and expensive. This computational cost can come from either the actual evolutionary algorithm or fitness/objective function evaluations. However, because these operations are all CPU-bound, they are not likely to get much help from mere concurrency. And with the apparent demise of Moore's Law [61, 62], we find ourselves in the same position as everyone else in software engineering: how do we take advantage of more cores (either on a single machine or across multiple machines)? One solution to this problem is distributed parallelism, but there is more than one way to implement distributed parallelism.

In this chapter we describe an Actor model implementation of FEA (Chapter 2) and DFEA (Chapter 3). As multi-population models, both FEA and DFEA are prime

candidates for a parallel implementation. Additionally, the computational needs of optimization suggest that they could benefit from a distributed implementation as well, whether across cores or across machines. However, both algorithms have a basic structure that involves iterative divide-and-conquer, so we must address the more general problem of concurrency as well. Although there are many programming models for concurrency and parallelism such as Software Transactional Memory (STM) [63] and Communicating Sequential Processes (CSP) [64], we have chosen the Actor model for its ability to execute distributed algorithms transparently. We begin with a discussion of the Actor model.

8.1 Actor Model

The Actor model was originally proposed by Hewitt *et al.* as a modular computational architecture for artificial intelligence [12]. The architecture was developed further by Agha into a metalinguistic model of the *potentially* concurrent process execution [65]. Later, Ericsson built the the Actor model into the OTP (originally "Open Telecom Platform"), part of the runtime of the Erlang programming language [66]. Using Erlang, OTP, and the Actor model, the AXD301 project was able to achieve "nine nine's" (i.e., 99.9999999%) reliability [67]. As a metalinguistic construct, there are Actor libraries available for many programming languages such as Akka [68] (for

JVM languages such as Java [69] and Scala [70]) and Thespian [71] (for Python [72]). In the following, we will focus mostly on the Akka/Thespian-style implementations of the "Classic Actor" model [73].

The key properties of an actor are:

1. they may only communicate via asynchronous messages;
2. messages may be received at anytime and are queued in a mailbox (queue); and
3. upon reading a message, the actor may perform a computation.

A variety of inferences can be drawn from these properties. First, there is no way to access the state of an actor without sending it an asychronous message. This differs from the Object model as we have generally come to know it, where "messages" are synchronous method calls. It has been reported apocryphally that Alan Kay—who coined "object oriented"— stated the Actor model is the closest to what he originally meant by the term. While the statement cannot be verified, Kay has repeated that message passing was the key idea of object oriented programming, and not inheritance or types [74]:

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.

which, although not an explicit endorsement of the Actor model, is a fairly good description of it.

Second, although actors are not operating system or green threads, they do represent a mechanism of concurrent programming. The actual details are handled by the *Actor System*, which is responsible for spawning actors, maintaining their addresses, monitoring mailboxes and delivering messages, restarting failed actors, and making sure every actor gets a chance to execute. The last point is handled by a thread pooling mechanism and load balancing. The default implementation is something like a "round robin" approach where every actor has a chance to act on a single message in its mailbox. However, actors are still subject to deadlocks if the state machines and messages are not designed properly.

Third, an actor can be run anywhere. An actor has local state and an interface defined by the messages it understands. When an actor sends a message to another actor, it sends it to the actor's *address* maintained by the Actor System. The receiving actor may be on the same core, a different core, or even a different machine.

In object oriented languages like Java and Python, actors are generally implemented as a subclass of some Actor base class. The instance fields of the Actor become its state and the subclass overrides something like a *receive* method with formal parameters *message* and *sender*. When it is the actor's turn to execute, the Actor *System* will call the actor's *receive* method if there is a message in the actor's mailbox. The actor may also implement instance methods for code organization but clients may *only* interact with an actor instance via messages through the actor's

Algorithm 8.18 Actor A - receive

Input: message *message*, sender *sender*
Output: None

```
1: if message instanceOf IncrementCount then
2:     count ← count + message.increment
3: else if message instanceOf RetrieveCount then
4:     tell(sender, CurrentCount(count))
5: end if
```

address (although we use the shorthand "sends a message to the actor").

Algorithm 8.18 shows a simple example of such a *receive* method. Actor A accepts two messages: *IncrementCount* and *RetrieveCount*. If an instance of Actor A receives an *IncrementCount* message (Line 1), the *count* is incremented by the value indicated (Line 2). If, instead, the instance of Actor A receives a *RetrieveCount* message (Line 3), a new message *CurrentCount* is sent back to the sender containing the current count, *count* (Line 4). This is a common pattern for implementing an Actor's *receive* method and in many respects acts like a finite state machine. For example, an actor may receive a message and update its state, and then optionally send a message as a result (or not).

Algorithm 8.19 shows a simple driver for Actor A. To start, we instantiate the Actor System (Line 1). In Line 2, we instantiate an instance of Actor A–although the variable *actor* is just the actor address. The best design principles for actor-based programs only permit actors to *send* messages to each other asynchronously; some libraries enforce this. An interesting result of this is that such programs become

Algorithm 8.19 Main

```
1: system ← ActorSystem()
2: actor ← system.actorOf(ActorA)
3: system.tell(actor, IncrementCount(10))
4: system.tell(actor, IncrementCount(5))
5: result ← system.ask(actor, RetrieveCount())          ▷ count is 15
6: system.stop()
```

reactive programs [75]. Such programs are event-driven, and nothing happens until something, somewhere sends a message to an actor.

In order to bridge the divide between the synchronous process *main* and the asynchronous *actor*, the Actor System provides a way for non-actor processes (like *main*) to send messages to actors. The *tell* method delivers the message asynchronously to the actor and is non-blocking. It works just like *send* between actors. The *ask* method delivers the message synchronously and is blocking. We require a blocking call—or something like it—because the *main* thread could finish before Actor A completed its computation and returned a message. For our example, we send two asynchronous messages to increment the count (Lines 3 and 4) and a single blocking call to retrieve the count (Line 5).

8.2 (D)FEA Actor Implementation

In this section, we discuss the translation of each algorithm from a serial version to an Actor model version. We will start with FEA and give an overview of the main

components of the algorithm. Following that discussion, we will explain a corresponding Actor model implementation. We will then do the same thing for DFEA. We will then walk through a sequence diagram for the DFEA actor implementation.

8.2.1 Translating FEA into FEA Actor

For reference, the pseudocode for FEA from Chapter 2, Algorithm 2.6 has been reproduced here as Algorithm 8.20. As we analyze the algorithm in order convert it to the Actor model, we can identify four main sections of code:

1. Initialization Step
   (a) Overall initialization (Lines 1, 3–4)
   (b) Swarm initialization (Line 2)
2. Update Step (Lines 6–10)
3. Compete Step (Line 11, also Algorithm 8.21)
4. Share Step (Line 12, also Algorithm 8.22)

A common pattern in Actor model implementations is a manager/workers pattern where a job is divided into units of work, and each unit is given to a worker to complete. The results are then aggregated back together and returned to the original client. Here our "units" are factors and the subpopulations assigned to them. We can

Algorithm 8.20 Factored Evolutionary Algorithms

Input: Function f, Evolutionary Algorithm ea
Output: Context $\mathbf{c}$ as candidate solution $\mathbf{x}$

1: $\boldsymbol{\mathcal{X}} \leftarrow factorize(\mathbf{X})$
2: $\mathbf{S} \leftarrow ea.initialize(f, \boldsymbol{\mathcal{X}})$
3: $\mathbf{c} \leftarrow$ *initialize-context*$(\mathbf{S})$
4: $\boldsymbol{\mathcal{O}} \leftarrow$ *identify-optimizers*$(\boldsymbol{\mathcal{X}})$
5: **repeat**
6: **repeat**
7: **for** S in $\mathbf{S}$ **do**
8: $S \leftarrow ea.update(S)$
9: **end for**
10: **until** stopping criteria
11: $\mathbf{c} \leftarrow compete(f, \mathbf{S}, \boldsymbol{\mathcal{O}}, \mathbf{c})$
12: $share(f, \mathbf{S}, ea, \mathbf{c})$
13: **until** stopping criteria
14: **return** $\mathbf{c}$

Algorithm 8.21 FEA Compete

Input: Objective function f, Subpopulations $\boldsymbol{\mathcal{S}}$, Optimizers $\boldsymbol{\mathcal{O}}$, Global context $\mathbf{c}$
Output: Global context $\mathbf{c}$

1: **for** $j = 1$ to d **do**
2: $fitness \leftarrow f(\mathbf{c})$
3: $value \leftarrow \mathbf{c}[j]$
4: **for** i in $\boldsymbol{\mathcal{O}}_j$ **do**
5: $candidate \leftarrow \mathcal{S}[i].best$
6: $\mathbf{c}[i] \leftarrow candidate.\mathbf{x}[i]$
7: **if** $f(\mathbf{c}) \leq fitness$ **then**
8: $value \leftarrow candidate.\mathbf{x}[i]$
9: $fitness \leftarrow f(\mathbf{c})$
10: **end if**
11: **end for**
12: $\mathbf{c}[i] \leftarrow value$
13: **end for**
14: **return** $\mathbf{c}$

Algorithm 8.22 FEA Share

Input: Objective function f, Subpopulations $\mathbf{S}$, Evolutionary Algorithm ea, Context $\mathbf{c}$
Output: Subpopulations $\mathbf{S}$

1: **for** S in $\mathbf{S}$ **do**
2: $\quad \mathbf{r} \leftarrow \mathbf{c} \setminus S.X$
3: $\quad f_r \leftarrow partial(f, \mathbf{r})$
4: $\quad p \leftarrow ae.worse(S)$
5: $\quad p.\mathbf{x} \leftarrow \mathbf{c} \setminus \mathbf{r}$
6: $\quad S.f \leftarrow f_r$
7: $\quad ae.reevaluate(S)$
8: **end for**

thus plan to create one actor for each factor/subpopulation. The worker will initialize the swarm and then complete the Update step. The manager will perform all the other steps including Overall initialization, Compete Step, and at least part of the Share Step. We will break the algorithm into two actors: the FEA Actor (Algorithm 8.23, and the FEA Factor Actor (Algorithm 8.24). The main challenge here is that the "disperse and collect" flow that normally accompanies the manager/workers pattern is repeated until some stopping criterion is met. We will thus require some way to coordinate workers before each Compete and Share Step starts.

The FEA actor responds to two messages: *InitFEA* (Line 1) and *NewValue* (Line 14). Additionally, it will send *InitFactor* (Line 10), *Update* (Lines 12 and 21), *NewSolution* (Line 20), and *CandidateSolution* (Line 23) messages. Whereas an object oriented solution might have a synchronous method call *solve* as an interface, the FEA actor's interface is the asynchronous messages *InitFEA* and *CandidateSolution*.

Algorithm 8.23 FEA Actor - receive

Input: message *message*, sender *sender*
Output: None

```
1: if message instanceOf InitFEA then
2:     client ← sender
3:     problem ← message.problem
4:     𝒳 ← factorize(X)
5:     S ← ea.initialize(f, 𝒳)
6:     c ← initialize-context(S)
7:     𝒪 ← identify-optimizers(𝒳)
8:     for X in 𝒳 do
9:         worker ← actorOf(FEAFactor())
10:        worker.send(InitFactor(problem, X))
11:        workers[X] ← worker
12:        worker.send(Update())
13:    end for
14: else if message instanceOf NewValue then
15:    cache[message.x_i] ← message.value
16:    if new values received from all actors then
17:        if FEA iterations not complete then
18:            compete()
19:            clearCache()
20:            broadcast(workers, NewSolution())
21:            broadcast(workers, Update())
22:        else
23:            client.send(CandidateSolution(c))
24:        end if
25:    end if
26: end if
```

Upon receipt of the *InitFEA* message, the FEA actor proceeds almost identically to the first part of Algorithm 8.20. The main difference is in Lines 2–3 and Lines 8–13. In Line 2 we save the client who sent the FEA actor the *InitFEA* message so that we can respond later. We also save the *problem* record, which encapsulates

Algorithm 8.24 FEA Factor Actor - receive

Input: message *message*, sender *sender*
Output: None

```
1: if message instanceOf InitFactor then
2:     problem ← message.problem
3:     S ← ae.initialize(f, X)
4: else if message instanceOf Update then
5:     for i times do
6:         S ← ae.update(S)
7:     end for
8:     sender.send(NewValue(X, S.best)
9: else if message instanceOf NewSolution then
10:    applySolution(S, message.c)
11: end if
```

information both about the problem and parameters for the algorithm. In Lines 8–13 we create the workers, FEA Factor actors, sending them both an *InitFactor* message and a *Update* message, after saving the reference to each worker's address.

Before continuing with the FEA actor and the $NewValue$ message, we describe the FEA Factor actor (Algorithm 8.24). The FEA Factor actor responds to three messages: *InitFactor* (Line 1), *Update* (Line 4), and *NewSolution* (Line 9). When the FEA Factor actor receives the *InitFactor* message it saves the *problem* from the message and then initializes the subpopulation based on the particular evolutionary algorithm, optimization problem, and factor. When it receives the *Update* message, the actor updates the subpopulation for i iterations. This is exactly the same as the corresponding lines in Algorithm 8.20. In order to make the loop in Algorithm 8.20 run concurrently, we have turned the iteration loop (Algorithm 8.20, Line 6) into a

Algorithm 8.25 Broadcast Helper Function

Input: actors **actors**, message *message*

```
1: for actor in actors do
2:     actor.send(message)
3: end for
```

message sending loop (Algorithm 8.23, Line 12 and Line 21). The inner loop from Algorithm 8.20 then runs on the individual actors. When the FEA Factor actor's part in this distributed Update Step is done, it sends a *NewValue* message back to the FEA actor.

Returning to Algorithm 8.23, the FEA actor responds to the *NewValue* message by caching the value (Line 15). It then tests to see if it has received all the expected new values. This *cache-and-test* pattern is one way to coordinate all the workers and the algorithm implements the bookkeeping required to implement the pattern. If all the expected new values have been received—and the desired number of FEA iterations have been completed—the Compete Step is executed. This Compete Step is otherwise identical to Algorithm 8.21 except that, instead of extracting the values from the subpopulations, the values have already been extracted and saved to the cache. After this new Compete Step is finished, the cache is cleared for the next round, and a *NewSolution* message is sent to all the workers.

In the context of this paper, *broadcast* is just a helper function (Algorithm 8.25) that loops over the actor references, sending each the same message. It is not a "fire-

and-forget" broadcast or any other type of pub/sub message passing. Actors always send messages to specific actors. This is followed by an *Update* message to everyone. If the FEA iterations (or, more generally, stopping criteria) have completed, then the FEA actor sends a *CandidateSolution* message to the original client.

8.2.2 Translating DFEA into DFEA Actor

In this section we discuss the conversion of DFEA (Algorithms 3.9 and 5.16, reproduced here as Algorithms 8.26 and 8.27) to the Actor model. Looking at Algorithm 8.26, it would appear that DFEA and FEA have the same structure and we could use the same actors as above. There is one crucial difference between FEA and DFEA, however. DFEA was designed with distributed state in mind with each subpopulation having its own local context, $\mathbf{c}_i$, instead of the centralized one in FEA. This means that the Update and Share Steps should belong to the workers instead of the supervisor. This makes the DFEA Actor implementation more of a peer pattern than supervisor/worker pattern although we will need a supervisor for the overall initialization. Changing the top-down, outside-in pseudocode to a decentralized but coordinated peer pattern will require more work than we had to do above.

The DFEA actor's "receive" implementation is shown in Algorithm 8.28. It is simpler than the corresponding FEA actor because the DFEA actor simply spawns DFEA Factor actors in response to a *InitDFEA* message (Line 1) and sends a *Can-*

Algorithm 8.26 Distributed Factored Evolutionary Algorithm

Input: Function f, Evolutionary Algorithm *ae*
Output: Best context $\mathbf{c}$ as candidate solution $\mathbf{x}$

1: $\boldsymbol{\mathcal{X}} \leftarrow factorize(\mathbf{X})$
2: $\mathbf{S} \leftarrow ae.initialize(f, \boldsymbol{\mathcal{X}})$
3: $\boldsymbol{\mathcal{C}} \leftarrow$ *initialize-contexts*$(\mathbf{S})$
4: $\boldsymbol{\mathcal{O}} \leftarrow$ *identify-optimizers*$(\boldsymbol{\mathcal{X}})$
5: $\boldsymbol{\mathcal{A}} \leftarrow$ *identify-arbiters*$(\boldsymbol{\mathcal{X}})$
6: **repeat**
7: **repeat**
8: **for** S in $\mathbf{S}$ **do**
9: $S \leftarrow ae.update(S)$
10: **end for**
11: **until** stopping criteria
12: $\boldsymbol{\mathcal{C}} \leftarrow compete(f, \mathbf{S}, \boldsymbol{\mathcal{O}}, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{C}})$
13: $share(f, \mathbf{S}, A, \mathbf{C})$
14: **until** stopping criteria
15: $\mathbf{c} \leftarrow$ *select-best-context*$(f, \boldsymbol{\mathcal{C}})$
16: **return** $\mathbf{c}$

didateSolution message in response to a *CandidateSolution* message from a worker.

The complexity of the DFEA Factor actor follows directly from the nature of the algorithm, which distributes a local context to each individual subpopulation. However, the algorithm does not specify a concurrent means of manipulating and coordinating those local contexts. This is to be expected since there is no single way to specify pseudocode appropriate for all possible concurrency implementations, and picking one could make the translation to another equally complicated. In this case, we at least have a clear idea of the intended semantics.

