Brigham Young University
BYU ScholarsArchive

Faculty Publications

2011-12-11

A Speculative Approach to Parallelization in Particle Swarm Optimization

Matthew Gardner
Brigham Young University

Andrew McNabb
Brigham Young University

Kevin Seppi
Brigham Young University, kseppi@byu.edu

Follow this and additional works at: https://scholarsarchive.byu.edu/facpub

Part of the Computer Sciences Commons

Original Publication Citation

Gardner, Matthew, Andrew McNabb, and Kevin Seppi. "A speculative approach to parallelization in particle swarm optimization." Swarm Intelligence 6.2 (2012): 77-116.

BYU ScholarsArchive Citation

Gardner, Matthew; McNabb, Andrew; and Seppi, Kevin, "A Speculative Approach to Parallelization in Particle Swarm Optimization" (2011). *Faculty Publications*. 1848.
https://scholarsarchive.byu.edu/facpub/1848

This Peer-Reviewed Article is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Faculty Publications by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.

Noname manuscript No.
(will be inserted by the editor)

A Speculative Approach to Parallelization in Particle Swarm Optimization

Matthew Gardner, Andrew McNabb, and Kevin Seppi

Received: date / Accepted: date

Abstract Particle swarm optimization (PSO) has previously been parallelized primarily by distributing the computation corresponding to particles across multiple processors. In these approaches, the only benefit of additional processors is an increased swarm size. However, in many cases this is not efficient when scaled to very large swarm sizes (on very large clusters). Current methods cannot answer well the question: "How can 1000 processors be fully utilized when 50 or 100 particles is the most efficient swarm size?" In this paper we attempt to answer that question with a speculative approach to the parallelization of PSO that we refer to as SEPSO.

In our approach, we refactor PSO such that the computation needed for iteration $t + 1$ can be done concurrently with the computation needed for iteration t. Thus we can perform two iterations of PSO at once. Even with some amount of wasted computation, we show that this approach to parallelization in PSO often outperforms the standard parallelization of simply adding particles to the swarm. SEPSO produces results that are exactly equivalent to PSO; that is, SEPSO is a new method of parallelization and not a new PSO algorithm or variant.

However, given this new parallelization model we can relax the requirement of exactly reproducing PSO in an attempt to produce better results. We present several such relaxations, including keeping the best speculative position evaluated instead of the one corresponding to the standard behavior of PSO, and speculating several iterations ahead instead of just one. We show that these methods dramatically improve the performance of parallel PSO in many cases, giving speed ups of up to six compared to previous parallelization techniques.

Keywords Parallel algorithms · Optimization methods · Particle swarm optimization · Speculative Decomposition

M. Gardner
Brigham Young University. 3361 TMCB, Provo, UT 84604 Tel.: 801-422-8717 Fax: 801-422-0169 E-mail: mjg82@byu.edu

A. McNabb
Brigham Young University. 3361 TMCB, Provo, UT 84604 E-mail: a@cs.byu.edu

K. Seppi
Brigham Young University. 3361 TMCB, Provo, UT 84604 E-mail: k@cs.byu.edu

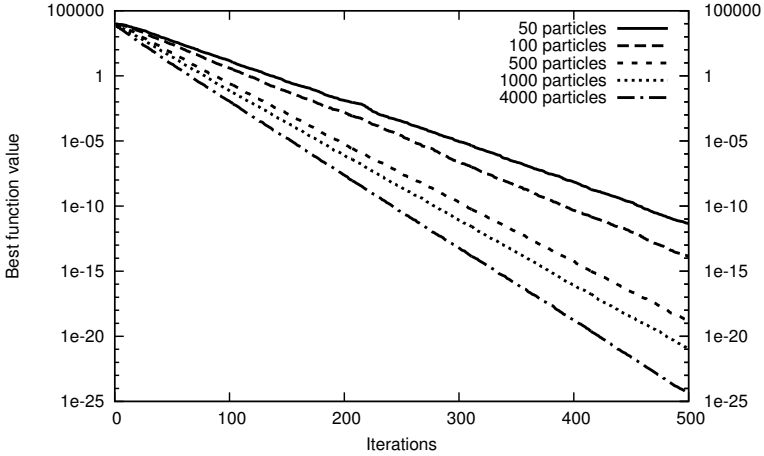

Fig. 1 Function Sphere with various swarm sizes, comparing performance with the number of iterations of the algorithm performed.

1 Introduction

Particle swarm optimization (PSO) has been found to be a highly robust and effective algorithm for solving many types of optimization problems (Poli, 2008a; Poli et al., 2007). For much of the algorithm's history, PSO was run serially on a single machine. However, the world's computing power is increasingly coming from large clusters of processors. In order to efficiently utilize these resources for computationally intensive problems, PSO needs to run in parallel.

Within the last few years, researchers have begun to recognize the need to develop parallel implementations of PSO, publishing many papers on the subject. The methods they have used include various synchronous algorithms (Parsopoulos et al., 2004) and asynchronous algorithms (Mostaghim et al., 2006). Parallelizing the evaluation of the objective function can also be done in some cases using standard techniques (Grama et al., 2003), though that is not an adaption of the PSO algorithm itself and thus is not the focus of this paper.

These previous parallel techniques distribute the computation needed by the particles in the swarm over the available processors. If more processors are available, these techniques increase the number of particles in the swarm, either by adding individual particles or by adding entire new sub-swarms. In almost all cases, adding additional particles produces better results in the same amount of time (McNabb et al., 2009). In Figure 1 we see an example of this on the well-known benchmark function Sphere (20 dimensions, reporting the average of twenty runs). In terms of the number of iterations performed (which is equivalent to wall-clock time if all particles are evaluated in parallel), every time the swarm size increases, the performance improves.

However, it can be seen from the graph that once the swarm is sufficiently large, there comes a point of diminishing returns with respect to adding particles. The increase in performance seen when moving from 50 to 100 particles is roughly equivalent to the increase seen when moving from 1000 to 4000. In Figure 2 we

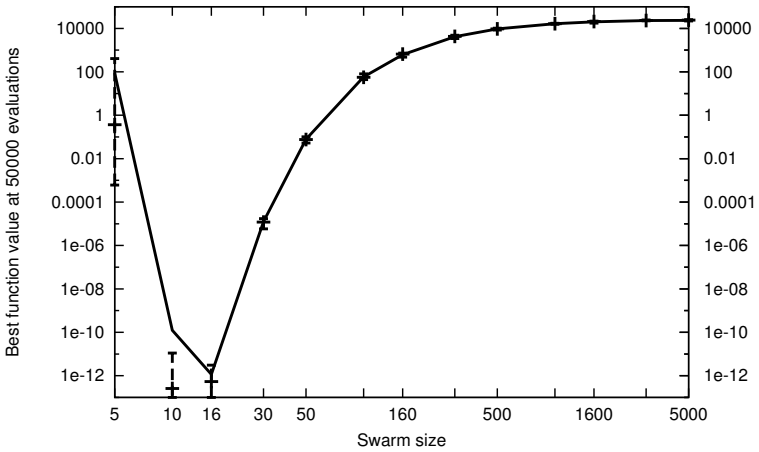

Fig. 2 Function Sphere with various swarm sizes, showing performance after a set number of function evaluations. Error bars show mean and 10th and 90th percentiles.

show the value obtained after 50,000 function evaluations (not iterations) as a function of swarm size, again for the function Sphere. Increasing the swarm size from 5 to 10 has a significant effect on the value obtained. However, increasing the swarm size from 16 to 30 makes the algorithm less efficient; that is, it reduces the progress the algorithm makes per evaluation. Other functions show similar trends, though often the optimal swarm size is slightly larger. For this reason, previous work has recommended the use of a swarm size of 50 for PSO (Bratton & Kennedy, 2007). Thus, in at least some cases, adding particles indefinitely will not yield an efficient implementation.

Our purpose is to explore the question of what to do with a thousand processors when 50 or 100 particles is the most efficient swarm size, and simply adding particles results in only incremental improvement. We thus consider PSO parallelization strategies for clusters of hundreds or thousands of processors and functions for which a single evaluation will take long enough to merit a parallelization of one particle per processor—at least hundreds of milliseconds, but perhaps several minutes or longer.

In order to solve the problem of diminishing returns, we apply the concept of speculative decomposition (Grama et al., 2003) to particle swarm optimization, using extra processors to perform two iterations of PSO at the same time. Speculative decomposition is analogous to speculative execution (also known as branch prediction), a technique commonly used in processors. Modern processors, when faced with a branch on which they must wait (e.g., a memory cache miss), guess which way the branch will go and start executing, ensuring that any changes can be undone. If the processor guesses right, execution is much farther ahead than if it had idly waited on the memory reference. If it guesses wrong, execution restarts where it would have been anyway. Thus the processor speculates about future paths of execution in an attempt to decrease overall processing time.

In this paper we show that the results of standard PSO can be reproduced *exactly*[1], two iterations at a time, using a speculative approach adapted from speculative execution. We show that the standard PSO equations can be factored such that a set of speculative positions can be found which will always include the position computed in the next iteration. By computing the value of the objective function for each of the speculative positions at the same time the algorithm evaluates the objective function for the current position, it is possible to know the objective function values for both the current and the next iteration at the same time. We demonstrate this principle by implementation and show that it produces exactly the same results as standard PSO, but two iterations at a time. The resulting implementation runs efficiently on large clusters where the number of processors is much larger than a typical or reasonable number of particles, producing better results in less wall-clock time.

We refer to this parallelization technique as "speculative evaluation in particle swarm optimization", or SEPSO. It is important to note here that SEPSO is not a variant of PSO. We simply propose a new way to think about the parallelization of PSO that we show takes fewer iterations to reach the same level of fitness than previous parallelizations in many instances.

Furthermore, we show that if we relax the requirements of the algorithm, no longer demanding that it strictly reproduce the exact behavior of standard PSO, we can introduce new speculative techniques that often out-perform both standard parallelizations of PSO and SEPSO. These relaxations make better use of the information obtained from the extra exploration made by the speculative function evaluations. We also explore the idea that, like branch prediction in processors, we need not speculatively evaluate *all* possible future positions, we can accelerate the algorithm even if we are just *likely* to have guessed right. By pruning the speculation to just paths that are statistically likely to reproduce the paths that are equivalent to PSO we can increase the swarm size without increasing the number of speculative evaluations. We also consider several recovery strategies for cases where the pruned set of speculative evaluations does not contain the evaluation that standard PSO would have done. A further improvement we explore is speculating several iterations ahead instead of just one, which is made possible by pruning the number of speculative evaluations.

This paper is organized as follows. Section 2 describes the particle swarm optimization algorithm, and Section 3 gives a brief overview of previous parallelization techniques for this algorithm. Section 4 shows mathematically how speculative evaluation can be done in parallel PSO to perform two iterations at once, leaving implementation concerns to the appendices. In Section 5, we discuss various methods of improving the performance of speculative evaluation in PSO, all of which break the requirement of strictly reproducing the behavior of the original algorithm. Section 6 describes the experiments we ran, and Section 7 presents the results of those experiments. In Section 8 and Section 9 we conclude and discuss future work.

[1] In fact it is only because the results are exactly the same that we are confident of our implementation. With the careful use of random seeds we were able to detect errors in our speculative implementation when particle positions were off in the tenth digit.

2 Particle Swarm Optimization

Particle swarm optimization was proposed in 1995 by James Kennedy and Russell Eberhart (Kennedy & Eberhart, 1995). The algorithm is used to intelligently search a multi-dimensional space in a way that is inspired by the swarming and flocking behavior of birds and other animals. It is a social algorithm that depends on interaction between particles to quickly and consistently approximate the optimal solution to a given objective function.

The motion of particles through the search space has three components: an inertial component that gives particles momentum as they move, a cognitive component where particles remember the best solution they have found and are attracted back to that place, and a social component by which particles are attracted to the best solution that any of their neighbors have found. We use the constricted PSO algorithm, which was recommended as the standard for PSO research (Bratton & Kennedy, 2007).

At each iteration of constricted PSO (Clerc & Kennedy, 2002), the position $\mathbf{x}_t$ and velocity $\mathbf{v}_t$ of each particle are updated as follows:

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

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}_{t+1} \tag{2}$$

where $\mathbf{U}_t^P$ and $\mathbf{U}_t^N$ are vectors of independent random numbers drawn from a standard uniform distribution, the $\otimes$ operator is an element-wise vector multiplication, $\mathbf{b}^P$ (called personal best) is the best position the current particle has seen, and $\mathbf{b}^N$ (called neighborhood best) is the best position the neighbors of the current particle have seen (Bratton & Kennedy, 2007). The parameters ϕ^N, ϕ^P, and χ are given prescribed values required to ensure convergence (2.05, 2.05, and .73, respectively) (Clerc & Kennedy, 2002).