Nevertheless, the DFEA Factor actor requires nine messages: *InitFactor* (Algo-

Algorithm 8.27 DFEA Reconcile

Input: Function f, Subpopulations $\mathbf{S}$, optimizers $\boldsymbol{\mathcal{O}}$, arbiters $\boldsymbol{\mathcal{A}}$, Local contexts $\boldsymbol{\mathcal{C}}$
Output: Local contexts $\boldsymbol{\mathcal{C}}$

```
1: for j = 1 to d do
2:     c ← C[A(x_j)]
3:     fitness ← f(c)
4:     value ← c[j]
5:     for k in O_j do
6:         candidate ← S[k].best
7:         c[j] ← candidate.x[j]
8:         if f(c) ≤ fitness then
9:             value ← candidate.x[j]
10:            fitness ← f(c)
11:        end if
12:    end for
13:    c[j] ← value
14:    for k = 1 to d do
15:        C[k].c[j] ← c[j]
16:    end for
17: end for
18: return C
```

rithm 8.29), *ArbiterOf* (Algorithm 8.30), *Update* (Algorithm 8.31), *NewValue* (Algorithm 8.32), *ReadyToArbitrate* (Algorithm 8.33), *StartArbitration* (Algorithm 8.34), *ArbitedValue* (Algorithm 8.35), and *ArbitrationComplete* (Algorithm 8.36). These are all shown individually as "message handler" code fragments intended to be part of a larger *receive* method similar to the one shown for the FEA Factor actor (Algorithm 8.24). Additionally, as the ideas and patterns are similar to those we have seen before, we will refer mostly to algorithms rather than line-by-line descriptions unless a specific detail requires attention.

Algorithm 8.28 DFEA Actor - receive

Input: message *message*, sender *sender*
Output: None

```
1: if message instanceOf InitDFEA then
2:     client ← sender
3:     problem ← message.problem
4:     𝒳 ← factorize(X)
5:     S ← ae.initialize(f, 𝒳)
6:     c ← initialize-context()
7:     𝒪 ← dfeaa:identify-optimizers(𝒳)
8:     𝒜 ← dfeaa:identify-arbiters(𝒳)
9:     for X in 𝒳 do
10:        workers[X] ← actorOf(DFEAFactor())
11:    end for
12:    broadcast(workers, InitFactor(problem, c, X, 𝒜))
13:    broadcast(workers, Update())
14: else if message instanceOf CandidateSolution then
15:    if all candidate solutions received then
16:        c ← select-best-context(f, 𝒞)
17:        client.send(CandidateSolution(c))
18:    end if
19: end if
```

Algorithm 8.29 *InitFactor* Message Handler

```
1: S ← ae.initialize(message.problem)
2: broadcast(coworkers, ArbiterOf(𝒜[X]))
```

As we saw in the DFEA actor, the worker actors—DFEA Factors—are all spawned and initialized with an *InitFactor* message (Algorithm 8.29). In response to a similar message, FEA Factors immediately send themselves an *Update* message after initialization. Instead, DFEA Factors send an *ArbiterOf* message to all the other workers. In the original algorithm, all the information needed to match optimizers and arbi-

Algorithm 8.30 *ArbiterOf* Message Handler

```
1: arbiters[message.X] ← sender
2: if all arbiters accounted for then
3:     self.send(Update())
4: end if
```

Algorithm 8.31 *Update* Message Handler

```
1: for i times do
2:     S ← ae.update(S)
3: end for
4: for X in X do
5:     sender.send(arbiters[X], NewValue(X, S.best[X]))
6: end for
```

trators is available to the algorithm itself, which still acts as a central coordinator of the local contexts. Although the DFEA actor still initializes factors, optimizers, and arbitrators, we opted to implement arbiter discovery through peers. Using the cache-and-test pattern, the DFEA Factor actor will send an *Update* message to itself when it knows who all of the arbiters are (Algorithm 8.30).

The *Update* message will start the Update Step just as with FEA/DFEA/FEA Actor (Algorithm 8.31). When the Update Step is complete, the actor will then send a *NewValue* message to the arbiter of each variable in its factor. Thus, if the actor is optimizing (X_1, X_2, X_3), it will send a *NewValue* message to the arbiter of X_1, one to the arbiter of X_2, and one to the arbiter of X_3. Again, from an implementation perspective, the interesting detail here is that the current actor may actually be the arbiter of one, all, or none of those variables.

As actors collect new values, they check to see if they have heard from all of

Algorithm 8.32 *NewValue* Message Handler

```
1: if message.x ∈ X then
2:     newValueCache[message.x] ← message.value
3:     if all values received from optimizers of x then
4:         broadcast(coworkers, ReadyToArbitrate(message.x))
5:     end if
6: end if
```

Algorithm 8.33 *ReadyToArbitrate* Message Handler

```
1: arbitrationCache[message.X] ← True
2: if all actors ready to arbitrate and self is first in arbitration order then
3:     send(self, StartArbitration(message.X))
4: end if
```

their optimizers (Algorithm 8.32). This is yet another instance of the cache-and-test pattern. When all actors have heard from all of their optimizers, the Update phase is over, and the Arbitration phase begins.

Using the cache-and-test pattern again, when all the actors have heard that all the other actors are ready to arbitrate, the actor whose factor contains the first variable in the arbitration order sends itself a *StartArbitration* message (Algorithm 8.33). Upon receiving a *StartArbitration* message, the DFEA Factor actor conducts a simplified version of *FEA*'s Compete Step using the cached values from its optimizers. It then broadcasts an *ArbitedValue* message to all co-workers (Algorithm 8.34). If the current actor contains the last variable in the arbitration order, it sends an *ArbitrationComplete* message to all its peers; however, if it is not the last variable, it sends a *StartArbitration* message to the actor who is next in variable arbitration order.

The easiest message to handle is the *ArbitedValue* message, in which case the

Algorithm 8.34 *StartArbitration* Message Handler

1: $\mathbf{c} \leftarrow compete(message.X, newValuesCache[message.X])$
2: $broadcast(\mathbf{coworkers}, ArbitedValue(message.X, \mathbf{c}[message.X])$
3: **if** this is the last arbiter **then**
4: $\quad broadcast(\mathbf{coworkers}, ArbitrationComplete())$
5: **else**
6: $\quad next \leftarrow arbitrationOrder(message.X)$
7: $\quad \mathbf{arbiters}[next].send(StartArbitration(next))$
8: **end if**

Algorithm 8.35 *ArbitedValue* Message Handler

1: $\mathbf{c}[message.X] \leftarrow message.value$

Algorithm 8.36 *ArbitrationComplete* Message Handler

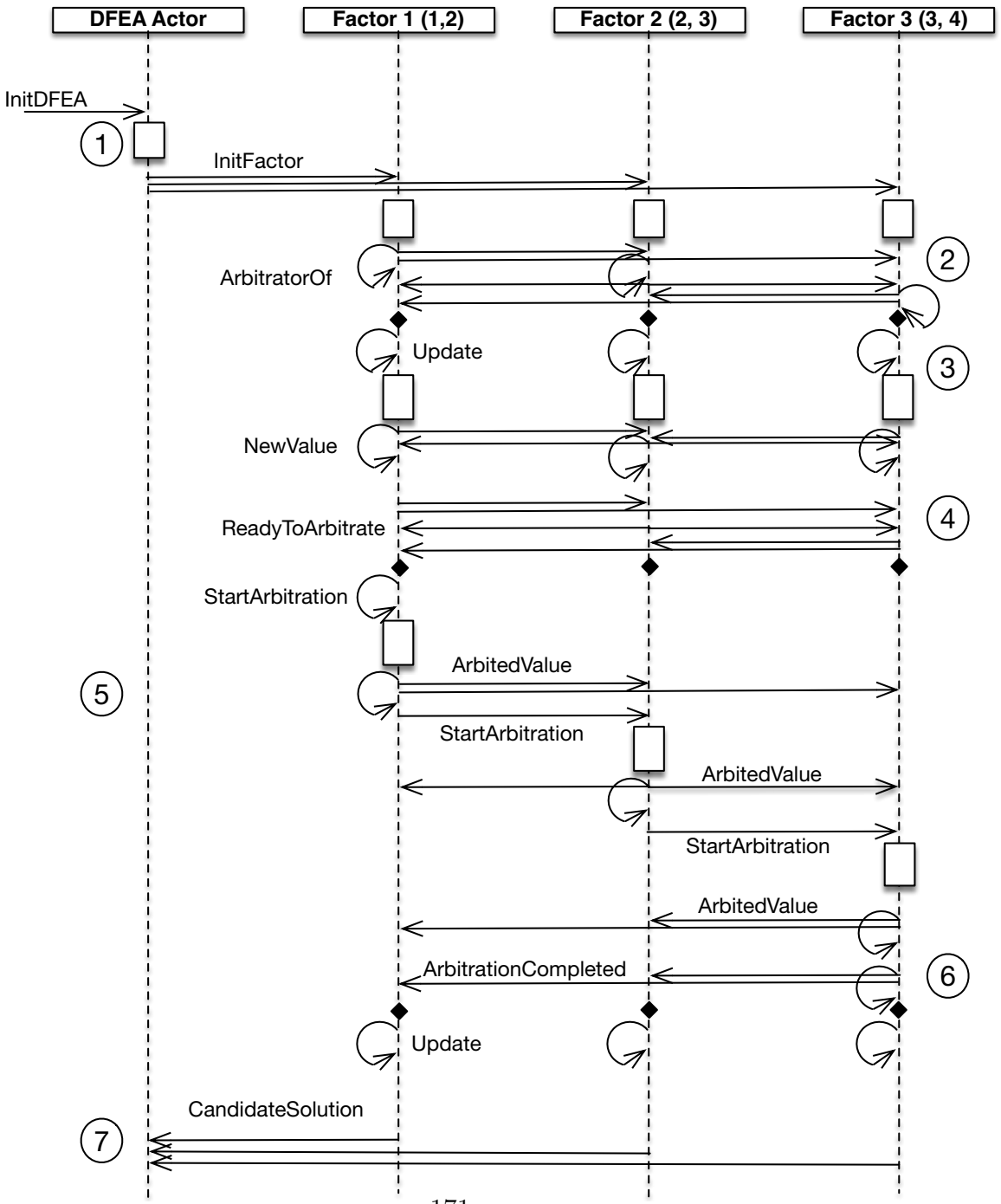
1: **if** all FEA iterations completed **then**
2: $\quad client.send(CandidateSolution(\mathbf{X}, \mathbf{c})$
3: **else**
4: $\quad clearCaches()$
5: $\quad broadcast(\mathbf{coworkers}, Update())$
6: **end if**

actor sets the corresponding value of its local context, **c** (Algorithm 8.35). As noted previously, the last actor to arbitrate sends an *ArbitrationComplete* message. All actors handle the message by checking to see if the FEA stopping criteria are met. If they are, then the client (DFEA Actor) is sent the current context as a candidate solution. Otherwise, the caches are cleared and the Arbitration phase is over and a new Update phase begins (Algorithm 8.36).

8.2.3 DFEA Sequence Diagram

In this section we will work through a concrete example of the DFEA actor and DFEA Factor actors using a sequence diagram. Our particular implementation uses eleven messages: *InitDFEA*, *InitFactor*, *ArbiterOf*, *Update*, *NewValue*, *ReadyToArbitrate*, *StartArbitration*, *ArbitedValue*, *ArbitrationComplete*, and *CandidateSolution*. Assuming a problem of $4d$, there will be four variables: X_1, X_2, X_3, and X_4. Factor 1 is optimizing (X_1, X_2) and arbitrating X_1. Factor 2 is optimizing (X_2, X_3) and arbitrating X_2. Factor 3 is optimizing (X_3, X_4) and arbitrating X_3 and X_4. As a result there will be four actors: one DFEA actor that spawns three DFEA Factor actors. The sequence diagram for this actor system is shown in Figure 8.1.

Figure 8.1: *Sequence Diagram for DFEA and DFEA Factor Actors*



We start in an Initialization phase. At (1) in Figure 8.1, an asynchronous *InitDFEA* message is sent to the DFEA actor's mailbox (All the messages are sent asynchronously in this example; denoted by open arrowheads). The message contains information about the problem being solved and the parameters for DFEA itself such as stopping criteria. Just as with the serial version (Algorithm 8.26, Lines 1, Lines4–5), the Actor model version begins by initializing factors, optimizers, and assigning arbiters to each factor. This is followed by creating a Factor actor for each factor, in this case: Factor 1, Factor 2 and Factor 3. After these actors are created, they each receive an *InitFactor* message on their individual mailboxes.

The *InitFactor* message signals each actor to take the information on the problem, factors, optimizers and arbitrators contained in the message and initialize their subpopulation using the indicated evolutionary algorithm. This corresponds to Algorithm 8.26, Line 2 in the serial version of the algorithm. After each Factor actor has initialized, (2) they send an *ArbiterOf* message to all their peers indicating the variables for which they are the arbiter. There is a corresponding coordination point (black diamond) where a Factor actor must wait until it has heard from all the arbiters for the variables it is optimizing. Using Factor 1 as an example, after it has been initialized, it sends an *ArbiterOf* message to all its peers indicating that it is the arbiter for X_1. Because Factor 1 is optimizing X_1 and X_2, it waits to hear from the Factors optimizing X_1 and X_2 before proceeding. This is a pattern that we saw

before in the FEA Actor, where a cache is used to coordinate multiple actors and then tested to change to the next state. We indicate this *cache-and-test* pattern with the black diamond in the diagram. The difference in this case is that when Factor 1 sent the *ArbiterOf* messages, it sent one to itself. We do this to avoid special case code in the actors which would require the actor to know who it is; as we will see later, we were not entirely successful.

After a Factor has discovered which actors will arbitrate each of the variables it optimizes, (3), it sends itself an *Update* message, which starts the Update phase. At this point, the Factor will run its evolutionary algorithm on its subpopulation until the stopping criteria are met. This corresponds to Algorithm 8.26, Lines 7–11. Upon completion, the factor will send a *NewValue* message for each of the variables it optimizes to the arbiter of that variable. In the case of Factor 1, it will send a *NewValue* message for X_1 and one for X_2. Again we see the pattern of avoiding special code: Factor 1 sends the message about X_1 to itself. It does not send a new value message to itself for X_2 because it is not an arbiter of X_2.

With (4) we enter another coordination point that uses the cache-and-test pattern. A Factor is ready to arbitrate X_1 if it has heard from all the optimizers of X_1. When it has, it sends a *ReadyToArbitrate* message to all of the actors. Factor 3, because it is the arbiter of X_3 and X_4 will await messages for both variables and send a message for each variable.

Once all of the Factor actors have received all of the *ReadyToArbitrate* messages, the Arbitration phase begins, (5). We have assumed that the variables will be arbitrated in order, X_1, X_2, X_3 then X_4, for simplicity. As a result, when Factor 1 knows that all the factors are ready to arbitrate, it sends itself a *StartArbitration* message. This message begins the reconciliation process described in Algorithm **??** with a few key differences. First, instead of iterating from the "outside" over all the variables, in the Actor model implementation, the actor is working from the "inside" with a specific variable to arbitrate. Second, the Factor does not have global access to the subpopulation information so it cannot reach into the subpopulations of optimizers of X_i and find x_i. Instead, these were the values communicated via *NewValue* messages before arbitration commenced.

When the current factor (Factor 1) is finished, it sends *ArbitedValue* messages to all of its peers. This will enable factors further down the arbitration order to use those values during reconciliation just as with the serial version. After sending those messages, it sends a single *StartArbitration* message to the next factor in the arbitration order. We can easily calculate this from the current information in Factor 1. If it arbitrates X_1 then the next arbiter must be X_2. If a different or changing arbitration order were desired, this would need to be communicated and coordinated as well.

When the final factor finishes arbitration, (6), it sends an *ArbitrationCompleted*

message to all of its peers. Factor 3 is able to discover that it is the final factor when it finishes arbitrating X_4 and discovers there is no "next" variable to arbitrate. At this point, we have another cache-and-test pattern as all the factors await the message indicating that the Arbitration phase is over. If the stopping criteria for the DFEA are not met, all the factors will send themselves *Update* messages and the Update phase begins anew (this is the case shown in the diagram). If the stopping criteria have been met, each Factor sends a *CandidateSolution* message to the DFEA actor, (7). Because each individual actor knows what the stopping criteria are, this message does not have to be coordinated on the Factor side.

We have not shown how the DFEA actor handles the *CandidateSolution* messages. For testing, we used the cache-and-test pattern to await all of the messages. When they were all received, the best solution was chosen and sent back to the driver program. However, there is no reason that the DFEA actor cannot maintain a single best solution and revise it as *CandidateSolution* messages come in. The DFEA actor could then be queried via a message at any time for what it thinks is the best solution so far.

8.3 Validating the Implementations

DFEA was meant to preserve the semantics of FEA in the presence of distributed state in the form of local contexts. Because of the similarities in the algorithms and the fact that neither the Compete/Reconcile or Share Steps consume random numbers, the output of each algorithm when initialized with the same random seed is identical. We can thus see that DFEA preserved the semantics of FEA.

The above Actor model implementations of both FEA and DFEA are meant to preserve the semantics of the original algorithms. However, because these are distributed algorithms running on multiple threads, we are unable to verify the implementations against the baseline the same way that we did for DFEA. Instead we have turned to experimental means.

8.3.1 Design

In order to test the hypothesis that the Actor implementations preserved the semantics of the original algorithm, we executed FEA (baseline), FEA Actor, and DFEA Actor implementations against 19 benchmark optimization functions. We picked benchmark optimization functions that were scalable to multiple dimensions from [18] and [8]. These are all minimization problems, and most of them have a minimum at $\mathbf{x}^* = [0, 0, ..., 0]$ and $f(\mathbf{x}^*) = 0$. The notable exceptions are the

Exponential, Eggholder and Michalewicz functions. For these experiments we used $32d$ versions of the functions.

For our evolutionary algorithm we chose PSO. The PSO parameters were $\omega = 0.729$ and $\phi_1 = \phi_2 = 1.49618$. In all cases, there were 20 FEA iterations with 5 update iterations per iteration. The factor architecture was the same for all algorithms and functions: we used a "Simple Centered" factor architecture of (X_i, X_{i+1}), thus, the first factor was (X_1, X_2), the second was (X_2, X_3), etc. Each subpopulation (swarm) had 10 particles.

8.3.2 Results

Each function was optimized by each algorithm 50 times, and the mean minimum value found was recorded for each run. The results were then bootstrapped 500 times to estimate 95% confidence intervals/credible intervals [59]. These mean minima and confidence intervals are shown in Table 8.1.

In every case, the FEA Actor implementation performed as well as the FEA baseline (serial) implementation. This was also true for the DFEA Actor implementation. Additionally, in almost every case except one (18 out of 19), the FEA Actor and DFEA Actor implementations performed equally as well. The odd function out was the Zakharov function where the DFEA Actor implementation appears to have performed slightly better than the FEA Actor implementation (last row of Table 8.1).

Table 8.1: *Results for FEA Baseline and FEA and DFEA Actor Implementations*

Function	FEA Baseline	Actor FEA	Actor DFEA
ackley-1	1.70e-07 (1.60e-09, 4.75e-07)	4.85e-07 (7.67e-10, 1.55e-06)	1.62e-04 (7.10e-09, 5.00e-04)
brown	1.45e-23 (2.49e-25, 4.23e-23)	4.05e-24 (3.50e-26, 9.68e-24)	1.53e-21 (1.04e-24, 5.66e-21)
dixon-price	2.75e+01 (2.01e+01, 3.52e+01)	3.20e+01 (2.35e+01, 4.05e+01)	3.22e+01 (2.25e+01, 4.17e+01)
eggholder	-2.13e+04 (-2.16e+04, -2.10e+04)	-2.14e+04 (-2.17e+04, -2.11e+04)	-2.13e+04 (-2.16e+04, -2.10e+04)
exponential	-1.00e+00 (-1.00e+00, -1.00e+00)	-1.00e+00 (-1.00e+00, -1.00e+00)	-1.00e+00 (-1.00e+00, -1.00e+00)
griewank	1.26e-01 (6.27e-02, 2.02e-01)	3.50e-02 (1.62e-02, 6.27e-02)	1.08e-01 (4.60e-02, 1.79e-01)
michalewicz	-3.07e+01 (-3.08e+01, -3.06e+01)	-3.08e+01 (-3.10e+01, -3.07e+01)	-3.08e+01 (-3.10e+01, -3.07e+01)
rastrigin	2.49e-01 (1.30e-01, 3.89e-01)	1.35e-01 (4.49e-02, 2.24e-01)	1.65e-01 (5.99e-02, 2.59e-01)
rosenbrock	1.26e+01 (7.00e+00, 1.95e+01)	2.20e+01 (1.19e+01, 3.26e+01)	1.95e+01 (1.12e+01, 2.76e+01)
salomon	1.83e+00 (1.71e+00, 1.96e+00)	1.73e+00 (1.62e+00, 1.87e+00)	1.83e+00 (1.69e+00, 1.97e+00)
sargan	3.43e+03 (1.71e+03, 5.28e+03)	1.71e+03 (6.06e+02, 3.03e+03)	2.09e+03 (9.22e+02, 3.69e+03)
schaffer-f6	8.35e-01 (7.37e-01, 9.42e-01)	8.57e-01 (7.57e-01, 9.46e-01)	8.16e-01 (7.25e-01, 9.21e-01)
schwefel-1.2	5.78e+04 (4.52e+04, 7.12e+04)	6.71e+04 (5.22e+04, 8.46e+04)	7.78e+04 (5.39e+04, 1.06e+05)
schwefel-2.22	3.29e-12 (7.00e-13, 7.67e-12)	9.14e-13 (3.12e-13, 1.72e-12)	1.84e-12 (1.05e-12, 2.87e-12)
schwefel-2.23	1.42e-94 (1.38e-99, 5.58e-94)	5.12e-98 (1.45e-109, 1.52e-97)	7.56e-96 (1.28e-100, 2.21e-95)
sphere	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
stretched-v	4.08e+00 (3.78e+00, 4.38e+00)	3.91e+00 (3.68e+00, 4.16e+00)	4.03e+00 (3.73e+00, 4.35e+00)
whitley	4.96e+02 (4.56e+02, 5.35e+02)	5.07e+02 (4.63e+02, 5.54e+02)	5.20e+02 (4.75e+02, 5.63e+02)
zakharov	8.09e+02 (7.82e+02, 8.34e+02)	8.21e+02 (7.92e+02, 8.51e+02)	7.56e+02 (7.29e+02, 7.83e+02)

Given the consistent performance of the algorithms, however, this is likely to have been a statistical fluke. We believe the evidence supports the hypothesis that the Actor implementations preserved the semantics of the baseline algorithms.

8.4 Discussion

Surprisingly, although DFEA was designed to represent distributed state, the translation of FEA into the Actor model was easier than DFEA's translation. This is largely because although the Actor model is effective for concurrency, it lacks primitives for coordinated, distributed state. It is thus easier for the FEA Actor to launch as many Factor Actors as needed and take care of the coordination required for the Compete and Share Steps than for the DFEA Actor to do the same. In the case of the DFEA Actor implementation, the DFEA Actor delegates any coordinating role to its Factor Actors who then coordinate among themselves. Another way to think of this is that DFEA Factor actors are cooperative peers whereas the FEA Factor actors are solitary workers.

When thinking about implementing a peer pattern using the Actor model, the implementation often becomes confusing because we have to think of the actor not only as the sender of the message but also the receiver of the message. There are a few places where this breaks down. For example, in order to start the Arbitration phase,

we have to test the current actor to see if they are supposed to go first. Similarly, when the Arbitration phase is over, we have to check to see if the current actor is the last arbiter to go and then it sends out a different message.

The reactive nature of the Actor model has interesting implications that are not fully utilized in these experiments. In some instances, we may have a very difficult optimization problem for which we would want a provisional answer and then updates to that answer. The Actor model, by virtue of its reactive nature, would support this use case directly. Either implementation could be reconfigured to run indefinitely rather than for some fixed number of FEA iterations. An API service could be launched in an Actor System that could talk to the Actor System this running (D)FEA Actor instance. The API service could then send a *RequestSolution* message to the (D)FEA Actor instance and wait for the reply returning it to the user. This API could be then be used to get up-to-date estimates by the client actually using the value as needed.

8.5 Conclusions

In this chapter we presented an Actor model implementation of Factored Evolutionary Algorithms and Distributed Factored Evolutionary Algorithms. The Actor implementation of FEA involved a fairly straight-forward translation of the serial

pseudocode to a parallel implementation. This involved a common pattern in Actor-based implementations where a supervisor breaks a task into pieces and then spins up a worker Actor for each piece. This pattern matched FEA exactly.

Although DFEA has the same general steps as FEA, the semantic intent is closer to that of peers rather than workers. This made the translation of the serial pseudocode into a parallel implementation a bit more challenging, even though the basics had been worked out. The Actor-based implementation involved using a peer pattern, which required us to think of each Actor as not only the sender of the message but the receiver of the message. In some cases, this required code to handle special cases as in the start of the Arbitration phase.

The evidence presented by our validation experiments strongly indicate that our implementations faithfully reproduce the semantic intent of the original algorithms. Using PSO as the evolutionary algorithm, we ran experiments for three implementations: FEA baseline, FEA Actor, and DFEA Actor. Using 19 benchmark optimization functions, we showed that both the FEA Actor and the DFEA Actor performed comparably to the FEA baseline. There was one strange case where the DFEA Actor implementation performed better than the FEA actor.

Chapter 9

Conclusions

The previous chapters have presented various results as we have applied a model of information exchange and conflict resolution to various single- and multi-population, biologically inspired algorithms. Although many of the results are perfectly general (DFEA, for example), we have concentrated on Particle Swarm Optimization variants (FEA-PSO, DFEA-PSO, Actor DFEA-PSO, PI-PSO). With respect to FEA and DFEA, the "-PSO" part does not affect the results but serves to focus and unify the research.

Traditionally, researchers have attributed the success of these algorithms over their single population counterparts to an increased degree of cooperation. This cooperation (multiple populations) versus competition (individuals within a population) way of looking the algorithms is very powerful, especially for algorithms rooted in

biological analogies.

In this dissertation, we have taken a different approach. After developing an initial version of DFEA and finding that the performance relative to FEA was not what we anticipated, we sought to understand better how these algorithms worked. We identified two main characteristics of these multi-population algorithms. First, they used a blackboard architecture through which the individual populations communicate. Second, the implicit merge operation for the blackboard involved a conflict resolution mechanism based on Pareto efficiency. We have applied these framework to analyze and develop various PSO variants.

9.1 Contributions

In this dissertation we make several significant contributions to the evolutionary computation and swarm intelligence in the field of computer science. These are:

- **Distributed Factored Evolutionary Algorithms (DFEA):** We developed a generalized version of Factored Evolutionary Algorithms (FEA) in the same way that Distributed Overlapping Swarm Intelligence (DOSI) extended Overlapping Swarm Intelligence (OSI) to the distributed case. Like FEA, DFEA can be used with any "evolutionary algorithm" (for example, Genetic Algorithm and Particle Swarm Optimization).

Our results showed that DFEA was competitive with FEA on a variety of problems including Bayesian Networks, NK Landscapes, and a selection of continuous function benchmark optimization problems. However, DFEA did not always perform identically to FEA which led us to consider how the information flow semantics were different between the two algorithms. This led us to consider a different way of looking at these kinds of algorithms. We also showed that it is possible to relax consensus in DFEA and still achieve relatively good performance, at least on problems with low epistasis.

- **Information Exchange and Conflict Resolution Framework:** FEA and DFEA are both the latest in a long line of multi-population algorithms that have emphasized the roles of cooperation and competition in biologically inspired algorithms. As an alternative we develop a framework based on information exchange via a blackboard architecture and conflict resolution using Pareto improvements. We then applied this framework to FEA and identified how information flow and conflict resolution work in that family of algorithms.

- **Revised DFEA:** FEA and DFEA (as well as OSI and DOSI) have always had inconsistent performance when, at least on the surface, it had seemed like the distributed versions should perform equally as well as the centralized versions. By applying the Information Exchange and Conflict Resolution Framework, we

are able to identify and improve the information flows in DFEA so that the performance of the two algorithms, DFEA and FEA matches.

Our results show that the revised DFEA performs identically to FEA when both are initialized identically. Even more interesting, our results showed that the original DFEA sometimes out-performed both FEA and DFEA.

- **Pareto Improving Particle Swarm Optimization:** We apply our Information Exchange and Conflict Resolution framework to the *gbest* Particle Swarm Optimization algorithm. By making the *gbest* a full blackboard architecture and extending variable by variable conflict resolution to particles, we create a single population algorithm that performs on a par with FEA. We also examine the comparative performance and scaling characteristics of the this PI-PSO as compared to PSO.

  Our results showed that PI-PSO performed better than PSO on our continuous function benchmark optimization problems and sometimes even better than FEA-PSO. The main problem is that PI-PSO requires many more fitness evaluations than the other algorithms. However, our comparative performance and scaling experiments showed a number of interesting results. First, PI-PSO with two particles per dimension was often able to achieve the same or better performance than PSO with 128 particles per dimension. Second, even in those cases

where PI-PSO did not perform as well as PSO on a problem of a particular dimension, when the dimension was increased, PI-PSO would often outperform PSO.

- **Actor-Based DFEA:** As developed, DFEA is distributed only in terms of state but leaves open questions of concurrency and synchrony. We provide an implementation based on the Actor model that explores the implications of parallelism and asynchrony for our blackboard architecture.

9.2 Future Work

A tree can get taller or broader, and the same is true for research. Here we summarize the problems and questions that the research in this dissertation has generated for future work.

- **Best Factor Architecture**. For our benchmark functions, the structure of the functions suggested a simple factor size of two, which limited us to an overlap size of one. But what is the best factor architecture for any given problem where factor architecture concerns both the factor size and the factor overlap? One obstacle to investigating this problem is that as factor sizes increase, they become subject to hitchhiking—the very problem we were trying to avoid. If we were to use PI-PSO as the optimizer in either FEA or DFEA, we could avoid

this problem. Additionally, the use of PI-PSO might improve the inclusion of a factor covering all variables to avoid pseudo-optima. Finally, the best factor architecture might also evolve as the algorithm progresses and this may be specific to the local contexts.

- **Neighbor Relation**. Throughout it was assumed that the neighbor relation induced a fully connected communications topology on the sub-population. Even with relaxed consensus, if there are sufficient Share Steps, any arbitrated value will make it to any local context eventually. But if future research on the best factor architecture found that a factor architecture favored some set of factors that were disjoint, the current neighbor relation would lead to a disconnected communications topology. Future research would need to modify the algorithm and investigate the ramifications of such communication topologies.

- **Optimal Arbitration Order**. All of the experiment used variable order $X_1, X_2, ..., X_d$ as the arbitration order. Although this order leads to Pareto improvements, any order would lead to Pareto improvements under the current merge operation. Future research would investigate the optimal arbitration order and whether this is dependent on the factor architecture. If it turns out to be the case that the factor architecture should change over time, how does the arbitration order change?

- **Alternative Merge Operations**. All of our algorithms, FEA, DFEA, and PI-PSO use a merge operation for the blackboard architecture. First, we can only have one value of any given variable. Second, when we have conflicting values, we pick the value that leads to an improvement in fitness. This is just one possible merge operation, and there may be other alternatives.

  For example, the original DFEA uses this merge operation for individual blackboards. This means that any given c_i was determined to be a Pareto improvement in the context of the other c_j. However, when the blackboards are brought into consensus, the result is not a set of values that were ever evaluated together. And yet the original DFEA does not perform badly. This suggests there may be other ways to think about the merge operation.

  Another example is suggested by the case of Simulated Annealing. In Simulated Annealing, we sometimes accept inferior successors. Using our framework, these are not Pareto improvements. In the context of our research, this might mean that not all hitchhiking is bad. We might need to look at hitchhiking with a more nuanced view.

- **Apply Framework to Other EAs**. We only applied the framework to Particle Swarm Optimization to develop PI-PSO. It is worth exploring whether and how we can apply the framework to Genetic Algorithm, Hill Climbing, and other

algorithms to get single population results similar to the FEA variants.

- **Pareto Efficiency versus Pareto Improvements**. All of our algorithms are based on making Pareto improvements. In fact, they are based on making very particular Pareto improvements. While Pareto efficiency would require an exponential number of comparisons, it might be possible to explore other options. For example, we might try to sample from a number of randomly generated Pareto improvements that try more and different combinations. We might also investigate how we could combine the significantly smaller number of possibilities in FEA as compared to PI-PSO.

- **Apply Algorithms to Different Problems**. For the most part, in order to keep reign in the research to a certain degree and have largely comparable results, we used the continuous function benchmark optimization problems. But there are other forms of optimization problems such as Integer optimization, Categorical optimization and Combinatorial optimization. It would be interesting to see how these algorithm would perform on those kinds of problems and what, if any, changes would need to be made.

- **Improve Actor Implementation** First, one of the hallmarks of the Actor model is resilience, and Erlang is famous for the aphorism, "let it crash." In the Actor model, exceptions are not caught. Instead, the actor is crashed and

its parent brings up a fresh version. How does this actor recover its state or, if the actor never comes back, how do the other actors adjust to this situation? This would present challenges for our current DFEA Actor implementation. Second, research on DFEA has suggested that for many problems, coordination between the peers (consensus) can be relaxed. We would like to enhance our current DFEA Actor implementation to address both of these areas.

Appendix A

Benchmark Optimization Functions

This Appendix includes information about the Benchmark Optimization Functions used throughout this dissertation. Most of the functions are from [18] with a few coming from [8]. These particular functions were chosen because they are scalable to any dimension. For each function, we show the formula, interval of interest, minimizing vector and value, and plot a 3-dimensional projection of the function with (X_1, X_2) and a heatmap version showing the function from above. The color map used is the perceptually uniform "Viridis."