Changing the way neighbors are defined, usually called the "topology," has a significant effect on the performance of the algorithm. In the Ring topology, each particle has one neighbor to either side of it; in the Complete topology[2], every particle is a neighbor to every other particle (Bratton & Kennedy, 2007). In all topologies a particle is also a neighbor to itself in that its own position and value are considered when updating the particle's neighborhood best, $\mathbf{b}^N$. Thus with p particles, using the Ring topology each particle with index i has three neighbors: $i-1$, i (itself), and $i+1$. With the Complete topology, each particle has p neighbors.

In this paper we use these topologies as well as a parallel adaptation of the Complete topology, called Random, that has been shown to approximate the behavior of Complete with far less communication (McNabb et al., 2009). In the Random topology, each particle randomly picks two other particles to share information with at each iteration, along with itself. Thus in both the Ring and the Random topologies, all particles have three neighbors.

The ideal topology and swarm size for PSO depend on the objective function. Researchers have devised various benchmark functions and have found that the ideal topology for one function may perform very poorly for another function. The

[2] The Complete topology has often been unfortunately named Star in the literature, which in graph theory refers to a completely different topology. Other names have also been used, including "global topology" and gbest. We use the graph theory term "Complete" in this paper.

No Free Lunch Theorems for Optimization show that this is true in general—if an algorithm performs well on average for one class of functions then it must do poorly on average for other problems (Wolpert & Macready, 1997). A good starting point seems to be a swarm of 50 particles using a Ring topology (Bratton & Kennedy, 2007).

3 Related Work

The idea of speculative decomposition in the parallelization literature is not new (Grama et al., 2003). In the field of function optimization, simulated annealing has previously been parallelized using this technique (Witte et al., 1991), though we are not aware of other evolutionary or swarm intelligence based algorithms having been parallelized with speculative decomposition.

There have been several parallelizations of PSO presented in the literature. The improvements described in these papers come in two major areas: innovations in implementation details and innovations in the use of topology and swarm size to scale PSO to many processors.

3.1 Innovative Implementations

There are many ways to parallelize the basic PSO algorithm. The most fundamental decision to make in parallel PSO is which parallel architecture to use. Several architectures have been proposed, including Master-Slave, fully distributed (sometimes called "diffusion"), and reformulating PSO into Google's MapReduce framework (Belal & El-Ghazawi, 2004; McNabb et al., 2007). For inexpensive objective functions, graphics processors (GPUs) have also been used to parallelize PSO using either texture rendering (Li et al., 2007) or a general purpose GPU API (Zhou & Tan, 2009).

An important part of the parallel architecture is the type of interprocessor communication used. PSO has previously been parallelized with both synchronous and asynchronous communication. Synchronous parallel implementations of PSO reproduce the standard serial algorithm exactly. This approach was first described analytically by Belal & El-Ghazawi (2004) and first implemented by Schutte et al. (2004). In a typical master-slave algorithm, the master assigns tasks to slave processors, and in parallel PSO, each task consists primarily of a function evaluation. Updating the particle's position and value may also be included in the task (Belal & El-Ghazawi, 2004), or this work may be performed in serial by the master (Schutte et al., 2004). Before proceeding to the next iteration, particles communicate, and each particle updates its neighborhood best. Whether this communication step happens sequentially on the master or in parallel, each particle must receive communication from its neighbors before proceeding. The benefits of the synchronous PSO include its simplicity, repeatability, and comparability with standard PSO, which may be essential in research applications.

Asynchronous parallel particle swarm optimization (Venter & Sobieszczanski-Sobieski, 2005; Koh et al., 2006) is a modification to the standard algorithm which removes the synchronization point at the end of each iteration. Instead, particles iterate independently and communicate asynchronously. In a typical master-slave

implementation of asynchronous parallel PSO, the master updates each particle's personal best, neighborhood best, velocity, and position immediately after receiving the function value from the slave processor. Since this update occurs while other particles are still being evaluated, it may use information from the previous iteration for some neighbors.[3] In a partially asynchronous implementation, particles might wait for some but not all neighbors to complete before proceeding (Scriven et al., 2008a). In some master-slave implementations, particles never get more than one iteration ahead of others (Venter & Sobieszczanski-Sobieski, 2005; Koh et al., 2006). However, in a fully distributed implementation, particles might never wait for information, and one particle could complete many more iterations than another particle (Scriven et al., 2008b). The main effect of asynchronous evaluation is that processors spend less time idle—this trait is particularly valuable when processors are heterogeneous or function evaluation times are varied (Venter & Sobieszczanski-Sobieski, 2005; Koh et al., 2006). Asynchronous parallel PSO behaves differently than the standard algorithm and may even produce different results between runs. Most reports conclude that asynchronous communication produces similar numerical results to the standard algorithm, but the question has not yet been thoroughly addressed (Venter & Sobieszczanski-Sobieski, 2005; Koh et al., 2006).

3.2 Scaling PSO to many processors

The other area of research in parallelizing PSO deals not with the implementation details of architecture and synchronicity, but with what should be done with the PSO equations when many hundreds or thousands of processors are available. The main issues that have been addressed are how many particles to use for a particular number of processors and what communication topology should be employed.

The number of particles per processor has typically been decided by how long it takes to evaluate the function being optimized. When the function takes longer than a few seconds to evaluate, previous techniques have assigned the number of particles in the swarm to be the number of processors available (Jin & Rahmat-Samii, 2005; McNabb et al., 2009), advocating using as many processors as possible to get the best performance. When the function takes far less time to evaluate than it takes to send a message across a network (e.g., through the TCP/IP stack), parallel implementations assign several or many particles to a single processor (Chu & Pan, 2006; Chang et al., 2005). Often the processor only sends information about the best particle it evaluated to other processors (Belal & El-Ghazawi, 2004).

Another popular method is simply to run PSO independently on each of the processors available, taking the best result when all of the runs complete. It should be noted that this is equivalent to the previously stated method of assigning many particles to each processor, only with no communication between processors instead of little communication. Both of these methods can be described as changes in the communication topology of the original PSO algorithm (McNabb et al., 2009).

[3] Asynchronous parallel PSO has been compared to the "asynchronous updates" variant of serial PSO (Koh et al., 2006). However, serial PSO with asynchronous updates differs from standard PSO in that particles use newer information, but asynchronous parallel PSO differs from standard PSO in that particles use older information.

Thus previous work in parallelizing PSO, apart from creating innovative implementations, has consisted entirely of increasing the swarm size and adapting the topology to be better suited to parallel computation.

With regard to increasing the swarm size in PSO, some recent work has suggested that increasing the swarm size throughout the course of the optimization process provides better results than having a set swarm size (Hsieh et al., 2009; Montes de Oca et al., 2010). However, these results focused on serial computation and are based on total number of function evaluations, which, when running in parallel on expensive functions, is less important than total number of iterations. Other work focusing on parallelization has shown that when extra processors are available they should be used, as performance increases with swarm size when measuring in terms of number of iterations (McNabb et al., 2009; Jin & Rahmat-Samii, 2005). If the swarm size were varied throughout the course of the optimization process, some processors would be sitting idle at most iterations.

The contribution of our work lies in the realm of efficiently utilizing parallel resources. In our work we use a synchronous, MapReduce implementation of parallel PSO. While we use a specific implementation, we describe how speculative evaluation can be performed in any of the synchronous architectures mentioned in the previous section. The adaptation of our methods to asynchronous PSO parallelization methods should be straightforward, though it is left to future work.

4 Speculative Evaluation in PSO

PSO can be trivially parallelized by assigning each particle's computation to an individual processor. But as we have seen in Figure 2, for some functions, and for large numbers of processors, just adding particles reaches a point of diminishing returns. That is, beyond some point adding processors with previous techniques does not help the algorithm reach any given level of fitness significantly faster. To fix this, instead of adding particles we employ extra processors in a speculative approach that allows us to perform two iterations at a time.

Our speculative methods require refactoring the PSO equations such that all possible positions for each particle at iteration $t + 1$ can be evaluated in parallel along with the position of each particle at iteration t. With some careful bookkeeping, we can then piece together the results of iteration $t + 1$ for each particle, thus using extra processors to evaluate two iterations of the algorithm in the time it takes to evaluate the function once. As we will show in Sections 4.1 and 4.2, a wise choice of topology limits the necessary speculative evaluations to seven per particle.

To see the value of this refactoring, suppose that 1000 processors are available, and that the evaluation of the objective function takes one hour. If we only want a swarm of 100 particles, 900 of the processors would be sitting idle for an hour at every iteration, and it would take two hours to run two iterations. If instead we perform speculative evaluation, sending each of the 7 possible speculative positions of a particle to be computed at the same time as its current position, we would use 800 of the 1000 processors and perform two iterations in one hour.

In order to do two iterations at once, we must use 8 times as many processors as there are particles in the swarm. If these processors were not performing speculative evaluation, they might instead be used for function evaluations needed to

Table 1 All possible updates for a particle with two neighbors

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

support a larger swarm. This raises the question of whether a swarm of 100 particles doing twice as many iterations outperforms a swarm of 800 particles. We show in Section 7 that in many, though not all, instances, a smaller swarm performing more iterations does in fact outperform a larger swarm.

Section 4.1 shows in detail how the PSO equations can be refactored to allow for speculative evaluation, proving that SEPSO exactly reproduces the behavior of the PSO algorithm. The section also introduces some notation used later in the paper. Section 4.2 gives a brief discussion of how the topology used affects the amount of speculative computation needed.

4.1 Refactoring the PSO Equations

To perform two iterations at a time we must first refactor PSO such that the determination of the value of the objective function is separate from the rest of the computation. For simplicity, this discussion will describe the case where PSO is performing function minimization using the Ring topology. In this example, each particle has two neighbors, the "right neighbor" and "left neighbor," whose positions are represented as $\mathbf{x}^R$ and $\mathbf{x}^L$ respectively. Though we will only describe the case of the Ring topology here, the math is straightforward for other topologies. Our discussion of the implementation in Section 9 is independent of specific topologies, and we use several different topologies in our experiments.

The refactoring hinges on the idea that there are only a few possible new positions, or updates, for $\mathbf{b}^P$ and $\mathbf{b}^N$ (assuming the random coefficients $\mathbf{U}_t^P$ and $\mathbf{U}_t^N$ have been drawn). For the Ring topology there are 7 possible update cases, identified in Table 1. We label each case with an identifier referring to the source of the update: a minus sign $(-)$ represents no update, L represents an update to $\mathbf{b}^N$ coming from the left neighbor, R represents an update to $\mathbf{b}^N$ coming from the right neighbor, and S represents an update to either $\mathbf{b}^P$ or $\mathbf{b}^N$ coming from the particle itself. As an example, $(S,-)$ refers to the case that the particle finds a new personal best, but neither it nor its neighbors find a position that updated its neighborhood best. In the equations that follow, we refer to an update case as c, and to the set of cases collectively as $\mathcal{C}$.

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

(which, as is shown in Table 1, means that the particle's personal best was updated, but its neighborhood best was not) can be written as follows:

$$I_{t+1}^{(S,-)}(f(\mathbf{x}_t), f(\mathbf{x}_t^L), f(\mathbf{x}_t^R), f(\mathbf{b}_{t-1}^P), f(\mathbf{b}_{t-1}^N)) = \begin{cases} 1 & \text{if } f(\mathbf{x}_t) < f(\mathbf{b}_{t-1}^P) \\ & \quad \text{and} \quad f(\mathbf{b}_{t-1}^N) < f(\mathbf{x}_t) \\ & \quad \text{and} \quad f(\mathbf{b}_{t-1}^N) < f(\mathbf{x}_t^L) \\ & \quad \text{and} \quad f(\mathbf{b}_{t-1}^N) < f(\mathbf{x}_t^R) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

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

$$\begin{aligned} &\mathbf{V}_{t+1}^{(S,-)}(\mathbf{v}_t, \mathbf{x}_t, \mathbf{x}_t^L, \mathbf{x}_t^R, \mathbf{b}_{t-1}^P, \mathbf{b}_{t-1}^N, \mathbf{U}_t^P, \mathbf{U}_t^N) \\ &\quad = \chi\big[\mathbf{v}_t + \phi^P \mathbf{U}_t^P \otimes (\mathbf{x}_t - \mathbf{x}_t) + \phi^N \mathbf{U}_t^N \otimes (\mathbf{b}_{t-1}^N - \mathbf{x}_t)\big] \end{aligned} \tag{4}$$

In the same way we can create notation for the position update function by substituting into (2). For compactness, we will drop the parameters to $\mathbf{V}_{t+1}^c$ since they can be inferred from the subscripts.

$$\mathbf{X}_{t+1}^c(\mathbf{x}_t, \mathbf{v}_t, \mathbf{x}_t^L, \mathbf{x}_t^R, \mathbf{b}_{t-1}^P, \mathbf{b}_{t-1}^N, \mathbf{U}_t^P, \mathbf{U}_t^N) = \mathbf{x}_t + \mathbf{V}_{t+1}^c \tag{5}$$

With this notation we can re-write the original PSO velocity equation (1), introducing our sum over cases with the indicator functions. Again, we represent the indicator functions and velocity functions without the parameters for compactness. The equation becomes:

$$\begin{aligned} \mathbf{v}_{t+1} &= \chi\big[\mathbf{v}_t + \phi^P \mathbf{U}_t^P \otimes (\mathbf{b}_t^P - \mathbf{x}_t) + \phi^N \mathbf{U}_t^N \otimes (\mathbf{b}_t^N - \mathbf{x}_t)\big] \\ &= \sum_{c \in \mathcal{C}} I_{t+1}^c \; \chi\big[\mathbf{v}_t + \phi^P \mathbf{U}_t^P \otimes (\mathbf{b}_t^P - \mathbf{x}_t) + \phi^N \mathbf{U}_t^N \otimes (\mathbf{b}_t^N - \mathbf{x}_t)\big] \\ &= \sum_{c \in \mathcal{C}} I_{t+1}^c \; \mathbf{V}_{t+1}^c \end{aligned} \tag{6}$$

Similarly, the position update equation (2) becomes:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}_{t+1} = \sum_{c \in \mathcal{C}} I_{t+1}^c \; \mathbf{X}_{t+1}^c \tag{7}$$

The value of the objective function at $\mathbf{x}_{t+1}$ is given by:

$$f(\mathbf{x}_{t+1}) = \sum_{c \in \mathcal{C}} I_{t+1}^c \; f(\mathbf{X}_{t+1}^c) \tag{8}$$

Returning our attention to the computation of $\mathbf{x}_{t+1}$ in (7) and writing it with the parameters which were omitted above, we obtain:

$$\mathbf{x}_{t+1} = \sum_{c \in \mathcal{C}} I^c_{t+1}(f(\mathbf{x}_t), f(\mathbf{x}^L_t), f(\mathbf{x}^R_t), f(\mathbf{b}^P_{t-1}), f(\mathbf{b}^N_{t-1}))$$
$$\mathbf{X}^c_{t+1}(\mathbf{x}_t, \mathbf{v}_t, \mathbf{x}^L_t, \mathbf{x}^R_t, \mathbf{b}^P_{t-1}, \mathbf{b}^N_{t-1}, \mathbf{U}^P_t, \mathbf{U}^N_t) \tag{9}$$

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

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

4.2 Topology in Speculative Evaluation

The number of speculative evaluations needed per particle depends on the number of neighbors each particle has. The number of update cases in a topology where each particle has n neighbors is $2(n+1)$; there are two possibilities for updates to $\mathbf{b}^P$ (updated by the particle itself and not updated), and $n+1$ possibilities for updates to $\mathbf{b}^N$ (updated by each neighboring particle and not updated). When the particle is also a neighbor to itself, as is always the case in commonly used topologies, one of the cases can be eliminated, as a particle cannot be the source of an update to its neighborhood best without also updating its personal best. Thus we have $2(n+1)-1$, or $2n+1$, speculative evaluations per particle. In a swarm with p particles and n neighbors per particle, $(2n+1)p$ speculative evaluations are needed.

Because the number of speculative evaluations depends on the number of neighbors a particle has, the choice of topology is an important one. The use of the Complete topology, where every particle is a neighbor to every other particle, would require $O(p^2)$ speculative evaluations per iteration. Clearly it is much more desirable to have a sparse topology, where $O(np)$ is much smaller than $O(p^2)$. However, some functions are better optimized with the Complete topology and the quick spread of information it entails than with sparse topologies. Accordingly, we use the Random topology described in (McNabb et al., 2009), which has been shown to approximate the Complete topology. In Section 7 we report results for SEPSO using both the Ring topology and the Random topology on a number of common benchmark functions.

5 Speculative Variations on PSO

Refactoring the PSO equations led us to find that speculative approaches are possible in the parallelization of PSO. SEPSO reproduces standard PSO exactly, two iterations at a time, at the expense of requiring several times the number of processors. In this section we consider other speculative techniques inspired by SEPSO that relax the requirement of exactly reproducing the behavior of the original PSO algorithm.

We outline three main improvements to speculative evaluation. First, in Section 5.1 we describe a method that uses all of the information found in doing speculative evaluations. Then Section 5.2 presents a technique that reduces the number of speculative evaluations that need to be done for each particle, allowing speculative evaluation to use larger swarm sizes with the same number of processors. Finally, Section 5.3 shows a method for speculating several iterations ahead, instead of just one.

None of these methods fundamentally change the PSO algorithm. They simply lead to particles being at different iterations and having different values for personal and neighborhood best positions than would have occurred in standard PSO, because they receive different information. These kinds of relaxations are fairly typical in the parallelization of PSO (Koh et al., 2006).

5.1 Pick the Best Child

In performing speculative evaluation as we have described it, $2n + 1$ speculative evaluations are done per particle, while all but one of them are completely ignored. It seems reasonable to try to make use of the information obtained through those evaluations instead of ignoring it.

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

This can be thought of as drawing a number of samples from the next iteration and accepting the best one. Speculative particles that move in good directions are kept. Intuition says that this technique favors exploitation over exploration, but as we will show in Section 7, that is not always the case.

At this point it is also interesting to note a parallel between our methods and parallel evolution strategies (Rudolph, 1991). In evolution strategies, a parent individual (representing a potential solution to some objective function) produces a number of offspring by a mutation operator. One of the individuals is selected by a selection operator, and that individual becomes the parent for the next generation (Beyer & Schwefel, 2002). Our methods are similar, where our mutation operator is simply the PSO motion equations and the selection operator is either the indicator function introduced in Section 4, in the case of our original speculative algorithm, or the standard selection operator based on fitness, in the case of this Pick Best technique.

5.2 Pruning the Speculative Evaluations

Because the SEPSO requires so many extra evaluations, a natural step to take is to eliminate some of them. If we could reliably predict which branch were going to be taken, we could limit ourselves to one speculative evaluation per particle instead of $2n+1$. With a fixed number of processors, this would allow us to greatly increase the swarm size relative to that needed in the original speculative algorithm (e.g., with 120 processors, a standard parallelization has a swarm of size 120, complete speculative evaluation has a swarm of size 15, and pruning the evaluations to only one per particle allows a swarm of size 60). As not all of the branches are evaluated in any given iteration, we call this technique pruning.

We look at the statistical behavior of PSO to find probabilities of taking any particular branch. While we cannot with certainty predict which branch a particle will take every time, if we can use statistics to narrow down the $2n+1$ possible evaluations to a few likely candidates, we can decrease the amount of computation required to do speculative evaluation and improve our performance.

5.2.1 Branch Statistics

In Table 1 we presented all possible branches that a particle with two neighbors could take. Here we lump all of the neighbors together and consider the statistics for the five branches shown in Table 2. In the identifiers, N represents an update to $\mathbf{b}^N$ coming from any neighbor.

Table 2 Five Branches to Consider for Statistics

Identifier	Source of $\mathbf{b}^P$ update	Source of $\mathbf{b}^N$ update
$(-,-)$	No update	No update
$(S,-)$	Self	No update
(S,S)	Self	Self
$(-,N)$	No update	Some Neighbor
(S,N)	Self	Some Neighbor

Table 3 Branch Statistics in PSO

Topology	Function	$(-,-)$	$(S,-)$	(S,S)	$(-,N)$	(S,N)
Ring	Sphere	53.0%	9.3%	11.4%	20.2%	6.2%
	Griewank	51.7%	8.4%	12.2%	20.7%	7.0%
	Rastrigin	49.5%	4.8%	14.6%	21.3%	9.9%
	Rosenbrock	51.3%	7.4%	12.9%	21.1%	7.3%
	Average	51.3%	7.5%	12.8%	20.8%	7.6%
Random	Sphere	66.7%	11.9%	2.6%	15.6%	3.1%
	Griewank	69.0%	10.9%	2.5%	14.9%	2.7%
	Rastrigin	81.9%	5.5%	1.5%	10.0%	1.0%
	Rosenbrock	74.2%	7.7%	2.2%	14.0%	1.8%
	Average	73.0%	9.0%	2.2%	13.6%	2.2%
Complete	Sphere	31.9%	9.2%	0.2%	45.1%	13.5%
	Griewank	35.3%	8.4%	0.2%	44.1%	11.9%
	Rastrigin	47.7%	6.7%	0.2%	38.2%	7.0%
	Rosenbrock	35.3%	3.4%	0.3%	54.4%	6.6%
	Average	37.6%	6.9%	0.2%	45.5%	9.8%

We seek to find the probability of taking any given branch, given whatever information is needed: $\Pr(\mathcal{C}_t|\cdot)$. In finding these probabilities, we do not attempt to derive any distribution from the PSO equations, we simply look at empirical distributions. However, even with empirical distributions, the problem with this approach is that it is not clear what information influences the probability of taking a branch. We look at two factors that we believe have a significant influence on $\Pr(\mathcal{C}_t)$: topology (T) and function (F). Thus we are looking at $\Pr(\mathcal{C}_t|T, F)$.

We show in Table 3 with what percentage a particle takes each of these branches for three different topologies and four different functions. All of our statistics are from swarms of 240 particles. Brief experimentation showed that other swarm sizes had similar statistics. We ran 750 iterations on all combinations of functions and topologies except for the functions Griewank and Rastrigin with the Complete topology. We found that those runs frequently converged past machine precision after 500 iterations, and that led to erroneously high values for the probability of $(-, -)$. Instead we ran for only 450 iterations on those two combinations. All of our results were averaged over 20 runs of the algorithm; thus the probabilities presented are the averages of 3.6 million trials for the branch taken (2.16 million for the two with only 450 iterations). Table 3 contains the results. The definitions for all of the functions in the table are found in Section 6.

The probabilities presented in Table 3 are interesting in and of themselves and could probably be used to better understand the characteristics of various topologies. It is notable that there is small variation between functions in any given topology, but the variation across topologies is far greater. However, our concern is with speculative evaluation. We are interested in predicting the branch that any given particle will take at a particular iteration. For our purposes, it appears that given a topology, the probability of selecting a branch and the function are close to independent, or $\Pr(\mathcal{C}_t|T, F) \approx \Pr(\mathcal{C}_t|T)$.

From Table 3 we can see that with the Random topology, we can pick the first branch, corresponding to stagnation, and be right around 70% of the time. With the Ring topology, we would be right 50% of the time. Branches $(-, N)$ and (S, N) really correspond to several actual branches, as all of the neighbors are lumped together. The 20% probability of taking branch $(-, N)$ with the Ring topology can be split into two branches, as there are only two neighbors. It also turns out that the neighbor that last updated the neighborhood best is the most likely to update it next time, so keeping track of that information could be fruitful in trying to predict that branch.

The statistics for the Complete topology are less promising, as there are 240 neighbors that branch $(-, N)$ splits into, instead of two. Pruning does, however, allow for the possibility of using the Complete topology in speculative evaluation while avoiding the explosion in the number of evaluations it would otherwise entail.

Because there are a few branches with very high probabilities in the topologies we are interested in, we can have hope that cutting out some of the evaluations that have low probability will lead to an increase in overall performance. In order to implement this kind of pruning, the only change that needs to be made to Algorithm 1 is in Step 2, where the speculative children are generated. Instead of generating all possible speculative children, generate the subset of the children that is desired.

5.2.2 Recovering from Pruning Too Much

When not all of the branches are evaluated, there is some probability of not evaluating the branch that was actually taken by the original particle. There are a few possibilities for recovery in this case. One is to leave the particle as it is, not accepting any of the speculative evaluations, because none of them were correct. This leads to particles being at different iterations, as some particles guess correctly while others do not. Thus we lose exact compatibility with the original PSO, though this particular relaxation is nothing new; asynchronous adaptations of PSO do the same thing (Koh et al., 2006). As an aside, it is equivalent in this case to simply increment the iteration number of particles which fail to correctly predict their branch. This keeps the iteration number constant across all particles, simplifying the work needed to be done in determining neighbors when dynamic topologies are involved. We call this technique Social Promotion.

Another possibility is to pick the best child, as described in Section 5.1. This ignores the fact that the branch might have been wrong; it does not matter, because we simply accept the child that had the best value.

5.3 More than one iteration ahead

We need not simply produce speculative children for the next iteration. We can view all possible speculative evaluations for a particle as an infinite tree with branching factor $2n + 1$. As we have already seen that doing one full level of the tree produces too many extra evaluations to be profitable, it is incredibly unlikely that doing two full iterations would produce decent results. But, if the idea of speculating more than one iteration ahead is combined with wisely pruning the possible evaluations based on branch statistics, we can use just a few extra evaluations to go two or more iterations ahead on the most likely branches.

When speculating more than one iteration ahead, the idea of Social Promotion cannot feasibly be implemented, as we can only determine correct branches for the first iteration. Thus in this case we always pick the child that has the highest value.

The question of which branches to take in this infinite tree is an intriguing one that we can only begin to explore here. If the branch corresponding to stagnation, $(-, -)$, has a 75% chance of being taken, as in the Random topology with most functions, we could speculate three iterations ahead on that branch and still have a 42% chance of predicting correctly. However, intuition would say that perhaps it is better to hope that the particle is productive instead of stagnant, so a branch where the particle updates its personal best might be more fruitful to try. In our experiments we try just one of the countless possibilities, but one that turns out to work very well. More work is needed to compare the different branching possibilities on various functions.