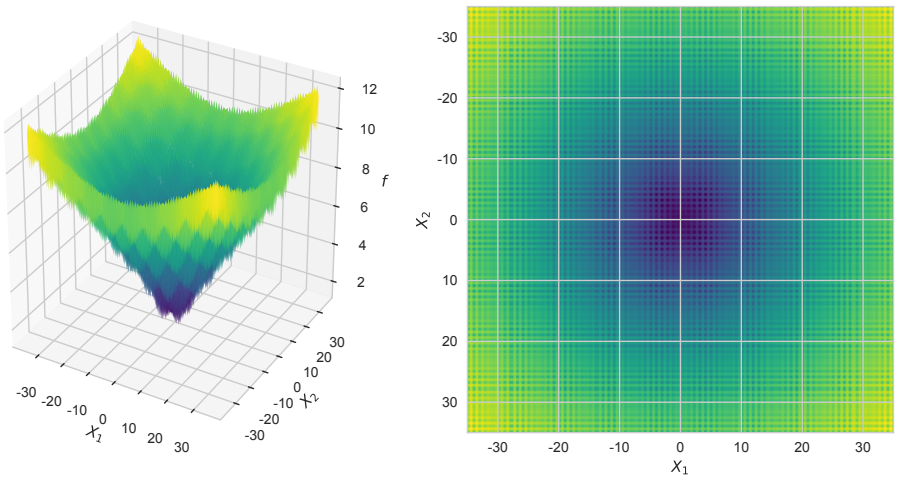
A.1 Ackley-1

$$f(\mathbf{X}) = -20e^{-0.02\sqrt{d^{-1}\sum_i^d X_i^2}} - e^{d^{-1}\sum_i^d cos(2\pi X_i)} + 20 + e$$

on interval: $-35.0 \leq X_i \leq 35.0$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.1: *Ackley-1 in 2 dimensions*



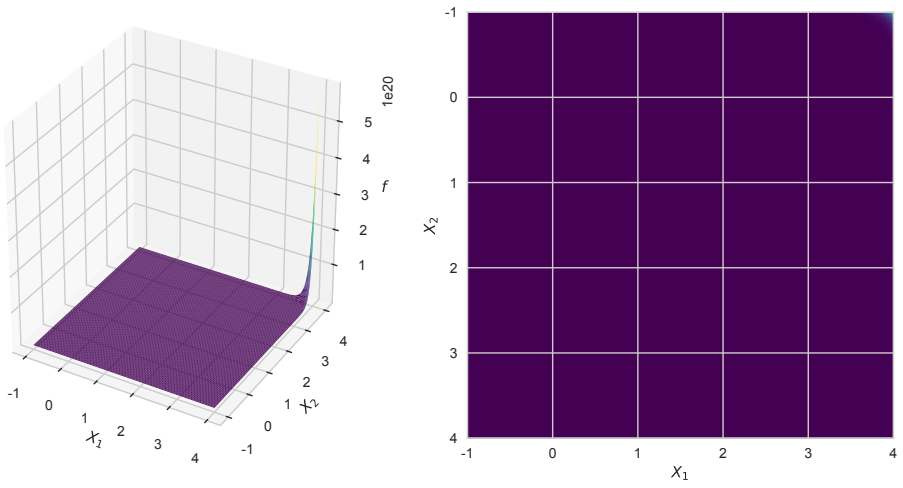
A.2 Brown

$$f(\mathbf{X}) = \sum_{i}^{d-1} (x_i^2)^{(x_{i+1}^2+1)} + (x_{i+1}^2)^{(x_i^2+1)}$$

on interval: $-1.0 \leq X_i \leq 4.0$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.2: *Brown in 2 dimensions*



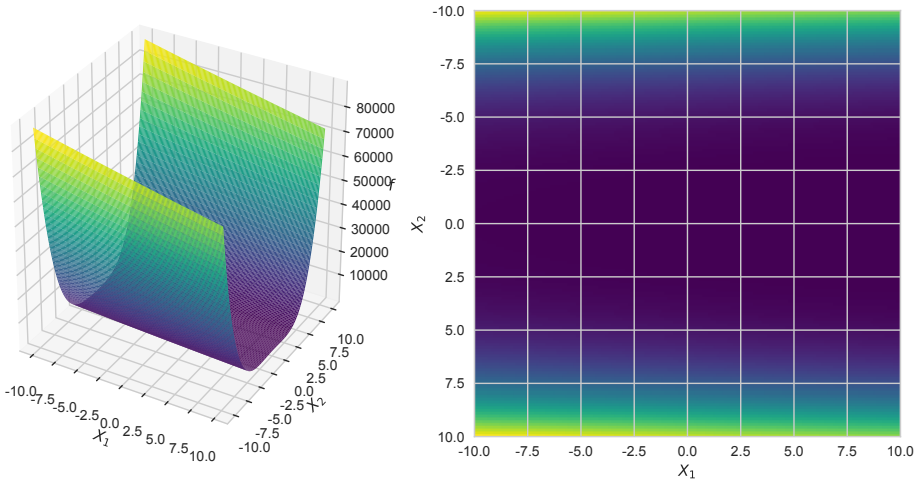
A.3 Dixon-Price

$$f(\mathbf{X}) = (x_1 - 1)^2 + \sum_{i=2}^{d} i(2x_i^2 - x_{i-1})^2$$

$$\text{on interval: } -10.0 \leq X_i \leq 10.0$$

$$\text{minimum at: } [2^{(\frac{2^i-2}{2^i})}] = 0$$

Figure A.3: *Dixon-Price in 2 dimensions*



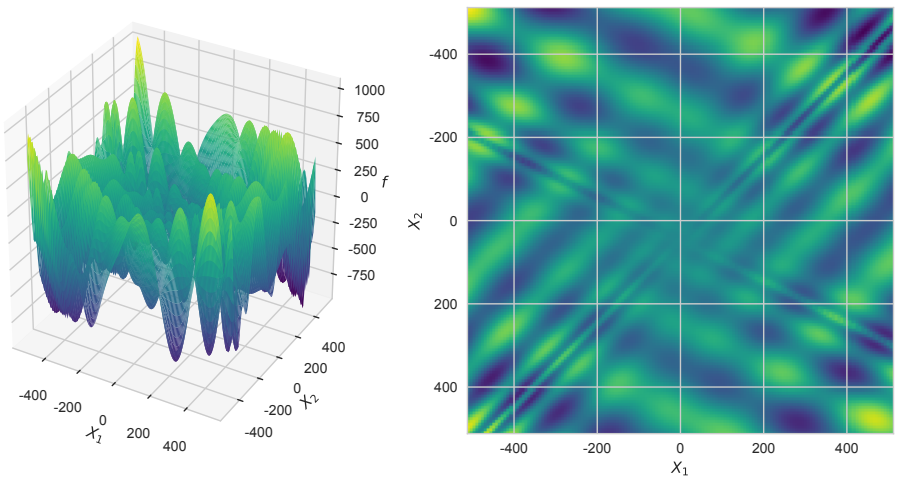
A.4 Eggholder

$$f(\mathbf{X}) = \sum_{i}^{d-1}[-(X_{i+1}+47)\sin\sqrt{|X_{i+1}+X_i/2+47|} - X_i \sin\sqrt{|X_i-(X_{i+1}+47)|}$$

on interval: $-512.0 \leq X_i \leq 512.0$

minimum at: *varies*

Figure A.4: *Eggholder in 2 dimensions*



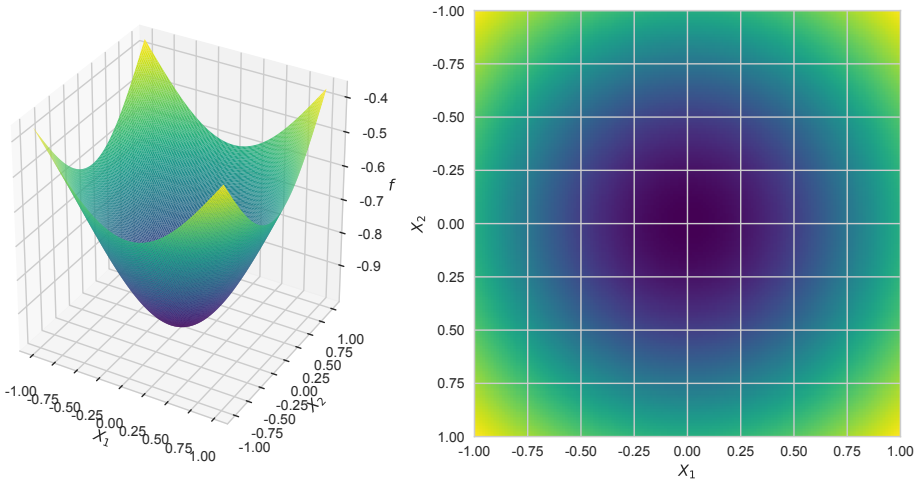
A.5 Exponential

$$f(\mathbf{X}) = -exp(-0.5\sum_{i}^{d} X_i^2)$$

on interval: $-1.0 \leq X_i \leq 1.0$

minimum at: $[0, 0, 0, ..., 0] = -1$

Figure A.5: *Exponential in 2 dimensions*



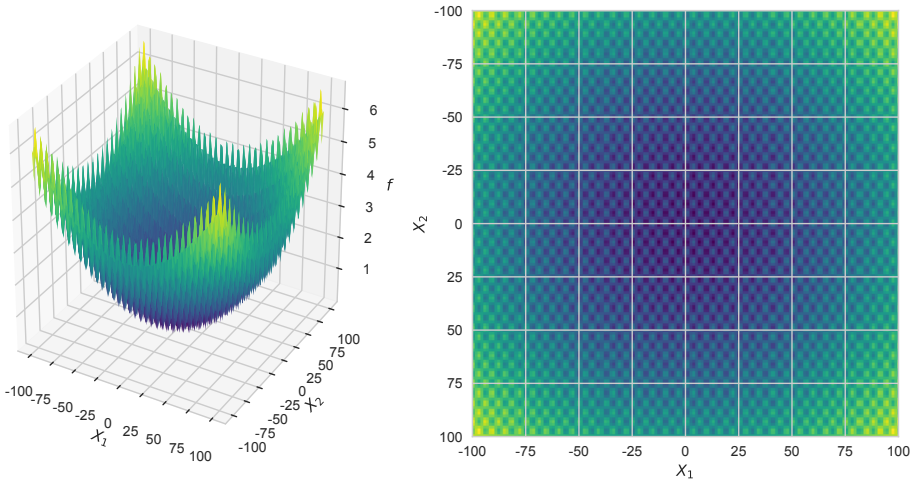
A.6 Griewank

$$f(\mathbf{X}) = \sum_{i}^{d} \frac{X^2}{4000} - \prod_{i}^{d} \cos(\frac{X_i}{\sqrt{i}}) + 1$$

on interval: $-1.0 \leq X_i \leq 1.0$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.6: *Griewank in 2 dimensions*



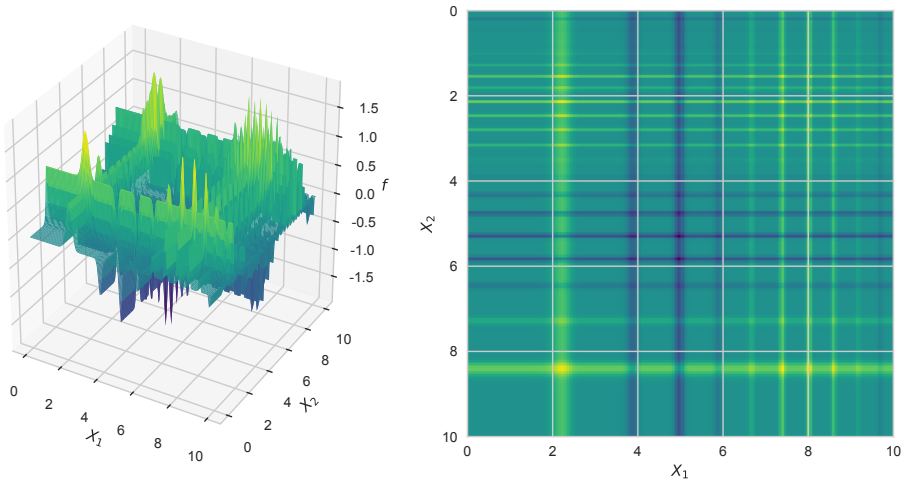
A.7 Michalewicz

$$f(\mathbf{X}) = -\Sigma_{i=1}^{d} \sin(X_i)[\sin(\frac{iX_i^2}{\pi})]^{2m}$$

on interval: $0 \leq X_i \leq 10.0$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.7: *Michalewicz in 2 dimensions*



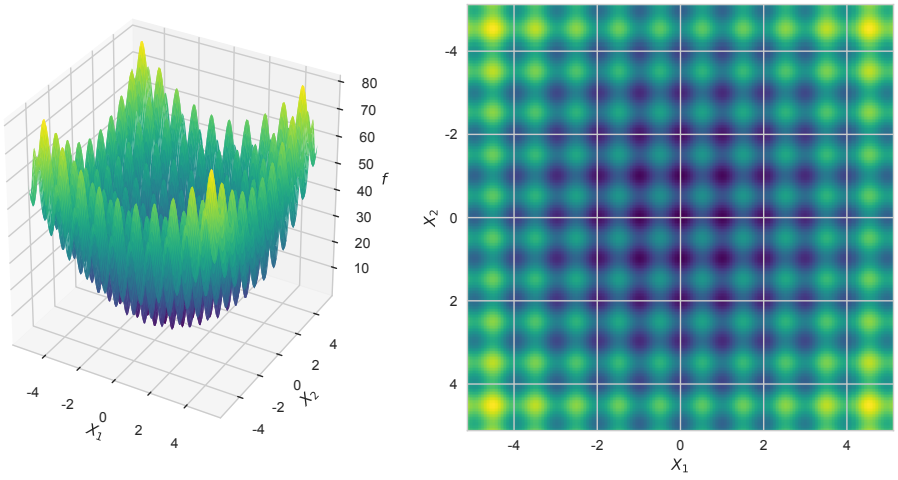
A.8 Rastrigin

$$f(\mathbf{X}) = 10d + \sum_{i=1}^{d}(X_i^2 + 10\cos(2\pi X_i))$$

on interval: $-5.12 \leq X_i \leq 5.12$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.8: *Rastrigin in 2 dimensions*



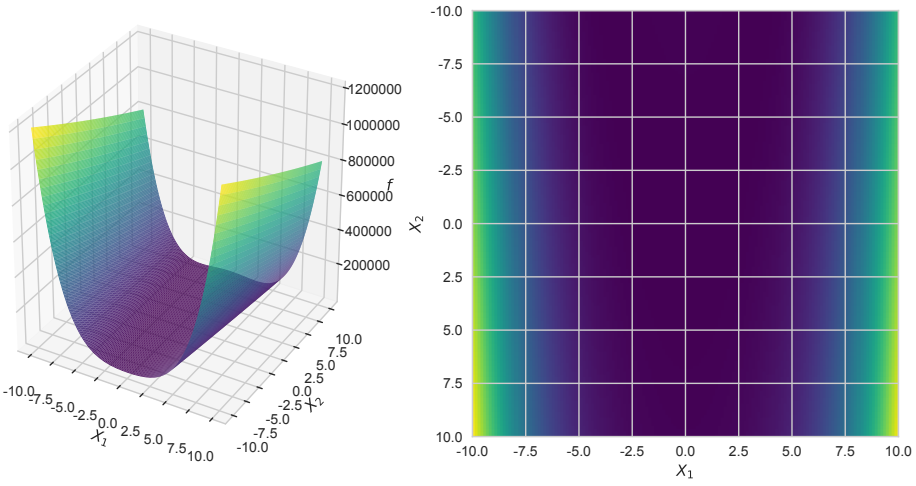
A.9 Rosenbrock

$$f(\mathbf{X}) = \sum^{d-1} [100(X_{i+1} - X_i^2)^2 + (X_i - 1)^2]$$

$$\text{on interval:} \quad -10.0 \leq X_i \leq 10.0$$

$$\text{minimum at:} \quad [1, 1, 1, ..., 1] = 0$$

Figure A.9: *Rosenbrock in 2 dimensions*



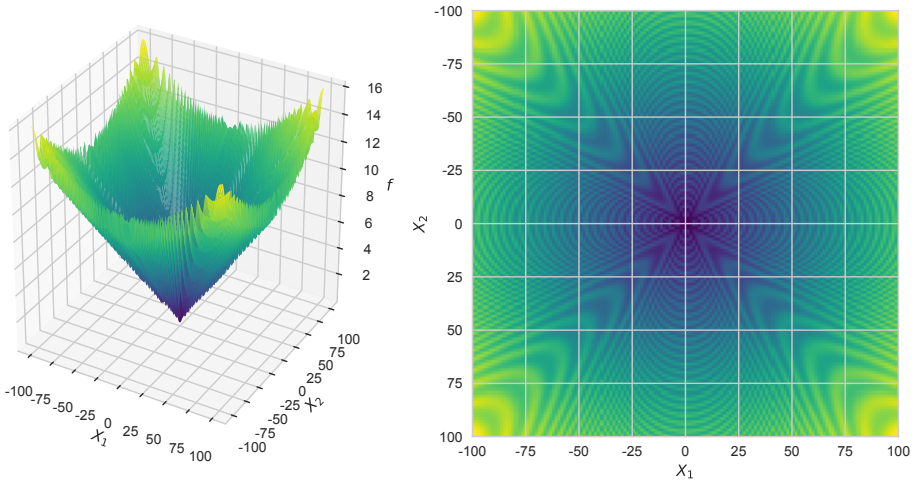
A.10 Salomon

$$f(\mathbf{X}) = 1 - \cos(2\pi\sqrt{\sum_{i=1}^{d} X_i^2}) + 0.1\sqrt{\sum_{i=1}^{d} X_i^2}$$

$$\text{on interval:} \quad -100.0 \leq X_i \leq 100.0$$

$$\text{minimum at:} \quad [0, 0, 0, ..., 0] = 0$$

Figure A.10: *Salomon in 2 dimensions*



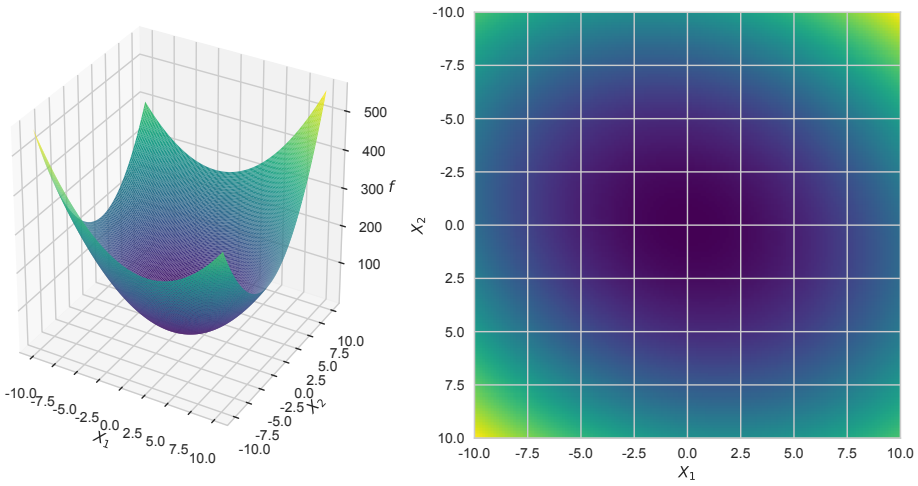
A.11 Sargan

$$f(\mathbf{X}) = \sum_{i=1} d(X_i^2 + 0.4 \sum_{j \neq i} X_i X_j)$$

$$\text{on interval: } -100.0 \leq X_i \leq 100.0$$

$$\text{minimum at: } [0, 0, 0, ..., 0] = 0$$

Figure A.11: *Sargan in 2 dimensions*



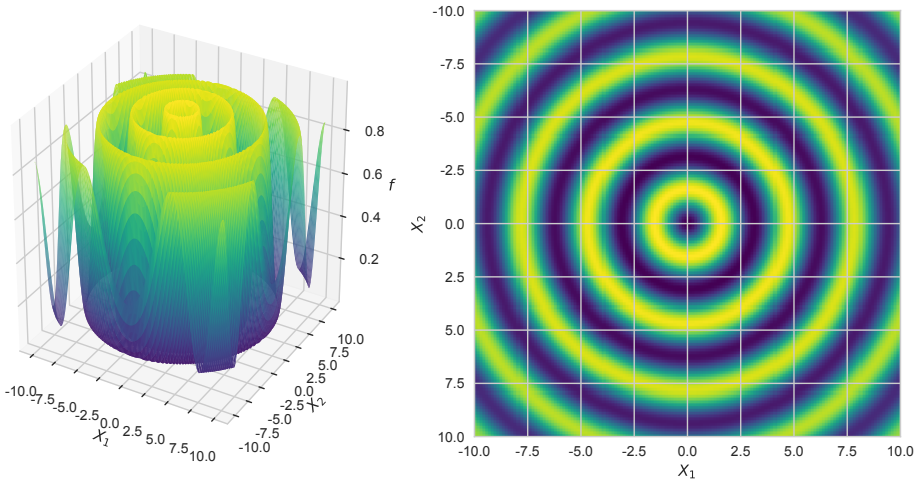
A.12 Schaffer-F6

$$f(\mathbf{X}) = \sum_{i=1}^{d} 0.5 + \frac{\sin^2(\sqrt{X_i^2 + X_{i+1}^2}) - 0.5}{1 + 0.001(X_i^2 + X_{i+1}^2)^2}$$