6 Experimental Setup

6.1 Parallel Architecture

At each iteration of the algorithms, we use one processor to perform one function evaluation. The exact evaluation time at which this architecture becomes reasonable depends on the amount of communication overhead in the parallel implementation and the number of particles in the swarm. For our implementation we found that time to be around 100 milliseconds for swarms of 50 particles. When the swarm size increases, the minimum evaluation time at which this architecture should be used decreases.

A general rule of thumb for when this architecture is useful is when the combined function evaluation time is greater than the communication overhead for a single iteration; i.e., $t * p > o + t$, where t is the function evaluation time, p is the desired number of particles (in our case p is equal to the number of available processors), and o is the amount of overhead per iteration. The amount of overhead per iteration will vary based on the communication topology and the number of particles, so this is only an approximation, but it can provide rough estimates of function evaluation times at which this parallel architecture should be used.

6.2 Function Evaluations vs. Time Steps

Results in serial PSO are typically presented in terms of function evaluations. This is a natural abstraction from implementation details that still allows a comparison of the implementation-independent aspects of each algorithm. The number of function evaluations performed is assumed to be proportional to wall-clock time, as all evaluations are performed in serial. Only reporting function evaluations could hide the fact that one algorithm requires more overhead than another and thus actually takes more time to perform the same number of function evaluations; however, function evaluations are still considered the standard method of reporting, as evaluation times for functions also vary greatly and could make the additional overhead negligible.

In parallel PSO on long-running functions, the natural way to present results is in terms of iterations, not function evaluations. This is because when all function evaluations at each iteration are performed concurrently, iterations are the direct equivalent of wall-clock time. Thus we report iterations in our results instead of function evaluations. But because SEPSO actually performs two iterations of PSO at each "iteration," and Social Promotion and Many Iterations make the idea of "iterations of PSO" somewhat nebulous, we instead call each "iteration" a "time step."

Just as serial PSO papers do not report actual running times of their specific implementations, we do not report running times, favoring the abstraction of time steps. However, given the time required for the evaluation of the objective function and the communication overhead per iteration for a specific implementation, a running time can be estimated from all of our results. Simply multiply the number of time steps by the sum of the function evaluation time and the overhead. We wish to stress that the "time steps" we report are proportional to wall-clock time, given the architecture we have assumed.

6.3 Functions Used

We experimented with five common benchmark problems defined in (Herrera et al., 2010): Rastrigin, Sphere, Schwefel 2.21, Griewank, and Bohachevsky. While benchmark functions take fractions of a second to evaluate and thus have no need of parallelization, they are useful for comparing algorithms because they stand as surrogates for the kinds of functions practitioners are actually interested in, and they allow us to explore the behavior of optimization algorithms in a simple and standardized way. To provide additional evidence that performance on benchmark functions corresponds to performance on real-world problems, we also tested our parallelization methods on a typical research problem, that of fitting a model to a large quantity of data, which we refer to as "the model fitting problem."

Rastrigin is initialized in $[-5.12, 5.12]^D$ and is defined as:

$$f_{Rastrigin}(\mathbf{x}) = \sum_{i=1}^{D} \left(z_i^2 - 10\cos(2\pi z_i) + 10 \right), z = x - c$$

Sphere is initialized in $[-50, 50]^D$ and is defined as:

$$f_{Sphere}(\mathbf{x}) = \sum_{i=1}^{D} z_i^2, z = x - c$$

Schwefel 2.21 is initialized in $[-500, 500]^D$ and is defined as:

$$f_{Schwefel}(\mathbf{x}) = \max_i |z_i|, 1 \leq i \leq D, z = x - c$$

Griewank is initialized in $[-600, 600]^D$ and is defined as:

$$f_{Griewank}(\mathbf{x}) = \frac{1}{4000} \sum_{i=1}^{D} z_i^2 - \Pi_{i=1}^{D} \cos\left(\frac{z_i}{\sqrt{i}}\right) + 1, z = x - c$$

And Bohachevsky is initialized in $[-15, 15]^D$ and is defined as:

$$f_{Bohachevsky}(\mathbf{x}) = \sum_{i=1}^{D} (x_i^2 + 2x_{i+1}^2 - .3\cos(3\pi x_i) - .4\cos(4\pi x_{i+1}) + .7)$$

In computing the branch statistics in Section 5.2.1, we also used the Rosenbrock function. That function is defined as:

$$f_{Rosenbrock}(\mathbf{x}) = \sum_{i=1}^{D} \left(100(x_{i+1} - x_i^2) + (x_i - 1)^2 \right)$$

The c in the function definitions is a shifted center, in order to avoid origin-seeking bias in the PSO algorithm and its variants (Monson & Seppi, 2005). We move the center of the shifted functions (all except Bohachevsky) to be halfway between the center and the boundary of the initialization region. For example, if the initialization region is $[-50, 50]^D$, the center is $(25)^D$. We tested all of these functions in their 20 dimension, 50 dimension, and 500 dimension varieties.

The model fitting problem is formulated as follows. We generated 10,000 data points from a radial basis function network with 10 bases and some added Gaussian noise. We then fit a radial basis function network to the data using PSO. This amounted to a 30 dimensional function to optimize, with a function evaluation time on the order of two seconds.

6.4 Parallelization Techniques Compared

The parallelization techniques we compare are the standard parallelization (here labeled Standard simply for ease of reference), our original speculative approach (recall that we refer to this as SEPSO), and the four relaxations of SEPSO discussed in Section 5. In presenting our results, we call the approach developed in Section 5.1 Pick Best. The methods described in Section 5.2 are called Pick Best Pruned and Social Promotion Pruned, and the method in Section 5.3 we call Many Iterations.

Using the same number of processors for each approach (and thus the same number of function evaluations per time step) requires that our speculative parallelizations have a smaller swarm size than the standard parallelization. For the topologies we used with SEPSO and Pick Best, a particle has three neighbors including itself. As shown in Table 1, this results in 7 speculative evaluations per particle. With one evaluation needed for the original, non-speculative particle, we have $8p$ evaluations for every two iterations, where p is the number of particles in the speculative swarm. The extra evaluations required in our speculative approach would instead be used to evaluate particles in standard parallelizations, so we compare swarms of size p in speculative evaluation with swarms of size $8p$ in standard approaches.

When performing pruning in Pick Best Pruned and Social Promotion Pruned, there are a large number of ways to prune speculative evaluations. We experimented with several, but present results for only one possible pruning. The pruning we present uses only the two branches where the $\mathbf{b}^N$ value was not updated: $(-,-)$ and $(S,-)$. Those branches are convenient in that no messages are needed from neighbors in order to produce the positions of the speculative particles—in distributed frameworks using several rounds of communication (see Appendix A), one of the rounds of communication can be dropped entirely. Pruning all but these branches also allows the use of arbitrarily dense topologies, as the number of speculative particles is no longer dependent on the number of neighbors the particle has.

Because pruning only requires two speculative evaluations per particle (along with evaluating the original particle), we can use swarms of size $\frac{1}{3}p$ when pruning to compare to a swarm of size p with the standard parallelization, instead of $\frac{1}{8}p$ with other techniques.

There are also many ways to speculate several iterations ahead, and with Many Iterations we again only show results for one of them. The combination of branches we tried uses seven speculative evaluations per particle, matching the swarm size of the original speculative algorithm. The seven evaluations we used corresponded to several iterations of branches $(-,-)$ and $(S,-)$. Two of the evaluations were just one iteration ahead, four were two iterations ahead, and one was three iterations ahead. The evaluations that were one iteration ahead were branches $(-,-)$ and

$(S, -)$; those that went two iterations ahead were formed by taking either branch $(-, -)$ or $(S, -)$and then branch $(-, -)$ or $(S, -)$; and the evaluation going three iterations ahead followed branch $(-, -)$ on all three iterations. As with Pick Best Pruned and Social Promotion Pruned, this choice of branches allows the use of arbitrarily dense topologies, as the number of speculative evaluations per particle is independent of the number of neighbors the particle has.

Finally, to show that our methods are orthogonal to traditional PSO variants, we show results of parallelizing a variation of the PSO motion equations, Bare Bones PSO (Kennedy, 2003), with both standard parallelizations and our speculative approaches. Our intent is not to exhaustively compare parallelization strategies on all possible PSO variants, but to show some evidence that speculative approaches are generally applicable and successful across PSO variants. These results are shown in Section 7.5.

6.5 Topologies

For each benchmark function we report results using the topology that is widely considered best for that function, as reported in the literature (e.g., Bratton & Kennedy (2007)). In this paper we limit ourselves to the Ring topology and the Complete topology, as is common practice, along with the Random topology (the parallel approximation to the Complete topology mentioned in Section 2). We also mentioned in Section 3 that some related work can be described as changes in topology, particularly that of having subswarms of fully connected topologies that occasionally communicate with each other. This related work focused mainly on functions with very fast evaluation times where such techniques drastically reduce interprocessor communication. With long function evaluations these topologies are not as practical, as only one particle is evaluated by each processor at each iteration. We experimented with a subswarm topology and found that in every instance except one it performed worse than either Complete or Ring, and thus we leave it out of the results except in the one instance where it improved performance.

Where the Complete topology would normally be preferred, we use a Random topology in SEPSO and Pick Best, as Complete leads to an explosion in the number of speculative evaluations (as noted, the other speculative techniques can still use a Complete topology; we often report results on both topologies for those methods). If speculative evaluation were not being performed, it is possible that the Complete topology would be used. However, the Complete topology also requires a very large amount of interprocessor communication in distributed PSO, so it is still quite possible that Random would be used even with standard parallelizations (McNabb et al., 2009). But, to be fair in our comparisons, we compare to the standard parallelization using both the Random topology and the Complete topology (labeled PSO Random and PSO Complete in our results). Again, the amount of communication overhead is heavily dependent on implementation details which are not the focus of this paper. A practitioner using a particular implementation can compare the results given here for whichever topology is most practical given the specific implementation.

7 Results

We frequently present tables summarizing our results. In each table, we bold the "best" method, meaning that it has at least a 90% success rate and its average time to completion is faster than all other methods that have at least a 90% success rate[4]. We used a double-sided t-test to test for statistical significance in these results, and all algorithms shown in boldface have a mean time to completion that is significantly lower than that of other algorithms, with a p-value of less than 10^{-5}.

We first present results in Section 7.1 for the 20 dimensional variants of the benchmark functions we tested. We give a discussion of each function individually at 20 dimensions, as there are interesting characteristics of the algorithms that are worth discussing. In Section 7.2 we then give a summary of results for the 50 dimensional variants of the benchmark functions; we do not go into as much detail in our discussion as the results are very similar to those in 20 dimensions. We finish our discussion of benchmark functions with their 500 dimensional variants in Section 7.3, and we discuss results on the model fitting problem in Section 7.4.

7.1 20 Dimensions

7.1.1 Sphere

First we look at Sphere, the simplest of common benchmark functions. The function has a single global optimum and no other local optima. Sphere is best optimized in terms of function evaluations with a small swarm using a Complete topology. We expect our methods to be perfect for such functions, and our results show this intuition to be correct. For this comparison we used 240 processors, so the methods had swarms of size 30 (for SEPSO, Pick Best, and Many Iterations), 80 (Pick Best Pruned and Social Promotion Pruned), and 240 (Standard). We compared too many methods here to fit into one graph, so we show just a few methods in Figure 3 and in Table 4 we show a summary of the results for all methods we tested.

In Figure 3 we see that Many Iterations using a Complete topology converges incredibly quickly on a very poor value. We found this behavior to be quite consistent across functions for this method, so we rarely show results for Many Iterations Complete. However, Pick Best Pruned works very well with Complete on this function.

7.1.2 Schwefel 2.21

Schwefel 2.21 is a function similar to Sphere (in that they are both unimodal), but Schwefel 2.21 is benefited more by larger swarms than Sphere is (as it is harder to optimize). Thus our speculative algorithms often fail to outperform Standard with 240 processors because the simple speculative methods only have 30 particles. As we will show later, when we use 800 processors at 50 dimensions, our methods perform much better. However, the pruned versions of our methods have 80 particles

[4] "Success rate" here means that the algorithm reached some predefined value which is mentioned in the table or figure wherever a success rate is reported.