$$\text{on interval:} \quad -10.0 \leq X_i \leq 10.0$$

$$\text{minimum at:} \quad [0, 0, 0, ..., 0] = 0$$

Figure A.12: *Schaffer-F6 in 2 dimensions*



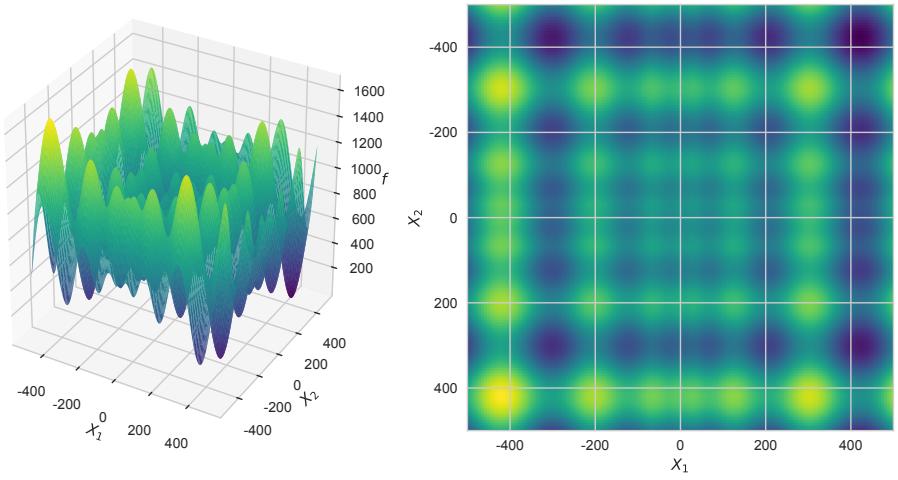
A.13 Schwefel

$$f(\mathbf{X}) = 418.9829d - \sum_{i=1}^{d} x_i \sin\sqrt{|x_i|}$$

$$\text{on interval:} \quad -500.0 \leq X_i \leq 500.0$$

$$\text{minimum at:} \quad [420.9867, 420.9867, ..., 420.9867] = 0$$

Figure A.13: *Schwefel in 2 dimensions*



(Used only in Chapter 3)

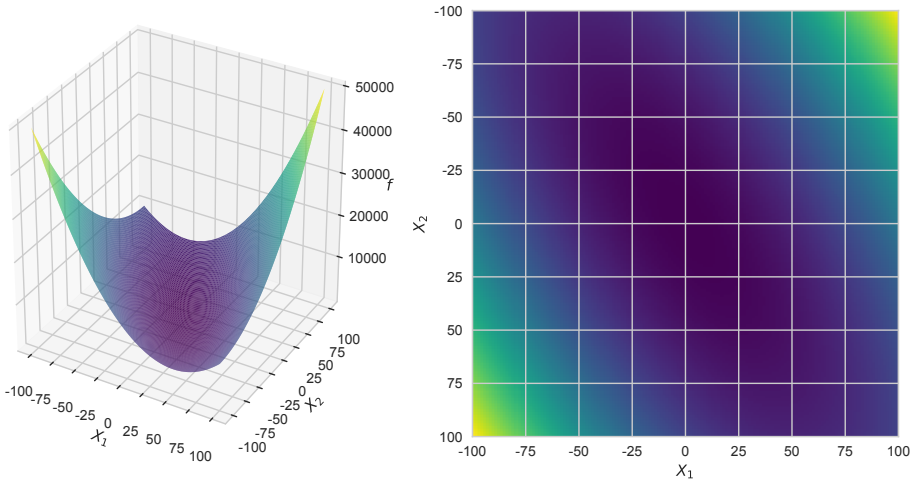
A.14 Schwefel-1.2

$$f(\mathbf{X}) = \sum_{i=1}^{d} (\sum_{j}^{i} X_j)^2$$

on interval: $-100.0 \leq X_i \leq 100.0$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.14: *Schwefel-1.2 in 2 dimensions*



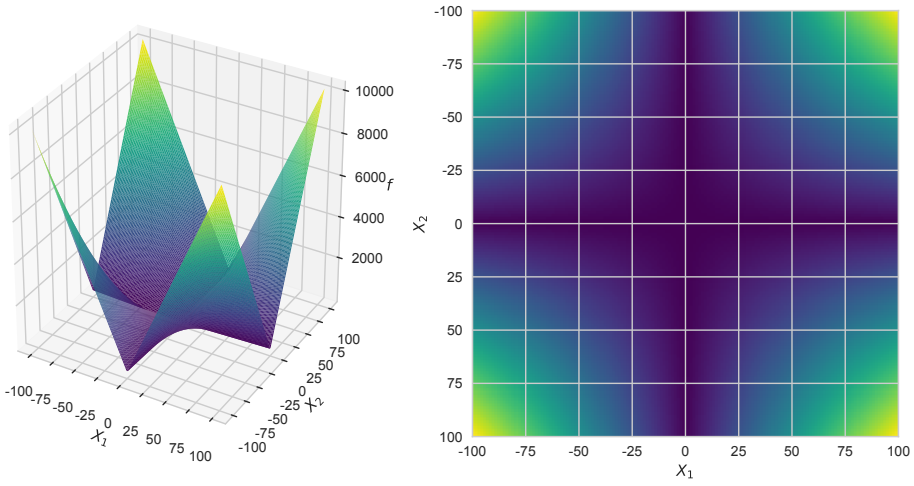
A.15 Schwefel-2.22

$$f(\mathbf{X}) = \sum_{i=1}^{d} |X_i| + \prod_{i}^{d} |X_i|$$

on interval: $-100.0 \leq X_i \leq 100.0$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.15: *Schwefel-2.22 in 2 dimensions*



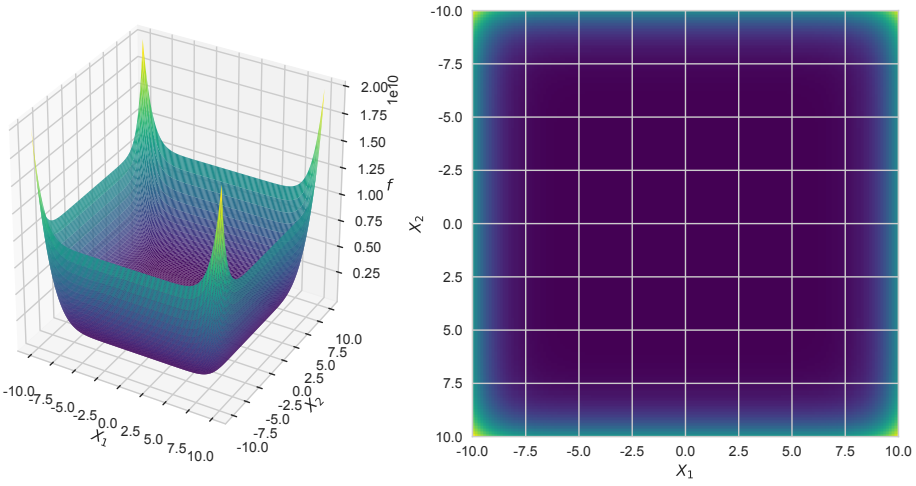
A.16 Schwefel-2.23

$$f(\mathbf{X}) = \sum_{i=1}^{d} X_i^{10}$$

on interval: $-10.0 \leq X_i \leq 10.0$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.16: *Schwefel-2.23 in 2 dimensions*



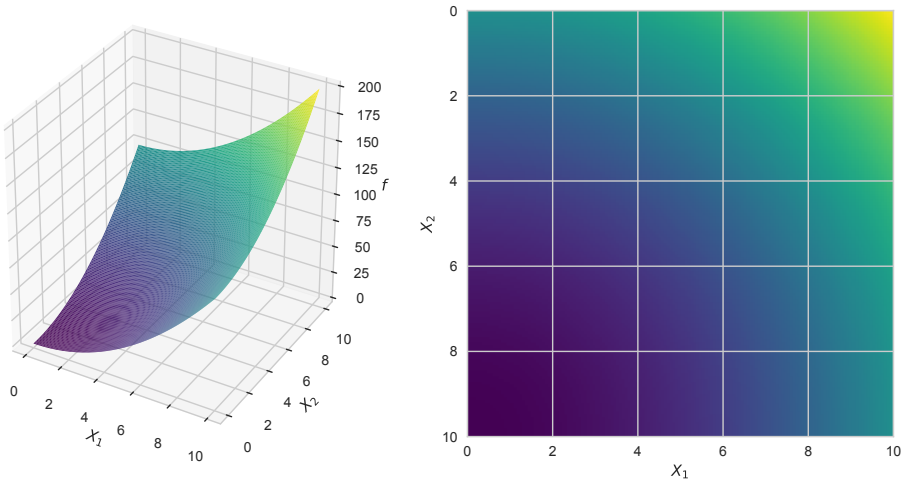
A.17 Sphere

$$f(\mathbf{X}) = \sum_{i}^{d} X_i^2$$

on interval: $0.0 \leq X_i \leq 10.0$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.17: *Sphere in 2 dimensions*



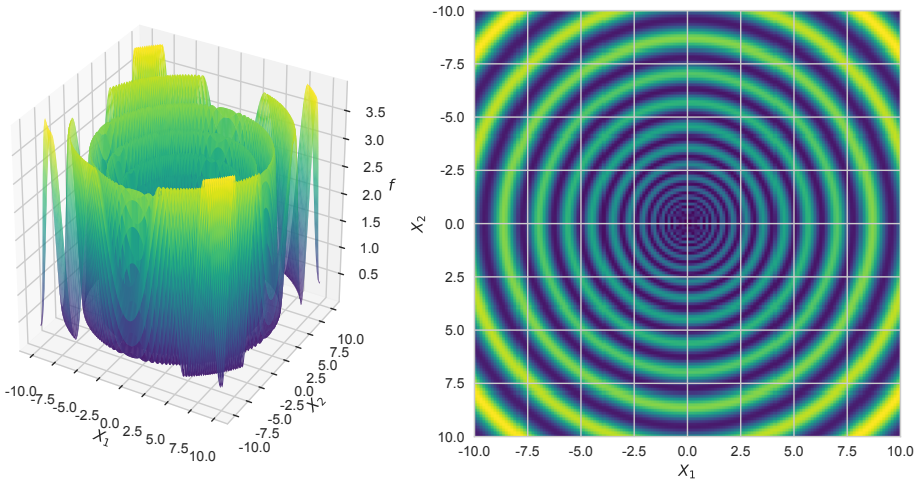
A.18 Stretched-V

$$f(\mathbf{X}) = \sum_{i}^{d-1} (X_{i+1}^2 + X_i^2)^{0.25} [\sin^2[50(X_{i+1}^2 + X_i^2)^{0.1}] + 0.1]$$

$$\text{on interval:} \quad -10.0 \leq X_i \leq 10.0$$

$$\text{minimum at:} \quad [0, 0, 0, ..., 0] = 0$$

Figure A.18: *Stretched-V in 2 dimensions*



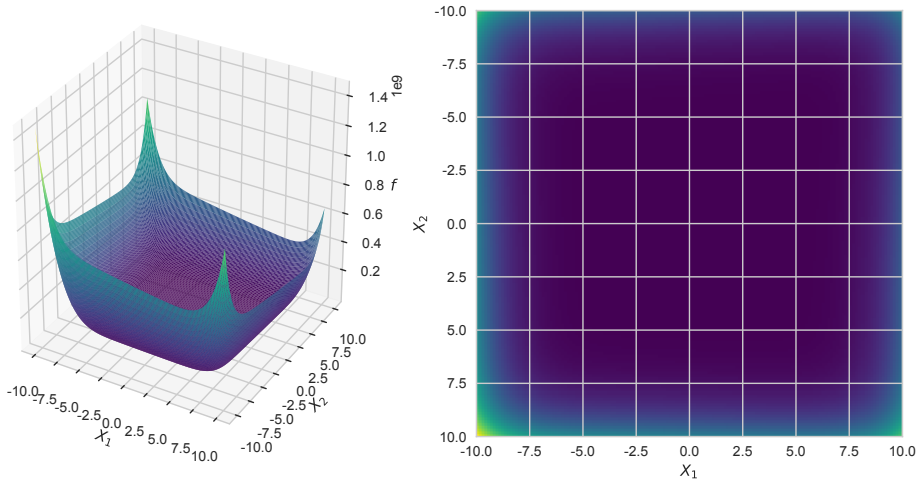
A.19 Whitley

$$f(\mathbf{X}) = \sum_{i=1}^{d}\sum_{j}^{d}[\frac{(100(X_i^2 - X_j)^2 + (1 - X_j)^2)^2}{4000} - \cos(100(X_i^2 - X_j)^2 + (1 - X_j)^2 + 1)]$$

$$\text{on interval: } -10.0 \leq X_i \leq 10.0$$

$$\text{minimum at: } [1, 1, 1, ..., 1] = 0$$

Figure A.19: *Whitley in 2 dimensions*



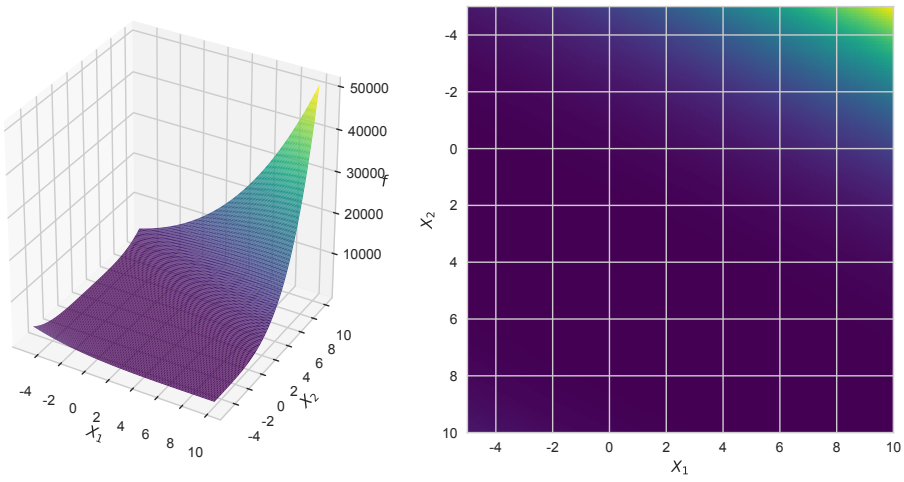
A.20 Zakharov

$$f(\mathbf{X}) = \sum_{i=1}^{d} X_i^2 + (0.5 \sum_{i=1}^{d} iX_i)^2 + (0.5 \sum_{i=1}^{d} iX_i)^4$$

on interval: $-5.0 \leq X_i \leq 10.0$

minimum at: $[0, 0, 0, ..., 0] = 0$

Figure A.20: *Zakharov in 2 dimensions*



Appendix B

Extended Chapter 7 Results

This chapter includes the full results for the $32d$ problems and charts for all dimensions for all 19 benchmark problems presented in Chapter 7.