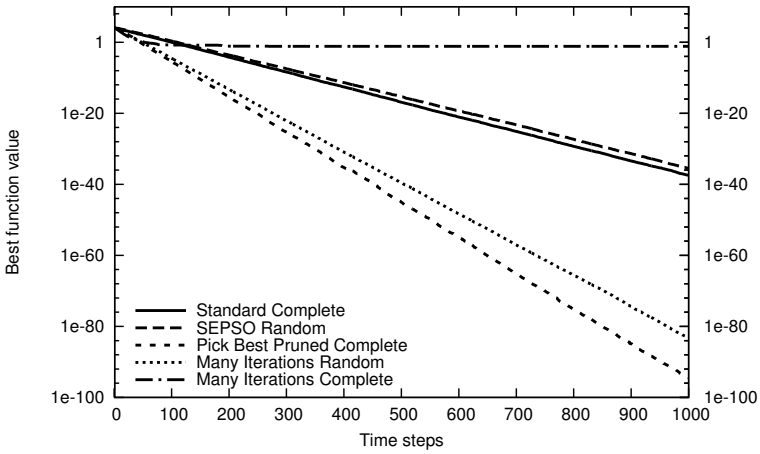

Fig. 3 Function Sphere with 20 dimensions, comparing a pertinent subset of possible methods. Each method performs one evaluation on each of 240 processors per time step.

Table 4 Summary of results for function Sphere with 20 dimensions, measuring number of time steps to reach a value of 1e-35. The "best" algorithm is shown in boldface. This indicates that at least 90% of runs successfully reach the value of 1e-35, and its mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Ring	0%	N/A	N/A
Standard Random	0%	N/A	N/A
Standard Complete	100%	931.0	19.1
SEPSO Ring	0%	N/A	N/A
SEPSO Random	85%	972.8	15.8
Pick Best Ring	100%	917.4	21.0
Pick Best Random	100%	768.1	14.3
Pick Best Pruned Ring	100%	967.8	9.9
Pick Best Pruned Random	100%	693.5	10.4
Pick Best Pruned Complete	**100%**	**389.6**	**9.6**
Social Promotion Pruned Ring	0%	N/A	N/A
Social Promotion Pruned Random	100%	971.1	10.6
Social Promotion Pruned Complete	100%	777.8	16.7
Many Iterations Ring	100%	575.2	9.8
Many Iterations Random	100%	442.4	7.1
Many Iterations Complete	0%	N/A	N/A

and thus are able to outperform Standard even with only 240 processors. Because the graph of Schwefel 2.21 looks very similar to that of Sphere, we simply present all of our results in Table 5.

7.1.3 Rastrigin

Rastrigin is a multi-modal function that is best optimized with a large, Complete swarm. It has been shown that with Rastrigin, the more particles there are in the swarm, the lower function value it finds, up to at least 4000 particles (McNabb et al., 2009). Smaller swarms get caught in local optima and converge to poorer

Table 5 Summary of results for function Schwefel with 20 dimensions, measuring number of time steps to reach a value of 1e-06. The "best" algorithm is shown in boldface. This indicates that at least 90% of runs successfully reach the value of 1e-06, and its mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Ring	0%	N/A	N/A
Standard Random	0%	N/A	N/A
Standard Complete	100%	837.2	44.8
SEPSO Random	0%	N/A	N/A
Pick Best Random	5%	938.0	0.0
Pick Best Pruned Random	**100%**	**639.9**	**29.4**
Pick Best Pruned Complete	**100%**	**597.7**	**77.9**
Social Promotion Pruned Random	75%	938.0	33.0
Social Promotion Pruned Complete	100%	815.2	43.5
Many Iterations Random	100%	783.5	94.5
Many Iterations Complete	0%	N/A	N/A

Table 6 Summary of results for function Rastrigin with 20 dimensions, measuring number of time steps to reach a value of 20. The "best" algorithm is shown in boldface. This indicates that at least 90% of runs successfully reach the value of 20, and its mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Ring	25%	743.4	212.0
Standard Random	100%	372.1	86.6
Standard Complete	**95%**	**273.1**	**99.7**
SEPSO Ring	5%	471.0	0.0
SEPSO Random	15%	208.0	17.0
Pick Best Ring	35%	489.0	152.0
Pick Best Random	15%	136.0	40.1
Pick Best Pruned Ring	95%	545.9	208.5
Pick Best Pruned Random	10%	95.0	11.0
Pick Best Pruned Complete	5%	59.0	0.0
Social Promotion Pruned Ring	5%	580.0	0.0
Social Promotion Pruned Random	80%	278.2	136.3
Social Promotion Pruned Complete	50%	200.5	130.2
Many Iterations Ring	10%	117.0	25.0
Many Iterations Random	5%	57.0	0.0
Many Iterations Complete	0%	N/A	N/A

values. In fact, none of the algorithms that we tested were able to solve this problem optimally. We thus set a somewhat arbitrary success threshold, simply to be able to compare algorithms.

Because our speculative algorithms require significantly smaller swarm sizes, we would expect to not perform very well on functions such as Rastrigin. Our experiments show our intuition to be correct. In this experiment we used 240 processors, so SEPSO and Pick Best each had 30 particles, and the Standard algorithms had 240 particles. As expected, SEPSO and Pick Best converge quickly to worse local optima than Standard does. Figure 4 shows the results graphically, and Table 6 shows results for all of the methods we tried.

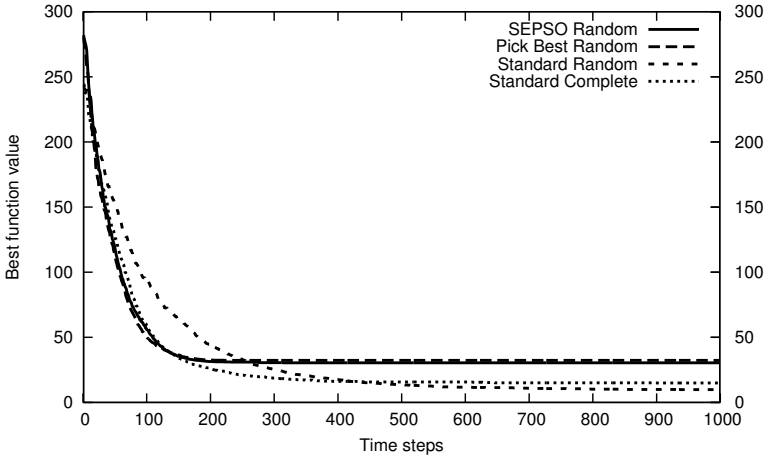

Fig. 4 Function Rastrigin with 20 dimensions. Each method performs one evaluation on each of 240 processors per time step.

Table 7 Summary of results for function Griewank with 20 dimensions, measuring number of time steps to reach a value of 1e-06. The "best" algorithms are shown in boldface. This indicates that at least 90% of runs successfully reach the value of 1e-06, and their mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Ring	100%	762.2	19.4
SEPSO Ring	100%	426.0	81.0
Pick Best Ring	**100%**	**272.0**	**17.7**
Pick Best Pruned Ring	**100%**	**282.9**	**10.7**
Social Promotion Pruned Ring	100%	482.0	16.6
Many Iterations Ring	84%	183.2	49.4

7.1.4 Griewank

It is recommended in the literature to use the Ring topology when optimizing the Griewank function, as Complete is prone to premature convergence on a local optimum (Bratton & Kennedy, 2007). The global optimum of Griewank has a value of 0. When most trials reach the global optimum but a few get stuck, the resultant "average value" graph has a flat line that is misleading. Thus instead we present plots showing the percent of trials that have passed some threshold at each time step, as is common practice with these functions (Mendes, 2004). The threshold we chose for this case was 10^{-6}, as that value is below any local optima and the swarm always successfully reaches 0 once is passes that point.

We ran 50 trials of each experiment with Griewank, so that the curves are more smooth. We show results in Figure 5 for swarms of size 100 and 800 using the Ring topology. One can see in the figure that SEPSO reaches the global optimum on average close to twice as fast as Standard, while Pick Best is close to three times as fast and Many Iterations is even faster when it is successful, though it is only successful 84% of the time. Table 7 shows all of our results in tabular form.

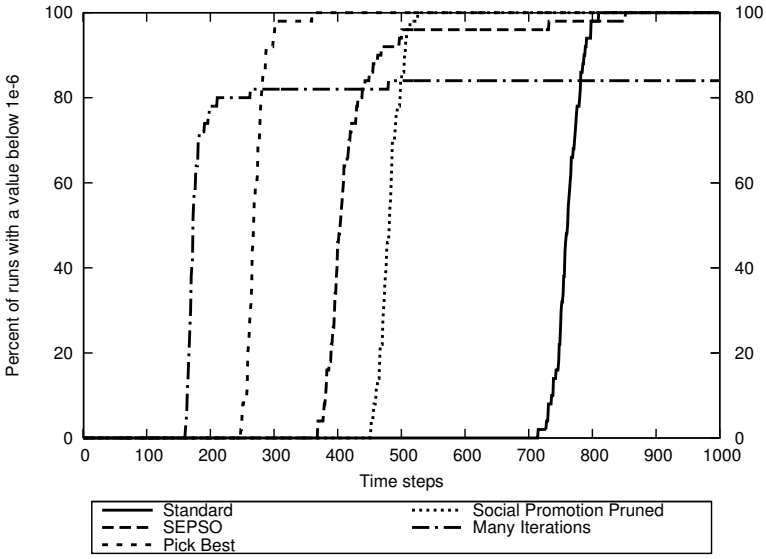

Fig. 5 Function Griewank with 20 dimensions. Each method performs one evaluation on each of 800 processors per time step. Instead of showing average function value, we show the percent of runs that are sufficiently close to the global optimum by each iteration (10^{-6}, as noted on the y-axis and explained in the text). All algorithms use the Ring topology.

We pause here to show some interesting characteristics of the speculative techniques we have proposed. With 800 processors our methods perform very well on this function. With 240 processors, however, the results are much more mixed. Because 240 processors is near the point at which speculative evaluation becomes useful, it is enlightening to see the behavior of the various algorithms at this point.

We show results in Figure 6 for swarms of size 30 and 240 using the Ring topology. One can see in the figure that when SEPSO is successful, it finds the optimum much faster than Standard. However, because the swarm size is so small, SEPSO gets stuck almost half of the time.

When we look at the performance of our Pick Best approach, we see that it greatly improves performance on Griewank. This is somewhat counter-intuitive, because Griewank is deceptive and Pick Best seems like a greedy algorithm. But in Figure 7 we see that Pick Best improves accuracy over SEPSO by 20%, while at the same time finding the optimum over 100 time steps sooner on average. It seems that while Pick Best is locally greedy, there is enough exploration in the seven speculative evaluations to overcome the inherent greediness of the approach.

When we introduce pruning, our intuition about Pick Best turns out to be more correct. While adding 50 more particles to the swarm (as pruning allows us to have 80 particles with 240 processors instead of only 30), Pick Best with pruning still gets stuck just as often as the original Pick Best. However, Social Promotion does well with pruning; it increases the success rate to close to 100%, while still finding the optimum on average much faster than the original PSO. These results are shown in Figure 8.

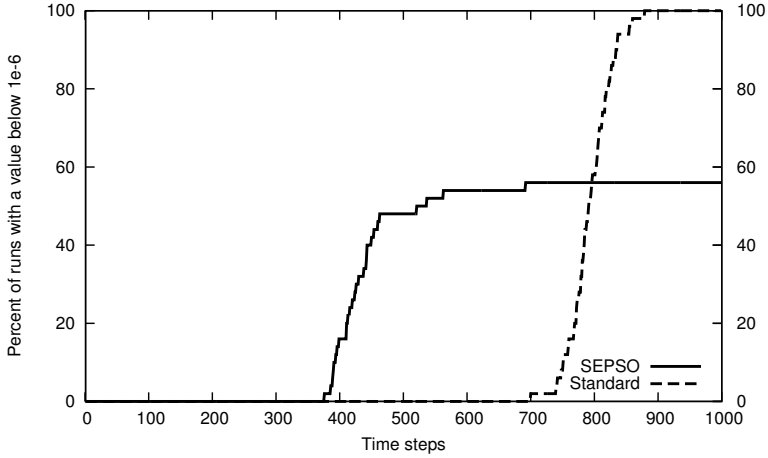

Fig. 6 Function Griewank with 20 dimensions. Each method performs one evaluation of the objective function on each of 240 processors per time step. We show the percent of runs that are sufficiently close to the global optimum at each iteration. All algorithms use the Ring topology.

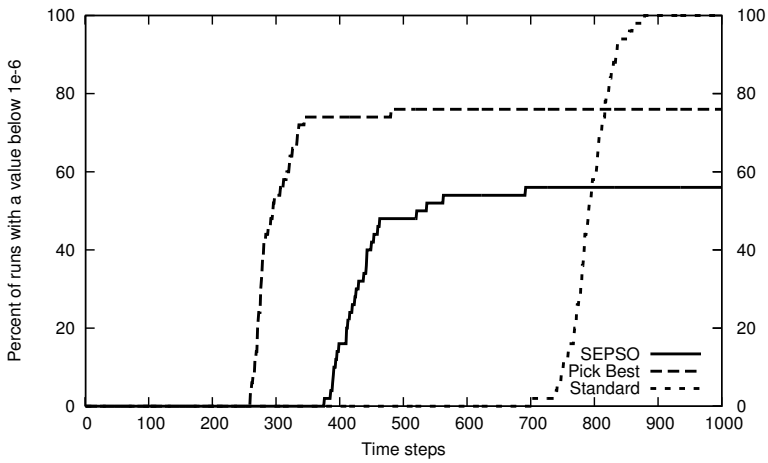

Fig. 7 Function Griewank with 20 dimensions, comparing Pick Best with results from Figure 6. Each method performs one evaluation on each of 240 processors per time step. We show the percent of runs that are sufficiently close to the global optimum at each iteration. All algorithms use the Ring topology.

With Griewank, the premature convergence problems inherent in picking the best child are exacerbated when speculating several iterations ahead. When Many Iterations finds the optimum, it finds it quicker than any other method we tried, on average four times faster than Standard. However, it also gets stuck and fails to find the optimum more than any other method. The results are shown in Figure 9. Figure 9 is also interesting in that it highlights the trade-off between accuracy and speed in the various approaches at this swarm size. The faster the approach finds the optimum, the less likely it is to be successful.

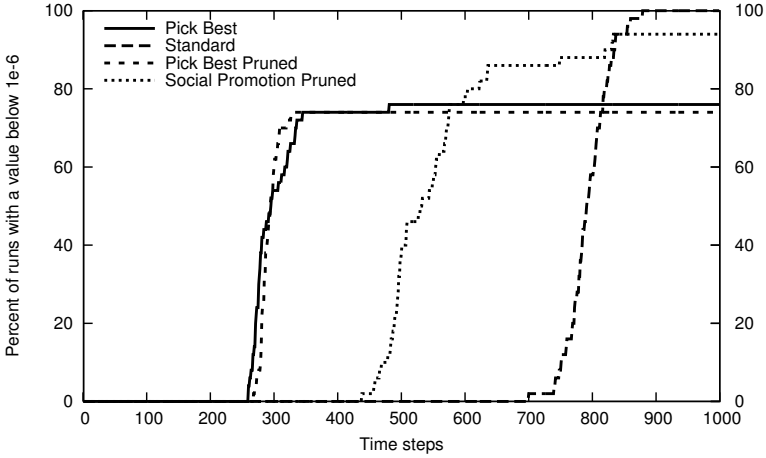

Fig. 8 Function Griewank with 20 dimensions, comparing pruning with the best results from Figure 7. Each method performs one evaluation on each of 240 processors per time step. We show the percent of runs that are sufficiently close to the global optimum at each iteration. All algorithms use the Ring topology.

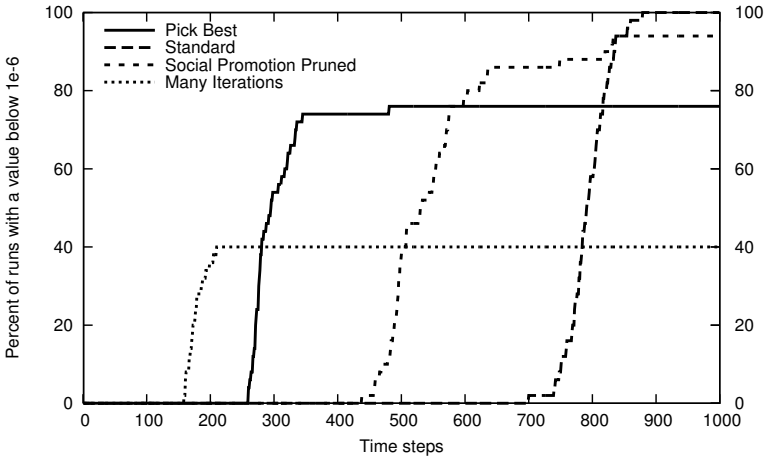

Fig. 9 Function Griewank with 20 dimensions, comparing speculating several iterations ahead with selections from Figure 8. Each method performs one evaluation on each of 240 processors per time step. We show the percent of runs that are sufficiently close to the global optimum at each iteration. All algorithms use the Ring topology.

Note here that Figure 9 is interesting to compare to Figure 5. The ordering of the methods in terms of the number of time steps to completion is the same in both figures. What is different is that at 240 processors, most of our methods fail to find the optimum 100% of the time, while at 800 processors, all but Many Iterations succeed 100% of the time, and Many Iterations is very close. At some point between 240 and 800 processors, SEPSO and Pick Best become successful, and at that point it is by far better to use speculative methods than standard

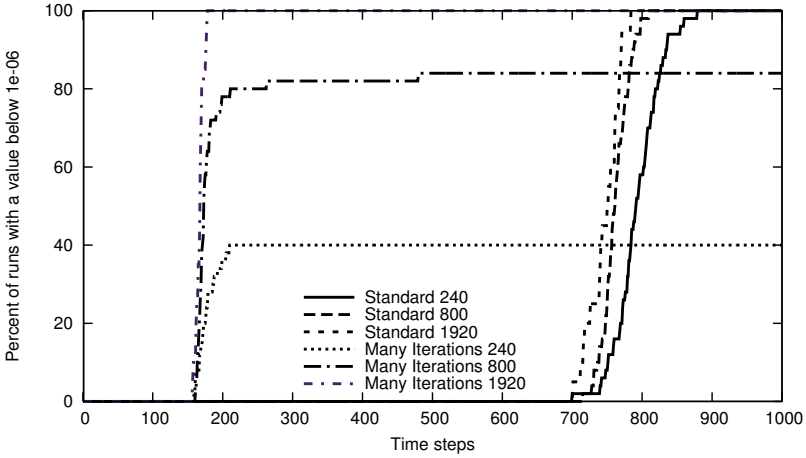

Fig. 10 Function Griewank with 20 dimensions, comparing the performance of standard and speculative parallelizations as the number of processors increases. We show the percent of runs that are sufficiently close to the global optimum at each iteration. All algorithms use the Ring topology.

parallelizations. At some number above 800 processors, Many Iterations will be become successful and will be the preferred method.

To further show this point, we present results using 1920 processors on the 20-dimensional Griewank function in Figure 10. With 1920 processors Many Iterations finds the optimum 100% of the time. What is interesting is that while going from 240 processors to 1920 processors, the standard method of parallelization only decreased its time to completion by 5.9%. That is, the extra 1680 processors in standard parallelizations provide *no appreciable benefit*. In contrast, Many Iterations with 1920 processors has the same swarm size of 240 particles and uses the additional processors for speculative evaluation, giving a decrease in time to completion of 78.9%, or a factor of almost 5. Once there are enough particles in the swarm to guarantee success when optimizing a function, the best use of additional processors is to speculate as far ahead as possible, speeding up the progress of the algorithm.

7.1.5 Bohachevsky

Bohachevsky is a unimodal function best optimized with a Complete swarm. It is similar to Griewank in that there is a global optimum with a value of 0, and the swarm sometimes finds it and sometimes does not. Thus we present a graph similar to those of Griewank, because "average value" graphs have a misleading flat line. We used 480 processors to optimize this function. In Figure 11 we show a plot with a few pertinent methods, while Table 8 shows all of the results. All of our speculative approaches found the optimum much quicker than Standard with a Random topology. However, SEPSO was slower than Standard Complete and got stuck 25% of the time. Pick Best, Pick Best Pruned, and Many Iterations all outperformed Standard Complete, with Many Iterations finding the optimum about twice as fast.

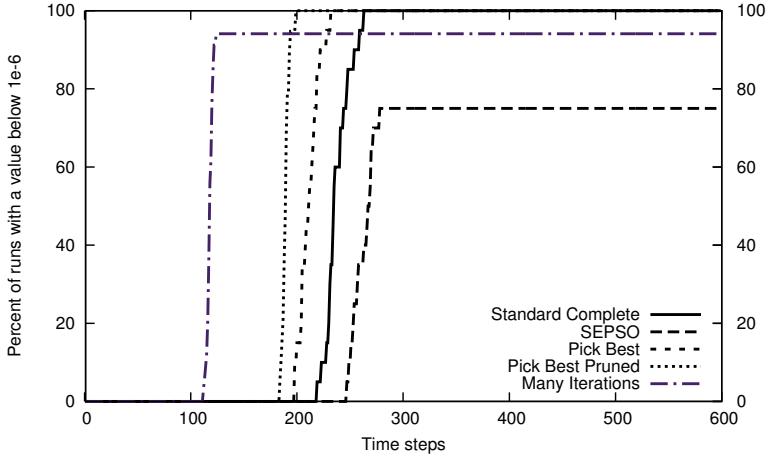

Fig. 11 Function Bohachevsky with 20 dimensions. Each method performs one evaluation on each of 480 processors per time step. We show the percent of runs that are sufficiently close to the global optimum at each iteration.

Table 8 Summary of results for function Bohachevsky with 20 dimensions, measuring number of time steps to reach a value of 1e-06. The "best" algorithm is shown in boldface. This indicates that at least 90% of runs successfully reach the value of 1e-06, and its mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Random	100%	472.3	6.6
Standard Complete	100%	238.2	11.2
SEPSO Random	75%	261.6	9.2
Pick Best Random	100%	211.8	9.2
Pick Best Pruned Random	100%	189.7	3.4
Pick Best Pruned Complete	25%	105.4	3.9
Social Promotion Pruned Random	100%	260.3	6.5
Social Promotion Pruned Complete	90%	197.9	12.2
Many Iterations Random	**94%**	**118.2**	**2.6**
Many Iterations Complete	0%	N/A	N/A

7.2 50 Dimensions

Given our observation from the 20 dimensional benchmark functions that Many Iterations most often outperforms other speculative approaches, we only present results for Many Iterations and Standard for the 50 and 500 dimensional variants. We used 800 processors for these experiments because the problems are more difficult, and because speculative approaches perform best when many processors are available, as we showed in Section 7.1.4. Many Iterations Complete showed the same premature convergence that was observed with the 20 dimensional benchmark functions, so we do not include those results in our tables; the success rate was 0% for all functions.

Tables 9 through 13 present summaries of our results for each of the benchmark functions. As with the 20 dimensional variants, our speculative methods did

Table 9 Summary of results for function Rastrigin with 50 dimensions, measuring number of time steps to reach a value of 100. The "best" algorithm is shown in boldface. This indicates that at least 90% of runs successfully reach the value of 100, and its mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Ring	5%	776.0	0.0
Standard Random	**100%**	**593.2**	**89.5**
Standard Complete	80%	343.8	145.8
Many Iterations Ring	25%	223.8	67.1
Many Iterations Random	25%	98.6	11.0

Table 10 Summary of results for function Sphere with 50 dimensions, measuring number of time steps to reach a value of 1e-06. The "best" algorithm is shown in boldface. This indicates that at least 90% of runs successfully reach the value of 1e-06, and its mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Ring	0%	N/A	N/A
Standard Random	30%	986.8	8.2
Standard Complete	100%	458.4	12.3
Many Iterations Ring	100%	288.7	3.9
Many Iterations Random	**100%**	**248.6**	**5.1**

Table 11 Summary of results for function Schwefel with 50 dimensions, measuring number of time steps to reach a value of 80. The "best" algorithms are shown in boldface. This indicates that at least 90% of runs successfully reach the value of 80, and their mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Ring	0%	N/A	N/A
Standard Random	70%	871.6	57.1
Standard Complete	100%	599.0	85.6
Many Iterations Ring	**100%**	**238.6**	**34.2**
Many Iterations Random	**100%**	**242.9**	**95.4**

Table 12 Summary of results for function Griewank with 50 dimensions, measuring number of time steps to reach a value of 1e-06. The "best" algorithm is shown in boldface. This indicates that at least 90% of runs successfully reach the value of 1e-06, and its mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Ring	100%	1898.5	25.7
Many Iterations Ring	**97%**	**312.4**	**6.3**

not perform well on Rastrigin, though for all other benchmarks Many Iterations significantly outperformed Standard.

A summary of the results is shown in Table 14. For functions where Many Iterations outperformed Standard, we report the average speed up. On all functions except Rastrigin, our methods showed an average speed up of from two to six times compared to previous methods.

Table 13 Summary of results for function Bohachevsky with 50 dimensions, measuring number of time steps to reach a value of 0.01. The "best" algorithm is shown in boldface. This indicates that at least 90% of runs successfully reach the value of 0.01, and its mean time to completion is significantly lower than all other algorithms, with a p-value of less than 10^{-5}.

Algorithm	% Complete	Mean	St. Dev.
Standard Ring	100%	1434.0	62.1
Standard Random	100%	704.5	19.7
Standard Complete	20%	377.5	14.9
Many Iterations Ring	**95%**	**229.8**	**11.4**
Many Iterations Random	35%	171.6	4.0

Table 14 Average speed up, comparing the number of time steps to completion for the best speculative topology to the time steps to completion for the best standard topology. All functions have 50 dimensions.