Table B.1: *Ackley-1 Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	2.42e+00 (2.35e+00, 2.49e+00)	**1.54e-04** (1.20e-04, 1.88e-04)
4 (128)	2.14e+00 (2.08e+00, 2.20e+00)	**1.01e-05** (5.73e-06, 1.54e-05)
8 (256)	1.93e+00 (1.86e+00, 1.98e+00)	**1.11e-07** (4.44e-16, 3.32e-07)
16 (512)	1.85e+00 (1.77e+00, 1.92e+00)	**4.44e-16** (4.44e-16, 4.44e-16)
32 (1024)	1.65e+00 (1.58e+00, 1.72e+00)	**4.44e-16** (4.44e-16, 4.44e-16)
64 (2048)	1.41e+00 (1.33e+00, 1.49e+00)	**4.44e-16** (4.44e-16, 4.44e-16)
128 (4096)	1.25e+00 (1.18e+00, 1.30e+00)	**4.44e-16** (4.44e-16, 4.44e-16)
256 (8192)	1.14e+00 (1.08e+00, 1.20e+00)	**4.44e-16** (4.44e-16, 4.44e-16)

Figure B.1: *Ackley-1 Benchmark: PSO v. PI-PSO Scaling*

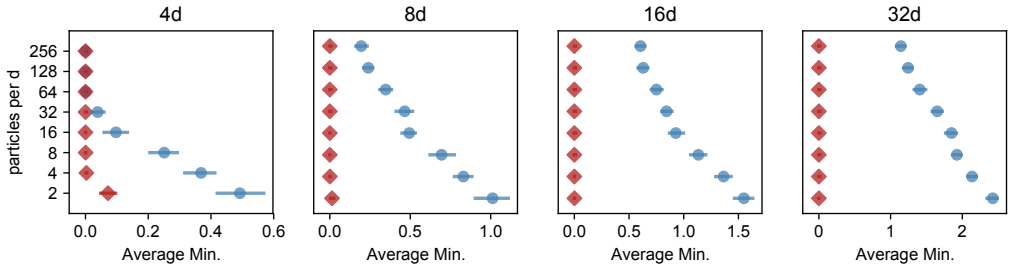


Table B.2: *Brown Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	1.95e+01 (1.42e+01, 2.50e+01)	**2.16e-07** (1.62e-07, 2.87e-07)
4 (128)	1.30e+01 (1.02e+01, 1.66e+01)	**1.99e-08** (1.62e-08, 2.37e-08)
8 (256)	1.04e+01 (7.92e+00, 1.36e+01)	**2.41e-09** (1.91e-09, 3.08e-09)
16 (512)	7.23e+00 (6.30e+00, 8.07e+00)	**3.41e-10** (2.90e-10, 3.91e-10)
32 (1024)	6.46e+00 (5.29e+00, 7.59e+00)	**5.72e-11** (4.75e-11, 6.74e-11)
64 (2048)	3.51e+00 (2.75e+00, 4.24e+00)	**8.90e-12** (7.47e-12, 1.04e-11)
128 (4096)	2.96e+00 (2.26e+00, 3.49e+00)	**1.51e-12** (1.32e-12, 1.71e-12)
256 (8192)	1.57e+00 (1.15e+00, 2.01e+00)	**2.90e-13** (2.36e-13, 3.57e-13)

Figure B.2: *Brown Benchmark: PSO v. PI-PSO Scaling*

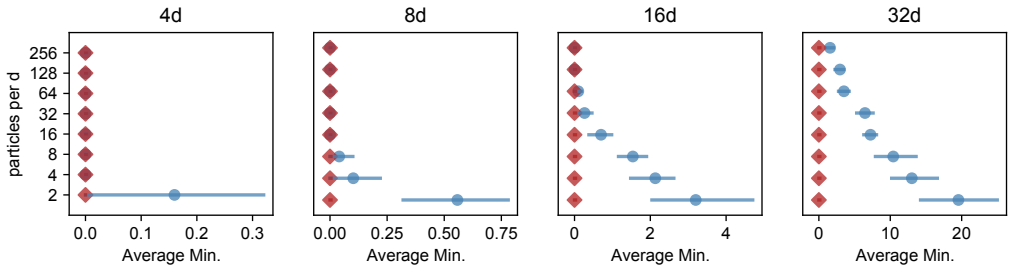


Figure B.3: *Dixon-Price Benchmark: PSO v. PI-PSO Scaling*

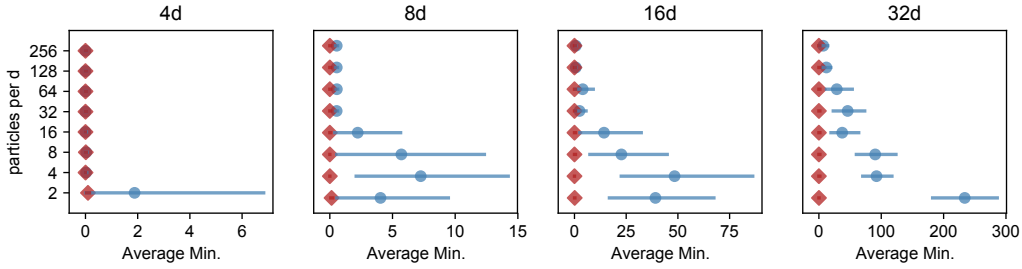


Table B.3: *Dixon-Price Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	2.34e+02 (1.82e+02, 2.86e+02)	**1.03e-01** (6.61e-03, 3.02e-01)
4 (128)	9.24e+01 (7.03e+01, 1.17e+02)	**1.11e-03** (9.33e-04, 1.30e-03)
8 (256)	9.03e+01 (6.01e+01, 1.24e+02)	**1.88e-04** (1.64e-04, 2.12e-04)
16 (512)	3.74e+01 (1.94e+01, 6.39e+01)	**2.96e-05** (2.49e-05, 3.45e-05)
32 (1024)	4.61e+01 (2.30e+01, 7.35e+01)	**5.96e-06** (4.81e-06, 7.25e-06)
64 (2048)	2.87e+01 (1.21e+01, 5.35e+01)	**9.67e-07** (8.08e-07, 1.09e-06)
128 (4096)	1.20e+01 (4.95e+00, 1.90e+01)	**1.85e-07** (1.63e-07, 2.10e-07)
256 (8192)	7.07e+00 (1.21e+00, 1.41e+01)	**4.32e-08** (3.62e-08, 5.33e-08)

Table B.4: *Eggholder Benchmark Results 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	-1.52e+04 (-1.56e+04, -1.49e+04)	**-2.30e+04** (-2.33e+04, -2.27e+04)
4 (128)	-1.56e+04 (-1.61e+04, -1.52e+04)	**-2.32e+04** (-2.34e+04, -2.28e+04)
8 (256)	-1.64e+04 (-1.68e+04, -1.61e+04)	**-2.35e+04** (-2.37e+04, -2.32e+04)
16 (512)	-1.67e+04 (-1.71e+04, -1.63e+04)	**-2.33e+04** (-2.36e+04, -2.30e+04)
32 (1024)	-1.72e+04 (-1.77e+04, -1.68e+04)	**-2.37e+04** (-2.39e+04, -2.34e+04)
64 (2048)	-1.76e+04 (-1.79e+04, -1.73e+04)	**-2.39e+04** (-2.42e+04, -2.36e+04)
128 (4096)	-1.85e+04 (-1.88e+04, -1.82e+04)	**-2.38e+04** (-2.42e+04, -2.36e+04)
256 (8192)	-1.86e+04 (-1.90e+04, -1.83e+04)	**-2.37e+04** (-2.40e+04, -2.33e+04)

Figure B.4: *Eggholder Benchmark: PSO v. PI-PSO Scaling*

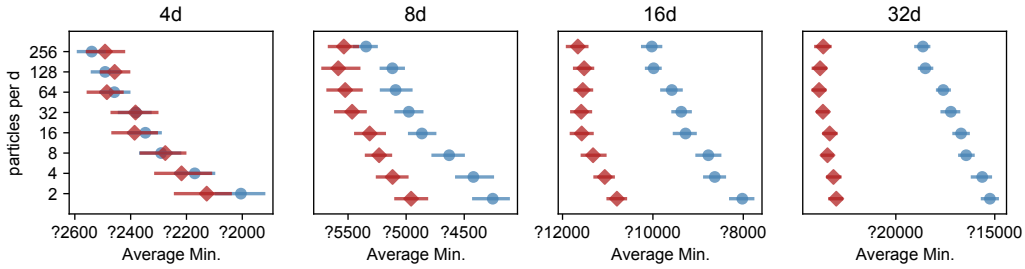


Table B.5: *Exponential Benchmark Results 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	-9.93e-01 (-9.94e-01, -9.92e-01)	**-1.00e+00** (-1.00e+00, -1.00e+00)
4 (128)	-9.98e-01 (-9.98e-01, -9.97e-01)	**-1.00e+00** (-1.00e+00, -1.00e+00)
8 (256)	-9.99e-01 (-9.99e-01, -9.99e-01)	**-1.00e+00** (-1.00e+00, -1.00e+00)
16 (512)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)
32 (1024)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)
64 (2048)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)
128 (4096)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)
256 (8192)	-1.00e+00 (-1.00e+00, -1.00e+00)	**-1.00e+00** (-1.00e+00, -1.00e+00)

Figure B.5: *Exponential Benchmark: PSO v. PI-PSO Scaling*

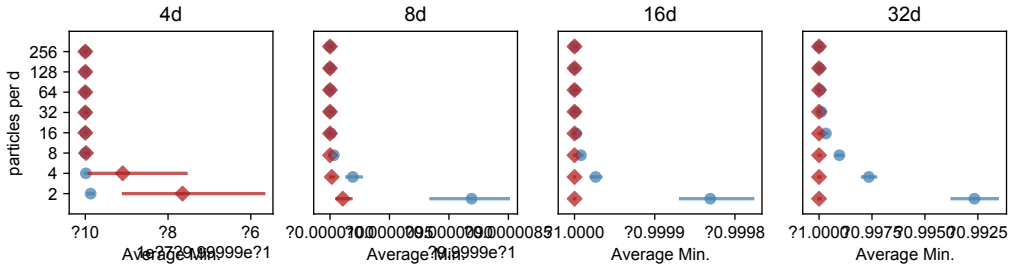


Table B.6: *Eggholder Benchmark Results 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	-1.52e+04 (-1.56e+04, -1.49e+04)	**-2.30e+04** (-2.33e+04, -2.27e+04)
4 (128)	-1.56e+04 (-1.61e+04, -1.52e+04)	**-2.32e+04** (-2.34e+04, -2.28e+04)
8 (256)	-1.64e+04 (-1.68e+04, -1.61e+04)	**-2.35e+04** (-2.37e+04, -2.32e+04)
16 (512)	-1.67e+04 (-1.71e+04, -1.63e+04)	**-2.33e+04** (-2.36e+04, -2.30e+04)
32 (1024)	-1.72e+04 (-1.77e+04, -1.68e+04)	**-2.37e+04** (-2.39e+04, -2.34e+04)
64 (2048)	-1.76e+04 (-1.79e+04, -1.73e+04)	**-2.39e+04** (-2.42e+04, -2.36e+04)
128 (4096)	-1.85e+04 (-1.88e+04, -1.82e+04)	**-2.38e+04** (-2.42e+04, -2.36e+04)
256 (8192)	-1.86e+04 (-1.90e+04, -1.83e+04)	**-2.37e+04** (-2.40e+04, -2.33e+04)

Figure B.6: *Eggholder Benchmark: PSO v. PI-PSO Scaling*

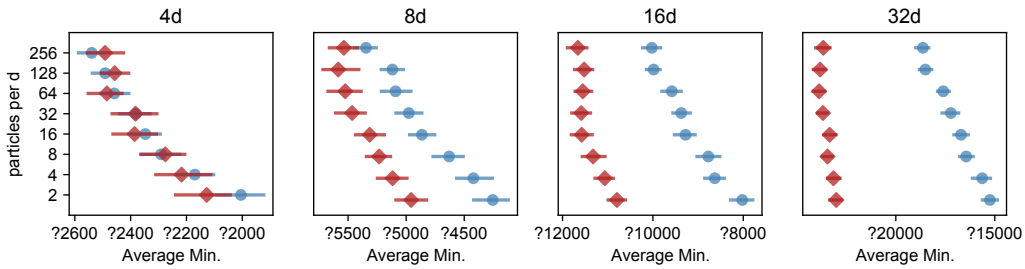


Table B.7: *Griewank Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	1.03e+00 (1.02e+00, 1.04e+00)	**1.65e-02** (1.00e-02, 2.33e-02)
4 (128)	8.65e-01 (8.31e-01, 8.98e-01)	**1.03e-01** (7.75e-02, 1.30e-01)
8 (256)	5.90e-01 (5.40e-01, 6.42e-01)	**1.44e-01** (1.15e-01, 1.72e-01)
16 (512)	3.22e-01 (2.79e-01, 3.64e-01)	**1.75e-01** (1.46e-01, 2.08e-01)
32 (1024)	**1.49e-01** (1.27e-01, 1.73e-01)	1.97e-01 (1.65e-01, 2.29e-01)
64 (2048)	**9.10e-02** (7.36e-02, 1.09e-01)	1.77e-01 (1.46e-01, 2.05e-01)
128 (4096)	**4.13e-02** (3.16e-02, 5.15e-02)	1.60e-01 (1.25e-01, 1.92e-01)
256 (8192)	**2.66e-02** (2.10e-02, 3.32e-02)	2.16e-01 (1.88e-01, 2.44e-01)

Figure B.7: *Griewank Benchmark: PSO v. PI-PSO Scaling*

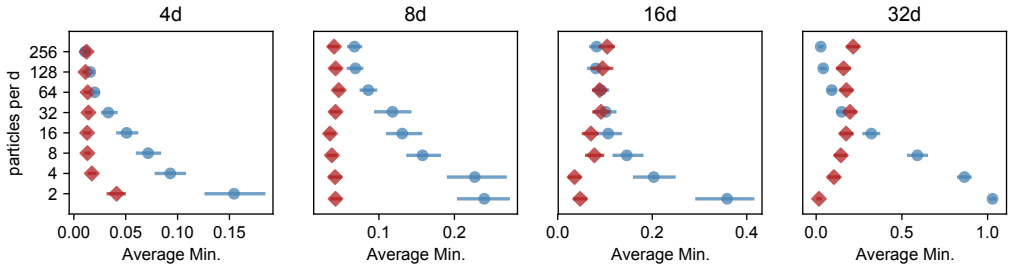


Table B.8: *Michalewicz Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	-7.42e+00 (-7.77e+00, -7.13e+00)	**-3.18e+01** (-3.19e+01, -3.18e+01)
4 (128)	-8.03e+00 (-8.40e+00, -7.66e+00)	**-3.19e+01** (-3.19e+01, -3.19e+01)
8 (256)	-8.24e+00 (-8.54e+00, -7.96e+00)	**-3.19e+01** (-3.19e+01, -3.19e+01)
16 (512)	-8.57e+00 (-8.90e+00, -8.23e+00)	**-3.19e+01** (-3.19e+01, -3.19e+01)
32 (1024)	-9.30e+00 (-9.60e+00, -9.06e+00)	**-3.19e+01** (-3.19e+01, -3.19e+01)
64 (2048)	-9.59e+00 (-9.90e+00, -9.27e+00)	**-3.19e+01** (-3.19e+01, -3.19e+01)
128 (4096)	-1.00e+01 (-1.04e+01, -9.68e+00)	**-3.19e+01** (-3.19e+01, -3.19e+01)
256 (8192)	-1.07e+01 (-1.12e+01, -1.03e+01)	**-3.19e+01** (-3.19e+01, -3.19e+01)

Figure B.8: *Michalewicz Benchmark: PSO v. PI-PSO Scaling*

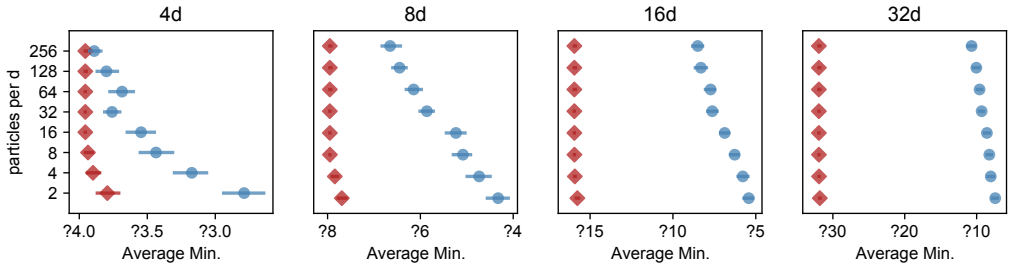


Table B.9: *Rastrigin Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	1.56e+02 (1.46e+02, 1.66e+02)	**4.54e-06** (1.39e-06, 9.46e-06)
4 (128)	1.33e+02 (1.24e+02, 1.42e+02)	**1.48e-07** (2.33e-08, 3.55e-07)
8 (256)	1.15e+02 (1.07e+02, 1.23e+02)	**0.00e+00** (0.00e+00, 0.00e+00)
16 (512)	1.04e+02 (9.70e+01, 1.11e+02)	**0.00e+00** (0.00e+00, 0.00e+00)
32 (1024)	7.97e+01 (7.35e+01, 8.62e+01)	**0.00e+00** (0.00e+00, 0.00e+00)
64 (2048)	7.21e+01 (6.65e+01, 7.77e+01)	**0.00e+00** (0.00e+00, 0.00e+00)
128 (4096)	6.69e+01 (6.23e+01, 7.22e+01)	**0.00e+00** (0.00e+00, 0.00e+00)
256 (8192)	5.54e+01 (5.09e+01, 6.02e+01)	**0.00e+00** (0.00e+00, 0.00e+00)