Function	Speed Up Factor
Rastrigin	N/A
Sphere	1.84
Schwefel	2.51
Griewank	6.08
Bohachevsky	3.07

7.3 500 Dimensions

At 500 dimensions the performance of constricted PSO on benchmark functions becomes rather dismal. In almost all cases, Standard fails to make any significant progress. Our search for an explanation seems to say that the space is too large for particles to converge to the same point (due to overshooting in some dimension), so all updates to the best position found come from a single particle wandering in the space. Every time that single particle finds a new best position, its velocity contracts, so the particle is unable to make significant progress on its own. In constricted PSO it is necessary to have a collection of particles exploring a promising location to keep the particles' velocities from contracting prematurely, and in 500 dimensions the space is too large to get the collection of particles to the same location.

The results that we present show that while Standard parallelizations suffer from this problem, Many Iterations does not. Though we can provide some intuition for why this might be the case, it remains as future work to provide a strong theoretical explanation. The intuition is that we are speculating about future positions along paths where each particle's velocity does not contract. The particular branches we selected corresponded to not having received a new value for $\mathbf{b}^P$ nor $\mathbf{b}^N$. In those branches, the particle continues in the same direction it was going, without a contraction in velocity.

Because none of the Standard approaches were successful at optimizing the 500 dimensional benchmark functions, we do not present tables as we did for the 50 dimensional variants. We do, however, show a few figures demonstrating the results we have just explained. Figure 12 shows the function Sphere, Figure 13 shows Griewank, and Figure 14 shows Rastrigin. Note that at 500 dimensions speculative approaches outperform standard parallelizations on Rastrigin, whereas they did not at 20 and 50 dimensions.

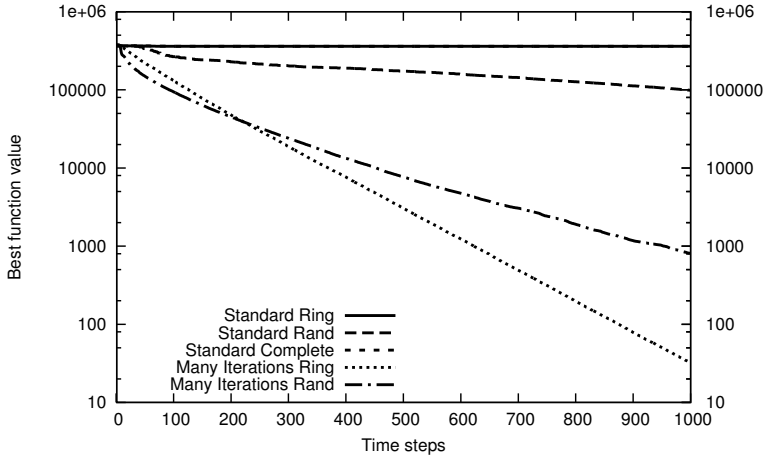

Fig. 12 Function Sphere with 500 dimensions. Each method performs one evaluation on each of 800 processors per time step.

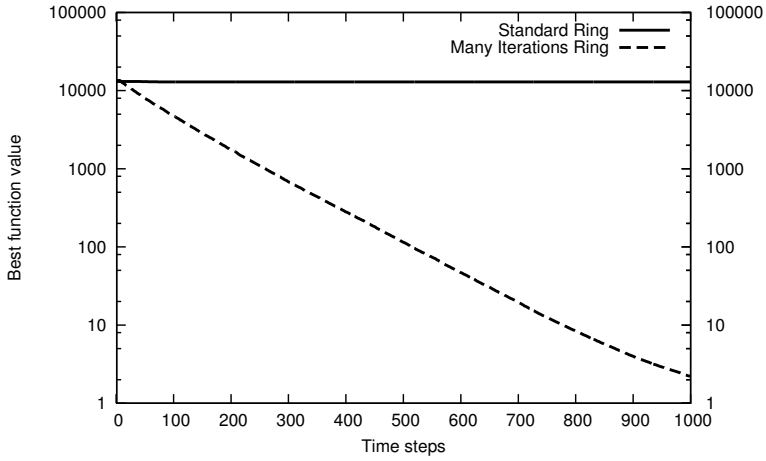

Fig. 13 Function Griewank with 500 dimensions. Each method performs one evaluation on each of 800 processors per time step.

7.4 Model Fitting

For the model fitting problem we used 144 processors. We show results for three methods: Standard with a Random topology, Standard with a Subswarm topology, and Many Iterations with a Random topology. Standard with a Subswarm topology had eight independent subswarms of 18 particles each, and the particles in each subswarm were connected with a Random topology. In the graphs we call this method Standard Subswarms.

Figure 15 shows our results for this function. The function value reported is the sum squared error of the model fit. Figure 16 shows the percent of runs that reached a value for sum squared error of 55,000 by each time step, which we

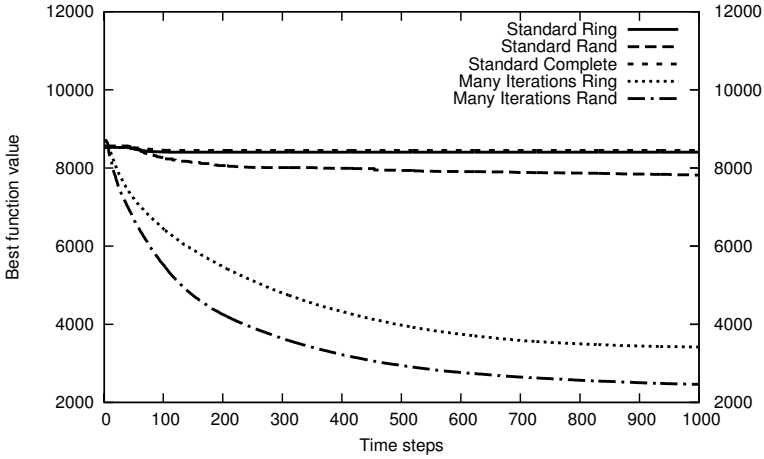

Fig. 14 Function Rastrigin with 500 dimensions. Each method performs one evaluation on each of 800 processors per time step.

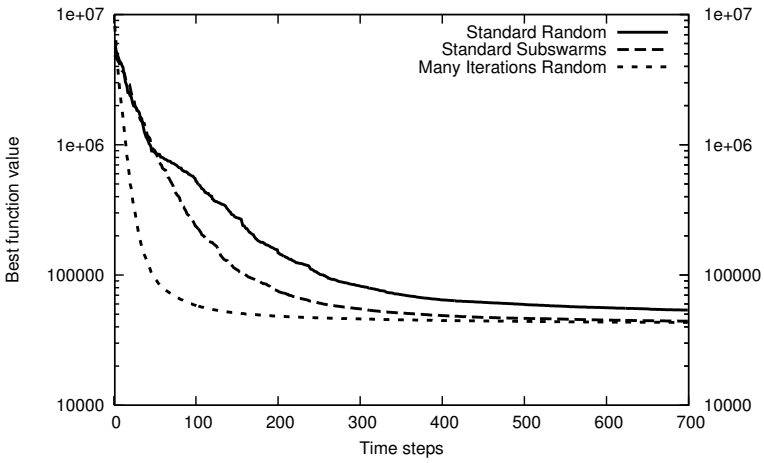

Fig. 15 Results for fitting a radial basis function to noisy data. We use 144 processors for each method, so each time step corresponds to 144 function evaluations.

designate as successful. Many Iterations took on average 126 time steps to reach this value, while Standard Subswarms took on average 298. The p-value for a t-test on this statistic is less than 10^{-8}. With our implementation of parallel PSO, each time step took on average 10.06 seconds; 1.83 seconds of that was function evaluation time and the rest was overhead. With this particular implementation, then, Many Iterations takes on average 21 minutes to reach a successful value, while Standard Subswarms takes 50 minutes.

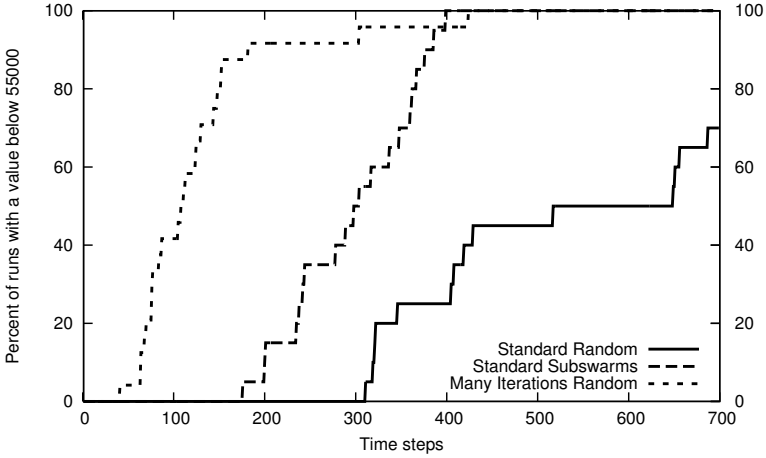

Fig. 16 Results for fitting a radial basis function to noisy data. We use 144 processors for each method, so each time step corresponds to 144 function evaluations. Here we show the percent of runs with a function value below 55000, which we considered successful.

7.5 Bare Bones PSO

In order to demonstrate that our methods are applicable across PSO variants, we ran experiments with Bare Bones PSO. For this simple experiment we use the Sphere function with 20 dimensions, taking the best Standard algorithm and the best speculative algorithm on that function, which in this case is Standard Complete and Pick Best Pruned Complete, as shown in Section 7.1.1. Figure 17 shows the results of running each of those algorithms with the Bare Bones PSO equations. We see a speed up of 1.6 when using speculative parallelizations instead of Standard techniques. While this is not quite as large a speed up as the 2.5 seen with constricted PSO, it is still a significant decrease in time. The key point of this experiment is that our methods still show improvements over Standard parallelizations in PSO variants other than the standard constricted PSO.

8 Conclusions

We have described a new technique for using processors in parallel PSO to improve the performance of the algorithm. To our knowledge, this is the first time extra processors have been used to do anything in PSO besides increase the swarm size. In an increasingly parallel world, such advancements will prove to be crucial to the continued effectiveness of PSO.

We have detailed how to perform speculative evaluation in PSO in several different parallel architectures. Using this methodology, the behavior of the original PSO algorithm can either be exactly reproduced, two iterations at a time, or the behavior can be modified in order to improve performance. While exactly reproducing PSO sometimes uses too many extra processors to be useful, when we allow ourselves some freedom with the algorithm we see great improvements over previous methods. We have shown results that conclusively demonstrate the

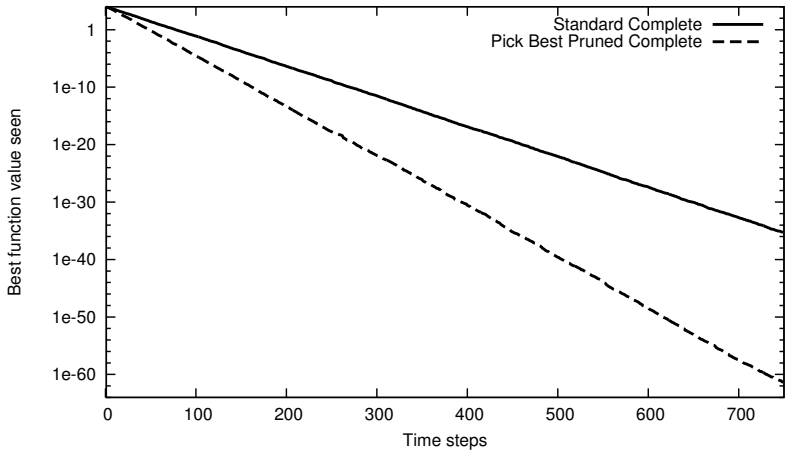

Fig. 17 Results for optimizing the 20-dimensional Sphere function with Bare Bones PSO. We use 240 processors for each method, so each time step corresponds to 240 function evaluations.

superiority of our techniques for several functions over the standard practice of adding particles to the swarm when extra processors are available, giving speed ups of up to six compared to previous parallelization techniques.

What we have presented is not a new variant of the PSO equations. We presented a new parallelization technique, so we compared parallelization strategies on the same algorithm, the original PSO. Our most promising techniques do change the behavior of the PSO algorithm slightly, as do some other previously proposed parallelization techniques, such as asynchronous PSO. However, our methods are applicable to almost all PSO variants, and so a comparison using the same variant for each of the parallelization techniques tested is justified. We presented results on a separate PSO variant that give evidence for this claim.

We have given five different possible approaches to speculative evaluation, each of which has different properties. These approaches perform differently on different functions and at different swarm sizes, as would be expected by the No Free Lunch Theorem for Optimization (Wolpert & Macready, 1997). We have given a brief evaluation of the premature convergence properties of these methods on deceptive functions when a smaller number of processors is available. We have also shown evidence that when many processors are available by far the best thing to do in most cases is to speculate as far ahead as the extra processors allow. This point is shown in a single graph in Figure 10. In that case, the extra processors in standard parallelizations provided *no appreciable benefit.* In contrast, speculative methods allow additional processors to provide additional speed ups, as the more processors are available the farther ahead the algorithm can speculate.