Figure B.9: *Rastrigin Benchmark: PSO v. PI-PSO Scaling*

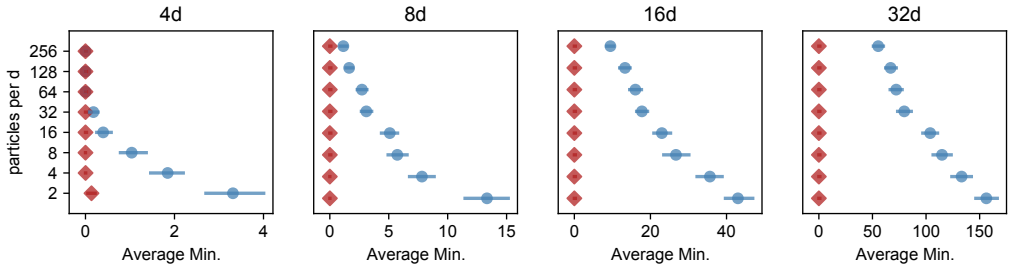


Figure B.10: *Rosenbrock Benchmark: PSO v. PI-PSO Scaling*

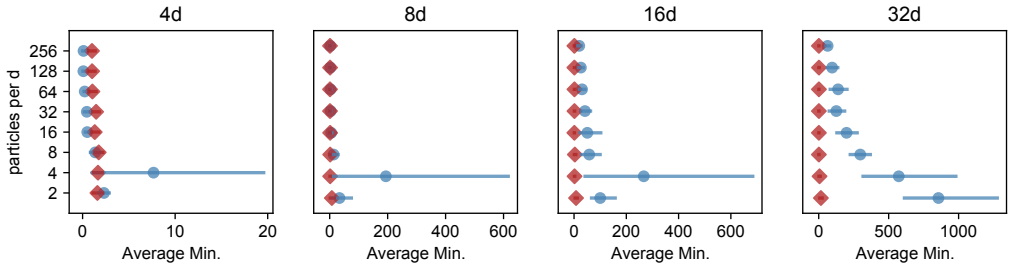


Table B.10: *Rosenbrock Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	8.56e+02 (6.13e+02, 1.28e+03)	**1.56e+01** (7.57e+00, 2.46e+01)
4 (128)	5.72e+02 (3.17e+02, 9.80e+02)	**5.87e+00** (1.77e+00, 1.05e+01)
8 (256)	2.97e+02 (2.25e+02, 3.69e+02)	**1.25e+00** (8.62e-01, 1.67e+00)
16 (512)	1.99e+02 (1.30e+02, 2.75e+02)	**1.11e+00** (7.71e-01, 1.54e+00)
32 (1024)	1.26e+02 (7.53e+01, 1.85e+02)	**1.17e+00** (7.63e-01, 1.61e+00)
64 (2048)	1.38e+02 (8.22e+01, 2.02e+02)	**1.41e+00** (9.75e-01, 1.83e+00)
128 (4096)	9.59e+01 (6.11e+01, 1.36e+02)	**1.86e+00** (9.17e-01, 3.77e+00)
256 (8192)	6.38e+01 (5.03e+01, 7.84e+01)	**1.20e+00** (7.09e-01, 1.82e+00)

Table B.11: *Salomon Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	3.18e+00 (2.93e+00, 3.54e+00)	**1.79e+00** (1.67e+00, 1.91e+00)
4 (128)	2.31e+00 (2.17e+00, 2.43e+00)	**1.74e+00** (1.61e+00, 1.87e+00)
8 (256)	**1.67e+00** (1.60e+00, 1.74e+00)	1.90e+00 (1.77e+00, 2.03e+00)
16 (512)	**1.18e+00** (1.11e+00, 1.23e+00)	1.87e+00 (1.74e+00, 1.97e+00)
32 (1024)	**8.55e-01** (8.18e-01, 8.94e-01)	1.88e+00 (1.79e+00, 1.98e+00)
64 (2048)	**6.39e-01** (6.14e-01, 6.64e-01)	1.85e+00 (1.74e+00, 1.95e+00)
128 (4096)	**5.18e-01** (4.92e-01, 5.40e-01)	1.79e+00 (1.66e+00, 1.91e+00)
256 (8192)	**4.14e-01** (3.96e-01, 4.30e-01)	1.91e+00 (1.82e+00, 2.02e+00)

Figure B.11: *Salomon Benchmark: PSO v. PI-PSO Scaling*

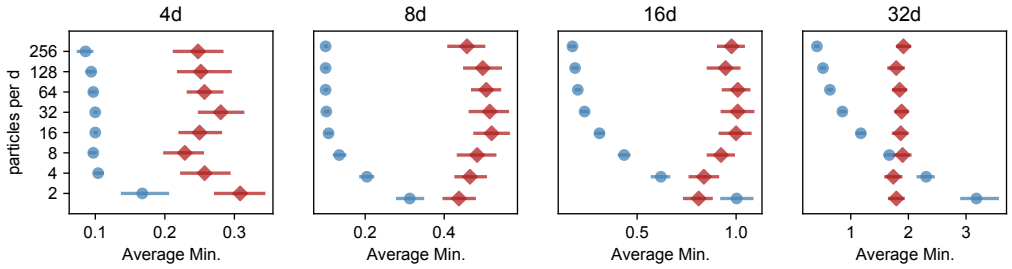


Table B.12: *Sargan Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	9.83e+01 (8.47e+01, 1.12e+02)	**6.93e-04** (5.52e-04, 8.65e-04)
4 (128)	3.57e+01 (3.06e+01, 4.01e+01)	**8.23e-05** (6.31e-05, 1.00e-04)
8 (256)	1.37e+01 (1.21e+01, 1.57e+01)	**6.60e-06** (4.31e-06, 9.03e-06)
16 (512)	5.14e+00 (4.56e+00, 5.71e+00)	**4.11e-07** (2.10e-07, 6.74e-07)
32 (1024)	1.91e+00 (1.63e+00, 2.16e+00)	**7.54e-09** (1.13e-09, 1.82e-08)
64 (2048)	8.20e-01 (7.06e-01, 9.33e-01)	**1.20e-11** (0.00e+00, 3.60e-11)
128 (4096)	3.38e-01 (3.00e-01, 3.75e-01)	**0.00e+00** (0.00e+00, 0.00e+00)
256 (8192)	1.22e-01 (1.04e-01, 1.40e-01)	**2.42e-10** (1.03e-12, 8.31e-10)

Figure B.12: *Sargan Benchmark: PSO v. PI-PSO Scaling*

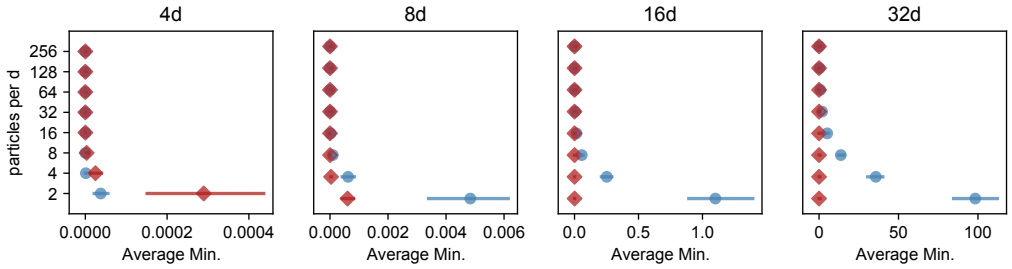


Table B.13: *Schaffer-F6 Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	3.49e+00 (3.26e+00, 3.73e+00)	**6.12e-01** (5.42e-01, 6.90e-01)
4 (128)	3.09e+00 (2.90e+00, 3.27e+00)	**6.54e-01** (5.67e-01, 7.56e-01)
8 (256)	2.81e+00 (2.64e+00, 3.00e+00)	**5.15e-01** (4.47e-01, 5.80e-01)
16 (512)	2.61e+00 (2.44e+00, 2.76e+00)	**5.38e-01** (4.66e-01, 6.16e-01)
32 (1024)	2.23e+00 (2.06e+00, 2.39e+00)	**4.61e-01** (3.92e-01, 5.36e-01)
64 (2048)	2.25e+00 (2.09e+00, 2.40e+00)	**4.72e-01** (4.07e-01, 5.32e-01)
128 (4096)	1.96e+00 (1.79e+00, 2.14e+00)	**4.66e-01** (3.97e-01, 5.37e-01)
256 (8192)	1.83e+00 (1.69e+00, 1.95e+00)	**3.98e-01** (3.44e-01, 4.55e-01)

Figure B.13: *Schaffer-F6 Benchmark: PSO v. PI-PSO Scaling*

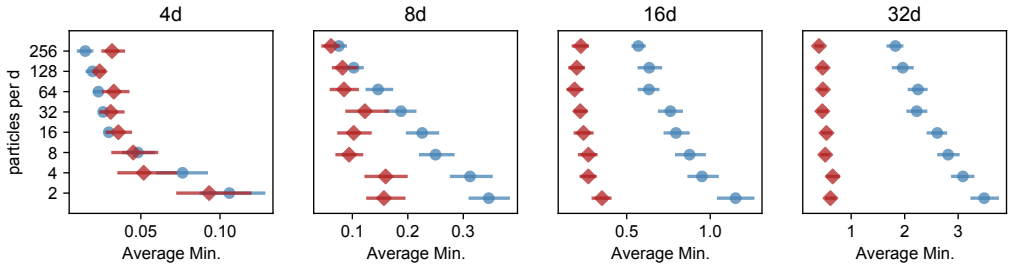


Table B.14: *Schwefel-1.2 Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	1.69e+04 (1.54e+04, 1.84e+04)	**1.52e+03** (1.37e+03, 1.66e+03)
4 (128)	1.31e+04 (1.16e+04, 1.44e+04)	**6.73e+02** (6.07e+02, 7.39e+02)
8 (256)	1.07e+04 (9.56e+03, 1.19e+04)	**4.21e+02** (3.84e+02, 4.63e+02)
16 (512)	7.22e+03 (6.16e+03, 8.29e+03)	**3.03e+02** (2.65e+02, 3.38e+02)
32 (1024)	5.48e+03 (4.60e+03, 6.48e+03)	**3.05e+02** (2.80e+02, 3.33e+02)
64 (2048)	3.21e+03 (2.66e+03, 3.79e+03)	**3.22e+02** (2.80e+02, 3.64e+02)
128 (4096)	2.17e+03 (1.67e+03, 2.79e+03)	**2.74e+02** (2.51e+02, 3.00e+02)
256 (8192)	1.23e+03 (9.41e+02, 1.50e+03)	**2.68e+02** (2.40e+02, 2.96e+02)

Figure B.14: *Schwefel 1.2 Benchmark: PSO v. PI-PSO Scaling*

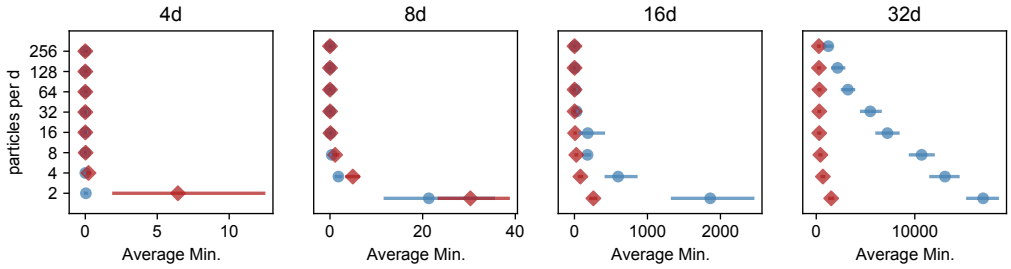


Figure B.15: *Schwefel 2.22 Benchmark: PSO v. PI-PSO Scaling*

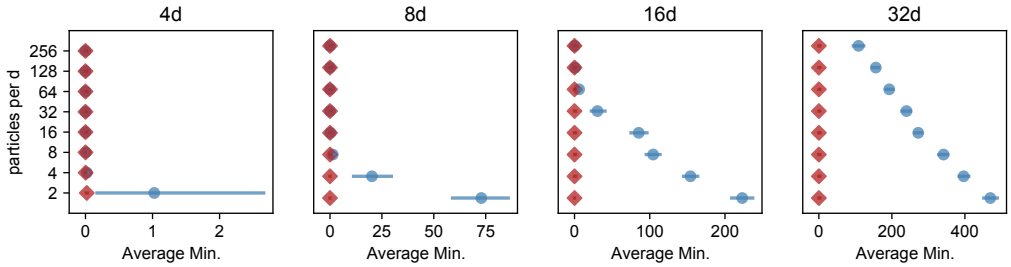


Table B.15: *Schwefel-2.22 Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	4.70e+02 (4.52e+02, 4.89e+02)	**3.14e-04** (0.00e+00, 9.03e-04)
4 (128)	3.97e+02 (3.84e+02, 4.11e+02)	**0.00e+00** (0.00e+00, 0.00e+00)
8 (256)	3.42e+02 (3.29e+02, 3.53e+02)	**0.00e+00** (0.00e+00, 0.00e+00)
16 (512)	2.72e+02 (2.63e+02, 2.81e+02)	**0.00e+00** (0.00e+00, 0.00e+00)
32 (1024)	2.40e+02 (2.28e+02, 2.51e+02)	**0.00e+00** (0.00e+00, 0.00e+00)
64 (2048)	1.93e+02 (1.84e+02, 2.02e+02)	**0.00e+00** (0.00e+00, 0.00e+00)
128 (4096)	1.56e+02 (1.47e+02, 1.64e+02)	**0.00e+00** (0.00e+00, 0.00e+00)
256 (8192)	1.09e+02 (9.45e+01, 1.23e+02)	**0.00e+00** (0.00e+00, 0.00e+00)

Table B.16: *Schwefel-2.23 Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	3.59e+02 (1.53e+02, 6.52e+02)	**8.61e-25** (1.89e-25, 1.92e-24)
4 (128)	2.36e+01 (9.92e+00, 4.46e+01)	**3.03e-29** (1.53e-30, 7.17e-29)
8 (256)	1.45e+00 (6.85e-01, 2.66e+00)	**1.35e-38** (4.47e-42, 3.46e-38)
16 (512)	3.39e-02 (1.52e-02, 5.94e-02)	**0.00e+00** (0.00e+00, 0.00e+00)
32 (1024)	1.60e-03 (1.98e-04, 3.82e-03)	**0.00e+00** (0.00e+00, 0.00e+00)
64 (2048)	1.17e-05 (2.35e-06, 2.65e-05)	**0.00e+00** (0.00e+00, 0.00e+00)
128 (4096)	1.81e-07 (6.14e-08, 3.32e-07)	**0.00e+00** (0.00e+00, 0.00e+00)
256 (8192)	2.73e-09 (6.32e-10, 5.86e-09)	**0.00e+00** (0.00e+00, 0.00e+00)

Figure B.16: *Schwefel 2.23 Benchmark: PSO v. PI-PSO Scaling*

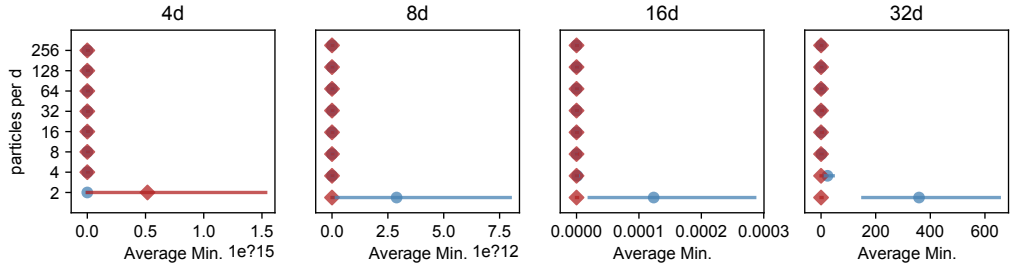


Table B.17: *Sphere Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	2.73e+01 (1.20e+01, 4.20e+01)	0.00e+00 (0.00e+00, 0.00e+00)
4 (128)	8.22e+00 (2.00e+00, 1.80e+01)	0.00e+00 (0.00e+00, 0.00e+00)
8 (256)	5.94e+00 (0.00e+00, 1.40e+01)	0.00e+00 (0.00e+00, 0.00e+00)
16 (512)	1.74e+00 (0.00e+00, 6.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
32 (1024)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
64 (2048)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
128 (4096)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)
256 (8192)	0.00e+00 (0.00e+00, 0.00e+00)	0.00e+00 (0.00e+00, 0.00e+00)

Figure B.17: *Sphere Benchmark: PSO v. PI-PSO Scaling*

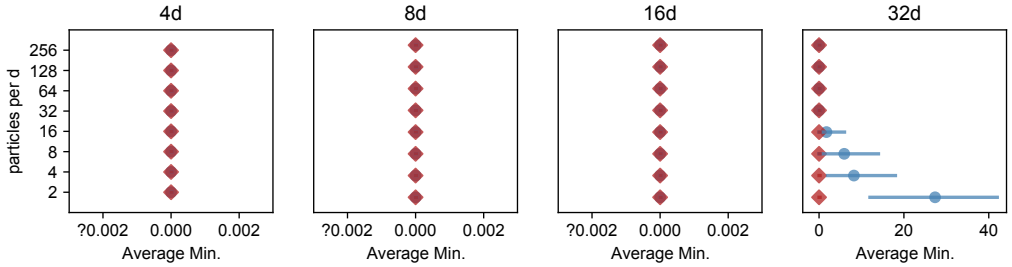


Table B.18: *Stretched-V Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	1.60e+01 (1.52e+01, 1.68e+01)	**3.52e+00** (3.21e+00, 3.83e+00)
4 (128)	1.43e+01 (1.34e+01, 1.51e+01)	**3.31e+00** (3.04e+00, 3.58e+00)
8 (256)	1.32e+01 (1.25e+01, 1.39e+01)	**3.17e+00** (2.84e+00, 3.48e+00)
16 (512)	1.16e+01 (1.11e+01, 1.23e+01)	**3.04e+00** (2.69e+00, 3.32e+00)
32 (1024)	1.10e+01 (1.03e+01, 1.17e+01)	**3.15e+00** (2.85e+00, 3.47e+00)
64 (2048)	1.00e+01 (9.43e+00, 1.06e+01)	**2.71e+00** (2.43e+00, 2.99e+00)
128 (4096)	9.67e+00 (9.07e+00, 1.03e+01)	**2.77e+00** (2.47e+00, 3.14e+00)
256 (8192)	9.20e+00 (8.69e+00, 9.72e+00)	**3.01e+00** (2.67e+00, 3.38e+00)

Figure B.18: *Stretched-V Benchmark: PSO v. PI-PSO Scaling*

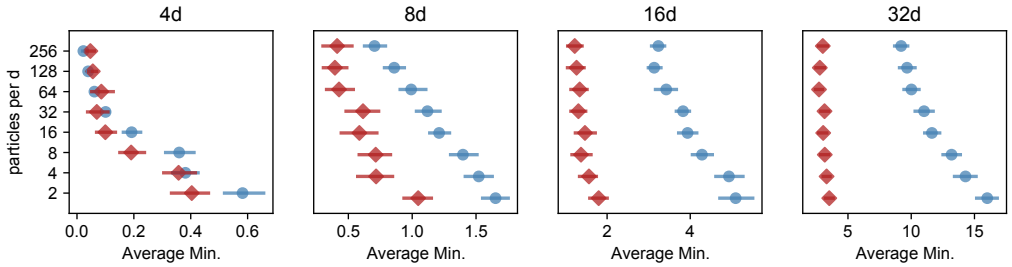


Table B.19: *Whitley Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	5.52e+03 (4.03e+03, 7.26e+03)	**3.59e+01** (1.70e+01, 5.96e+01)
4 (128)	1.36e+03 (1.23e+03, 1.50e+03)	**1.94e+01** (8.74e+00, 3.34e+01)
8 (256)	1.04e+03 (1.02e+03, 1.06e+03)	**2.02e+01** (1.01e+01, 3.58e+01)
16 (512)	9.37e+02 (9.23e+02, 9.50e+02)	**6.26e+01** (2.56e+01, 1.08e+02)
32 (1024)	8.78e+02 (8.66e+02, 8.90e+02)	**2.04e+02** (1.46e+02, 2.59e+02)
64 (2048)	8.07e+02 (7.91e+02, 8.26e+02)	**3.95e+02** (3.57e+02, 4.32e+02)
128 (4096)	7.56e+02 (7.34e+02, 7.72e+02)	**4.29e+02** (3.86e+02, 4.70e+02)
256 (8192)	6.80e+02 (6.65e+02, 6.96e+02)	**3.93e+02** (3.30e+02, 4.47e+02)

Figure B.19: *Whitley Benchmark: PSO v. PI-PSO Scaling*

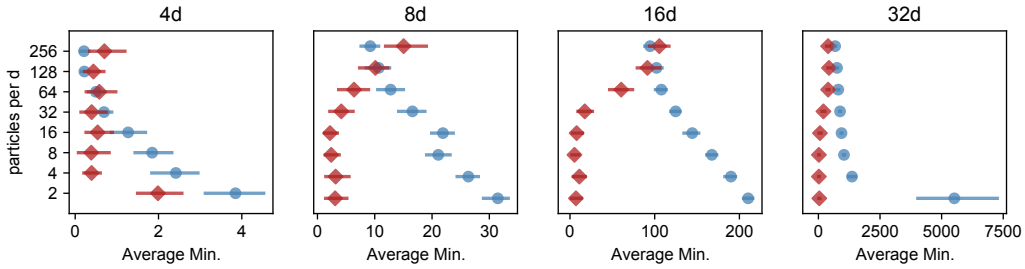


Figure B.20: *Zakharov Benchmark: PSO v. PI-PSO Scaling*

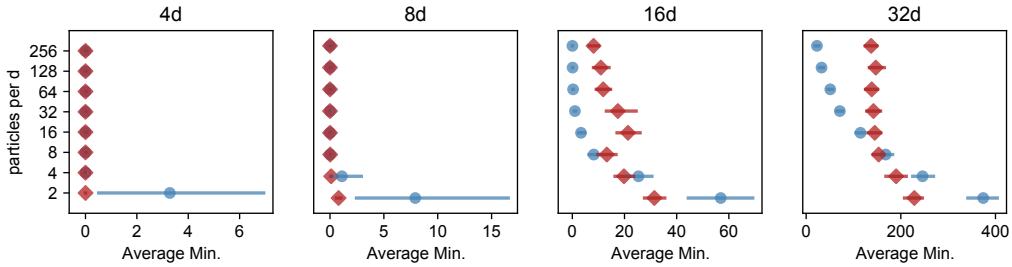


Table B.20: *Zakharov Benchmark 32d Results for Different Particle Counts*

p	PSO	PI-PSO
2 (64)	3.74e+02 (3.42e+02, 4.04e+02)	**2.29e+02** (2.08e+02, 2.45e+02)
4 (128)	2.46e+02 (2.25e+02, 2.69e+02)	**1.90e+02** (1.69e+02, 2.11e+02)
8 (256)	1.68e+02 (1.54e+02, 1.83e+02)	1.53e+02 (1.42e+02, 1.64e+02)
16 (512)	**1.15e+02** (1.05e+02, 1.24e+02)	1.45e+02 (1.31e+02, 1.57e+02)
32 (1024)	**7.12e+01** (6.50e+01, 7.73e+01)	1.42e+02 (1.28e+02, 1.57e+02)
64 (2048)	**5.09e+01** (4.61e+01, 5.55e+01)	1.39e+02 (1.25e+02, 1.51e+02)
128 (4096)	**3.25e+01** (2.81e+01, 3.65e+01)	1.47e+02 (1.34e+02, 1.65e+02)
256 (8192)	**2.29e+01** (1.91e+01, 2.71e+01)	1.37e+02 (1.25e+02, 1.49e+02)

Bibliography

[1] S. J. Russell and P. Norvig, *Artificial intelligence: A modern approach*, 3rd ed. Prentice Hall, 2009.

[2] B. K. Haberman and J. W. Sheppard, "Overlapping particle swarms for energy-efficient routing in sensor networks," *Wireless Networks*, vol. 18, no. 4, pp. 351–363, 2012.

[3] G. Hornby, A. Globus, D. Linden, and J. Lohn, "Automated antenna design with evolutionary algorithms," in *Space 2006*, 2006, p. 7242.

[4] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[5] J. H. Holland, *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[6] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of the IEEE International Conference on Neural Networks*, 1995, pp. 1942–1948.

[7] M. A. Potter and K. A. De Jong, "A cooperative coevolutionary approach to function optimization," in *Parallel Problem Solving from Nature (PPSN III)*. Springer, 1994, pp. 249–257.

[8] F. Van den Bergh and A. P. Engelbrecht, "A cooperative approach to particle swarm optimization," *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 3, pp. 225–239, 2004.

[9] S. Strasser, N. Fortier, J. Sheppard, and R. Goodman, "Factored evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 21, no. 3, pp. 281–293, 2017.

[10] R. Engelmore, *Blackboard Systems*, T. Morgan, Ed. Reading, MA: Addison Wesley, 1988.

[11] V. Pareto *et al.*, "Manual of political economy," 1906.

[12] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *IJCAI*, vol. 3. Stanford Research Institute, 1973, pp. 235–245.

[13] N. Fortier, J. W. Sheppard, and K. Pillai, "DOSI: training artificial neural

networks using overlapping swarm intelligence with local credit assignment," in *Joint 6th International Conference on Soft Computing and Intelligent Systems (SCIS) and 13th International Symposium on Advanced Intelligent Systems (ISIS)*, 2012, pp. 1420–1425.

[14] K. G. Pillai and J. Sheppard, "Overlapping swarm intelligence for training artificial neural networks," in *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, 2011, pp. 1–8.

[15] T. M. Mitchell, "The need for biases in learning generalizations," New Brunswick, New Jersey, USA, Tech. Rep. CBM-TR 5-110, 1980.

[16] J. C. Spall, *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. John Wiley & Sons, 2005.

[17] M. N. Katehakis and J. Arthur F. Veinott, "The multi-armed bandit problem: Decomposition and computation," *Mathematics of Operations Research*, vol. 12, no. 2, pp. 262–268, 1987.

[18] M. Jamil and X.-S. Yang, "A literature survey of benchmark functions for global optimisation problems," *International Journal of Mathematical Modelling and Numerical Optimisation*, vol. 4, no. 2, pp. 150–194, 2013.

[19] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[20] K.-L. Du and M. N. S. Swamy, *Search and Optimization by Metaheuristics: Techniques and Algorithms Inspired by Nature*, 1st ed. Birkhäuser Basel, 2016.

[21] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 6, no. 6, pp. 721–741, Nov. 1984.

[22] J. Brownlee, *Clever Algorithms: Nature-Inspired Programming Recipes*, 1st ed. Lulu.com, 2011.

[23] S. Baluja and R. Caruana, "Removing the genetics from the standard genetic algorithm," Pittsburgh, PA, USA, Tech. Rep., 1995.

[24] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs (3rd Ed.)*. London, UK, UK: Springer-Verlag, 1996.

[25] R. Eberhart, J. Kennedy, and Y. Shi, *Swarm Intelligence*. San Francisco, CA: Morgan Kaufman, 2001.

[26] K. E. Parsopoulos and M. N. Vrahatis, *Particle Swarm Optimization and Intelligence: Advances and Applications*. Hershey, PA: Information Science Reference - Imprint of: IGI Publishing, 2010.

[27] F. J. Solis and R. J.-B. Wets, "Minimization by random search techniques," *Mathematics of operations research*, vol. 6, no. 1, pp. 19–30, 1981.

[28] Y.-C. Ho, "On the numerical solutions of stochastic optimization problem," *IEEE Transactions on Automatic Control*, vol. 42, no. 5, pp. 727–729, 1997.

[29] S. Chen, J. Montgomery, and A. Bolufé-Röhler, "Measuring the curse of dimensionality and its effects on particle swarm optimization and differential evolution," *Applied Intelligence*, vol. 42, no. 3, pp. 514–526, Apr. 2015.

[30] M. Mitchell, S. Forrest, and J. H. Holland, "The royal road for genetic algorithms: Fitness landscapes and GA performance," in *Proceedings of the first European conference on artificial life*. Cambridge: The MIT Press, 1992, pp. 245–254.

[31] S. Strasser, J. Sheppard, and S. Butcher, "A formal approach to deriving factored evolutionary algorithm architectures," in *Proceedings of the IEEE Swarm Intelligence Symposium*, December 2017, pp. 556–563.

[32] S. T. Strasser, "Factored evolutionary algorithms: Cooperative coevolutionary optimization with overlap," Ph.D. dissertation, Montana State University, 2017.

[33] F. Van Den Bergh and A. P. Engelbrecht, "Training product unit networks using cooperative particle swarm optimisers," in *Proceedings of International Joint Conference on Neural Networks*, vol. 1. IEEE, 2001, pp. 126–131.

[34] N. Fortier, J. Sheppard, and S. Strasser, "Learning Bayesian classifiers using overlapping swarm intelligence," in *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, 2015.

[35] N. Fortier, J. Sheppard, and S. Strasser, "Parameter estimation in Bayesian networks using overlapping swarm intelligence," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2015, pp. 9–16.

[36] N. Fortier, J. Sheppard, and S. Strasser, "Abductive inference in Bayesian networks using distributed overlapping swarm intelligence," *Soft Computing*, vol. 19, no. 4, pp. 981–1001, 2014.

[37] N. Fortier, J. Sheppard, and K. G. Pillai, "Bayesian abductive inference using overlapping swarm intelligence," in *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, 2013, pp. 263–270.

[38] K. Veeramachaneni, L. Osadciw, and G. Kamath, "Probabilistically driven particle swarms for optimization of multi valued discrete problems: Design and analysis," in *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, 2007, pp. 141–149.

[39] S. A. Kauffman, *The origins of order: Self-organization and selection in evolution*. Oxford university press, 1993.

[40] T. Jones, "Evolutionary algorithms, fitness landscapes and search," Ph.D. dissertation, University of New Mexico, Department of Computer Science, 1995.

[41] D. Koller and N. Friedman, *Probabilistic Graphical Models - Principles and Techniques.* MIT Press, 2009.

[42] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann, 1988.

[43] R. Storn and K. Price, "Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[44] M. Scutari, "Bayesian network repository," http://www.bnlearn.com/bnrepository, 2012.

[45] S. H. Clearwater, B. A. Huberman, and T. Hogg, "Cooperative problem solving," in *Computation: The Micro and the Macro View*, B. Huberman, Ed. Singapore: World Scientific, 1992, pp. 33–70.

[46] D. D. Corkill, K. Q. Gallagher, and P. M. Johnson, "Achieving flexibility, efficiency, and generality in blackboard architectures." in *AAAI*, 1987, pp. 18–23.

[47] F. Y. Edgeworth, *Mathematical psychics: An essay on the application of mathematics to the moral sciences.* Kegan Paul, 1881, vol. 10.

[48] J. F. Nash Jr, "The bargaining problem," *Econometrica: Journal of the Econometric Society*, pp. 155–162, 1950.

[49] U. Baumgartner, C. Magele, and W. Renhart, "Pareto optimality and particle swarm optimization," *IEEE Transactions on Magnetics*, vol. 40, no. 2, pp. 1172–1175, 2004.

[50] G. Moslehi and M. Mahnam, "A Pareto approach to multi-objective flexible job-shop scheduling problem using particle swarm optimization and local search," *International Journal of Production Economics*, vol. 129, no. 1, pp. 14–22, 2011.

[51] G. Zhang, X. Shao, P. Li, and L. Gao, "An effective hybrid particle swarm optimization algorithm for multi-objective flexible job-shop scheduling problem," *Computers & Industrial Engineering*, vol. 56, no. 4, pp. 1309–1318, 2009.

[52] M. Abido, "Multiobjective particle swarm optimization for environmental/economic dispatch problem," *Electric Power Systems Research*, vol. 79, no. 7, pp. 1105–1113, 2009.

[53] L. Wang and C. Singh, "Environmental/economic power dispatch using a fuzzified multi-objective particle swarm optimization algorithm," *Electric Power Systems Research*, vol. 77, no. 12, pp. 1654–1664, 2007.

[54] T. Cura, "Particle swarm optimization approach to portfolio optimization," *Nonlinear analysis: Real world applications*, vol. 10, no. 4, pp. 2396–2406, 2009.

[55] M. T. Jensen, "Guiding Single-Objective Optimization Using Multi-objective Methods," in *Applications of Evolutionary Computing. Evoworkshops 2003: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, and EvoSTIM.* Essex, UK: Springer. Lecture Notes in Computer Science Vol. 2611, April 2003, pp. 199–210.

[56] F. Neumann and I. Wegener, "Can Single-Objective Optimization Profit from Multiobjective Optimization?" in *Multi-Objective Problem Solving from Nature: From Concepts to Applications.* Berlin: Springer, 2008, pp. 115–130, iSBN 978-3-540-72963-1.

[57] S. Butcher, S. Strasser, J. Hoole, B. Demeo, and J. Sheppard, "Relaxing consensus in distributed factored evolutionary algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO).* ACM, 2016, pp. 5–12.

[58] S. Jurjanovic and D. Bingham, "Optimization test problems," url-https://www.sfu.ca/ ssurjano/optimization.html, 2013.

[59] B. Efron, "Bootstrap methods: Another look at the jackknife," *Ann. Statist.*, vol. 7, no. 1, pp. 1–26, 01 1979.

[60] A. Engelbrecht, "Fitness function evaluations: A fair stopping condition?" in *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*, 2014, pp. 1–8.

[61] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, no. 3, 1965.

[62] G. E. Moore, "Progress in digital integrated electronics [technical literature, copyright 1975 ieee. reprinted, with permission. technical digest. international electron devices meeting, ieee, 1975, pp. 11-13.]," *IEEE Solid-State Circuits Society Newsletter*, vol. 20, no. 3, 2006.

[63] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[64] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[65] G. Agha, "Actors: A model of concurrent computation in distributed systems," Ph.D. dissertation, Massachusetts Institute of Technology, 1986.

[66] J. Armstrong, R. Virding, and M. Williams. (1986) Erlang. http://www.erlang.org/. Accessed: 2018-03-19.

[67] J. Armstrong, "Making reliable distributed systems in the presence of software

errors," Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, 2003.

[68] J. Boner. (2009) Akka. https://akka.io/. Accessed: 2018-03-19.

[69] J. Gosling. (1995) Java. https://java.com/en/. Accessed: 2018-03-19.

[70] M. Odersky. (2004) Scala. https://www.scala-lang.org/. Accessed: 2018-03-19.

[71] K. Quick. (2015) Thespian: Python actor model library. http://thespianpy.com/. Accessed: 2018-03-19.

[72] G. van Rossum. (1991) Python. https://www.python.org/. Accessed: 2018-03-19.

[73] J. De Koster, T. Van Cutsem, and W. De Meuter, "43 years of actors: a taxonomy of actor models and their key properties," in *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. ACM, 2016, pp. 31–40.

[74] A. Kay, "Clarification of "object-oriented"," http://www.purl.org/stefan_ram/pub/doc_kay_oop_en, 2003, acceessed: 2018-03-19.

[75] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 52, 2013.

Vita

Stephyn Geoffrey William Butcher was born on Thanksgiving, November 28, 1963 in Lubbock, Texas. He received a B. A. degree in Economics in 1987 from The California State University, Sacramento; an M. A. degree in Economics in 1998 from The American University, Washington, DC; and an M. S. degree in Computer Science from The Johns Hopkins University in 2006.

He taught macroeconomics at The American University in 1988-1990. He was a grader for Artificial Intelligence at the Johns Hopkins University Engineering for Professionals (JHU EP), 2006-2009. He has taught Machine Learning (Homewood, JHU, 2009), Artificial Intelligence (APL/online, JHU EP, 2009-Present), Reasoning Under Uncertainty (APL, JHU EP, 2011) and Data Science (APL/online, JHU EP, 2013-Present). He won the JHU Engineering for Professionals Excellence in Teaching Award in 2013.

He is a coordinator for JHU Engineering for Professionals Data Science and Cloud Computing track. He is a curriculum committee member for the JHU Engineering

for Professionals' M. S. degree in Data Science. He was a founding member of the Numerical Intelligent Systems Laboratory at The Johns Hopkins University and a member of the reconstituted NISL at Montana State University. Through NISL, he has worked on research grants sponsored by the US Navy.

He has worked as a Software Engineer for the last twenty years at Ntercept, Connections Academy, Mercury Analytics, National Institutes of Health, LivingSocial, Clubhouse Software, Cisco, and, currently, Appriss.

He is a Zen Priest, ordained Myozan Jushin by Abbot Dokai Georgesen of Hokyoji, Eitzen, Minnesota in 2017.

His future plans include programming, teaching, and research and a bit of Zen.