Though our methods show great improvements on some functions, they do not work for all functions at the swarm sizes we were able to experiment with. As is commonly known, in PSO there is a trade-off between exploration and exploitation. Some functions need only minimal exploration, and some never seem to have enough. Increasing the swarm size is a natural way to increase exploration in a parallel environment. However, once "enough" exploration has been reached for

any particular function, adding additional particles adds only incremental benefits. At this point, a better use of the additional processors, as we have shown, is to perform some amount of speculative evaluation.

Large parallel clusters are often required to successfully optimize practical modern problems, and the world's computing resources are increasingly coming from such clusters. To properly use PSO with these systems, a balance needs to be made between using processors to increase the swarm size and using them to increase the speed of the algorithm. This work is a first step in that direction that opens the door to many future improvements on speculative methods in the parallelization of PSO.

9 Future Work

In this work we have focused on PSO itself and not all of its variants. It remains as future work to apply speculative approaches to recent and popular PSO variants, such as the Fully Informed Particle Swarm (Mendes et al., 2004). While our methods will not always be immediately applicable to every variant, we are confident that some kind of speculative approach will be beneficial to the parallelization of all forms of PSO, especially as the number of processors used gets into the thousands.

We mentioned related work showing that increasing the swarm size throughout the course of the algorithm could provide improved performance over a fixed swarm size in serial PSO (Montes de Oca et al., 2010). If this method were extended to parallel PSO, most processors would be idle in the first few iterations, while more would be utilized at the end. During iterations where there are many un-utilized processors, a natural use of them would be speculative evaluation, performing two or more of those iterations at a time.

The issues of the sampling distribution of speculative relaxations, branch statistics, and why standard PSO fails at 500 dimensions were briefly mentioned in this paper. Each of those issues needs further treatment. The sampling distribution of our speculative methods could be compared to Poli's description of standard PSO's sampling distribution (Poli, 2008b). The branch statistics could be used to analyze topologies and discover why certain topologies work well on some functions but not on others; perhaps PSO performance is more dependent on the branch statistics of a combination of topology and function than on the topology itself. And discovering exactly why constricted PSO fails at 500 dimensions while Many Iterations does not could lead to improvements in the standard PSO algorithm, even when not running in parallel.

We opened the door to speculative parallelization methods in PSO and described the possible speculative evaluations to perform as an infinite tree from which branches are selected. However, we only presented a few of the countless possibilities for selecting those branches. Our methods for determining which speculative evaluations to perform were independent of the particle; all particles performed the same number and type of evaluations. Another way to allocate speculative evaluations is to somehow use the performance of each particle to determine how many and which extra evaluations it can have.

Any other optimization algorithm that only depends on current sampling positions when computing the next position to sample can be parallelized with this

technique. In particular, genetic algorithms produce future generations by combining individuals from the current generation. With a large population size there would be an unwieldy amount of possible future individuals, but the potential exists to modify the algorithm to use some kind of speculative evaluation.

Appendix A Implementing Speculative Evaluation

It is not trivial in some parallel architectures to determine which speculative position was the correct next position of each particle. In this section we discuss in detail some important considerations in the implementation of our methods. First we discuss the relatively easy case of a centralized parallel PSO algorithm with a master computer and many slaves. In such an architecture, the master keeps track of all necessary information with only trivial message passing needed. Then we discuss the more complicated case of a distributed algorithm, where each particle is on its own and needs to send and receive messages to and from other particles. Finally we discuss the further complications of a dynamic topology such as Random, where a particle's neighbors change from one iteration to another.

A.1 Terminology

To aid in describing our methodology, we introduce a few terms. A particle's set of *speculative children* is the set of all possible next iteration states (including the particle's position, $\mathbf{b}^N$ and $\mathbf{b}^P$ positions) that a particle could have. We use p_t to denote a particle at iteration t and s_{t+1} to denote one of p_t's speculative children, corresponding to one of the rows in Table 1. n_t is a neighbor of particle p_t. Sets of particles are given by $\mathbf{p}$, $\mathbf{s}$, or $\mathbf{n}$, whereas single particles are simply p, s, or n.

We separate each iteration of PSO into several steps. First there is the motion step, where a particle updates its position and velocity. Then a particle's position is evaluated, and the particle updates its current value and its personal best. Finally, a particle gets information from its neighbors and updates its neighborhood best.

A particle at iteration $t-1$ that has been moved to iteration t using (1) and (2), but whose position has not yet been evaluated, is denoted as p_t^{-e}. Once its position has been evaluated, but it has still not yet received information from its neighbors, it is denoted as p_t^{-n}. Only when the particle has updated its neighborhood best is it a complete particle at iteration t. It is then simply denoted as p_t.

A.2 Centralized Algorithms

In a centralized, or Master-Slave, parallel PSO algorithm, one machine, the master, keeps track of all necessary information, and all other machines are merely used to evaluate the objective function at various positions as directed by the master (Belal & El-Ghazawi, 2004). To perform speculative evaluation in such an architecture, the master generates the positions to evaluate speculatively as in (5). After having the slaves evaluate the objective function at all necessary positions, the master then decides which position to accept for each particle, as in (9). The outline of the procedure is given in Algorithm 1.

Given a set of particles at iteration $t-1$ (perhaps which have just been initialized), the master must move each particle using (1) and (2) to obtain the set $\mathbf{p}_t^{-e}$. For each particle p_t^{-e}, the master must then get its set of neighbors $\mathbf{n}_t^{-e}$ and use their positions, along with the position of p_t^{-e}, to calculate all possible values of $\mathbf{X}_{t+1}^c$, using (5). These positions, along with the original particle's associated information (such as values for $\mathbf{b}^P$ and $\mathbf{b}^N$), define a set of speculative children, $\mathbf{s}_{t+1}^{-e}$. The master then has a set of particles $\mathbf{p}_t^{-e}$, and for each particle a set of speculative children $\mathbf{s}_{t+1}^{-e}$, which can all be evaluated at once.

The master then has the slaves evaluate the particles. Once all particles, speculative and original, have been evaluated and the values reported to the master, the master determines

Algorithm 1 Speculative Evaluation in a Centralized PSO

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

which speculative child of each particle was the correct one. Mathematically, this corresponds to the evaluation of an indicator function similar to that found in (3). In practice, this is done first by updating each (original) particle's $\mathbf{b}^P$, if necessary, then by updating the particle's $\mathbf{b}^N$ with information from the particle's neighbors. This is simply the original PSO algorithm, and corresponds to steps 1–5 in Algorithm 1. Given the updates to $\mathbf{b}^P$ and $\mathbf{b}^N$, the case from Table 1 can be determined, as per (9). The child with the matching case is kept, and all other speculative children are discarded (step 7 in Algorithm 1).

The parent p_t must pass its personal best value to the child, as the child knows only the position that it guessed, not the function value at that position. It is possible that both p_t and s_{t+1} update their personal bests, but p_t's value is better. For example, suppose that p_{t-1} has a personal best value of 3, and that we are seeking to minimize the function. p_t^{-e} is created, and s_{t+1}^{-e} is moved assuming that p_t has updated its personal best with its position at time t. Then both p_t^{-e} and s_{t+1}^{-e} are evaluated, with values 1 and 2, respectively. s_{t+1}^{-n} would think that its current position is its personal best, as the value it found, 2, is better than its previous personal best value of 3. It needs to receive the personal best value from its parent to know that its personal best position $\mathbf{b}^P$ is actually the position of p_t, not s_{t+1}.

The parent also needs to pass the value of the neighborhood best that the child guessed. The child only knows the position and needs the value in order to make future comparisons between neighborhood best positions (step 8).

Upon picking the correct branch for each particle and updating the child's personal best and neighborhood best value (from iteration t), the result is the set $\mathbf{p}_{t+1}^{-n}$, as the particles are now no longer speculative. What remains is to update the neighborhood best of those particles from their neighbors (from iteration $t+1$), as above, to obtain $\mathbf{p}_{t+1}$. That set of particles can subsequently be used to produce the sets $\mathbf{p}_{t+2}^{-e}$ and $\mathbf{s}_{t+3}^{-e}$ (steps 1 and 2 in Algorithm 1), and the process repeats itself.

A.3 Distributed Algorithms

In a distributed parallel PSO algorithm, individual processors not only perform evaluations of particles, but also their movement. The information for each particle is not held by a central machine that directs the algorithm; instead, each processor has the information for the particle or particles that it is in charge of and must perform the steps of the algorithm for those particles (McNabb et al., 2007). Messages such as values and positions for the neighborhood best are sent between processors. There may still be some machine that collects information from all of the particles and outputs the result of the algorithm, though that machine's importance is much less than in centralized algorithms.

To perform speculative evaluation in a distributed PSO algorithm, there must be some way to have processors evaluate the speculative children of particles, without giving the speculative particles the same treatment as actual particles, as the speculative children only live for one iteration. One way that can be done is by assigning each particle a set of machines instead of a single machine, and the particle directs its extra machines to evaluate its speculative children.

The same information needs to be passed between particles no matter the framework used. We describe here the messages each particle needs to receive to perform speculative evaluation.

A processor that is controlling a single particle p_{t-1} must first move the particle to p_t^{-e} and produce the particle's speculative children $\mathbf{s}_{t+1}^{-e}$. This is done in the same way as described above. However, in order to produce $\mathbf{s}_{t+1}^{-e}$, the processor needs information about the particle's neighbors, so there must be some message passing to get that information. Particularly, the information that the processor needs is the position of each of the particle's neighbors at iteration t.

To get that information, a round of message passing is required. Each particle sends its position to its neighbors at iteration t, so that all particles can generate $\mathbf{s}_{t+1}^{-e}$. After each particle evaluates its position and the positions of its speculative children, it passes information about the outcome of iteration t to its neighbors, so that neighboring particles can update their neighborhood bests to move from p_t^{-n} to p_t. Once that communication is finished, the particle can select the speculative child which matched the branch that iteration t actually produced. Then another round of information passing follows, for iteration $t+1$, so that p_{t+1}^{-n} can be updated to p_{t+1}. Two iterations have then been completed with only one round of evaluations, and the next iteration can start again with the first round of message passing.

In distributed frameworks, synchronizing all of the machines for a round of message passing can be expensive. The method just described uses three rounds of message passing for every two iterations (corresponding to steps 2, 5 and 10 in Algorithm 1). It is possible to perform speculative evaluation in PSO with only one round of communication per two iterations. However, the methodology is tedious and distracting from the present discussion, so we defer its description to Appendix B.

A.4 Dynamic Topologies

Performing speculative evaluation in PSO with a dynamic topology (where neighbors change from iteration to iteration) raises a sticky issue of its own. In a static topology, at iteration t a particle already has all of the information about the positions of its neighbors during iterations 1 through $t-1$. If the neighbor finds a better position at iteration t, the particle updates its neighborhood best, but if it does not, it still has its old neighborhood best from its neighbors for all previous iterations.

In a dynamic topology, a particle might not have information about the previous positions of its neighbors at iteration t. That means that its new neighborhood best could come not only from its neighbors' positions at iteration t, but also from their personal best from iteration $t-1$, as neighbors' personal bests are what are used to update a particle's neighborhood best. That creates a problem for speculative evaluation—there are potentially more than $2n+1$ possible next positions, increasing the amount of work that must be done to perform the second iteration at the same time as the first.

This is easily fixed by updating each new particle p_{t+2}^{-e} with the currently available information about its neighbors $\mathbf{n}_{t+2}^{-e}$ before producing its children $\mathbf{s}_{t+3}^{-e}$. If a particle p_{t+2}^{-e} updates its neighborhood best with the personal bests of $\mathbf{n}_{t+2}^{-e}$ before calculating the next possible positions for $\mathbf{s}_{t+3}^{-e}$, there are still only $2n+1$ possible next positions, and the problem is avoided.

Appendix B Alternate form of message passing

Here we describe a method that requires only one round of communication for each pair of iterations, which happens at step 5 of Algorithm 1. Many more messages are needed, but that is sometimes more desirable than synchronizing all of the machines three times.

This second method only requires one round of passing information because information about both iterations t and $t+1$ is passed at the same time. Each processor reconstructs from the messages it receives all of the information that it needs about its neighbors. Messages are passed directly after evaluating each particle and its children, so all messages are of the form of p_t^{-n} or s_{t+1}^{-n}. The first iteration needs to be treated specially, so each particle can produce its initial set of speculative children—neighbors need only pass their initial position. This kind of message passing necessitates the careful use of random seeds, so that when each processor

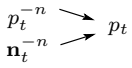

Fig. 18 The production of p_t from the original particle p_t^{-n} and the messages $\mathbf{n}_t^{-n}$.

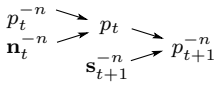

Fig. 19 The production of p_{t+1}^{-n} from the original particle p_t^{-n}, messages $\mathbf{n}_t^{-n}$ and $\mathbf{s}_{t+1}^{-n}$, and intermediate particles.

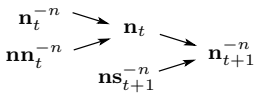

Fig. 20 The production of each n_{t+1}^{-n} from the original particle n_t^{-n}, messages $\mathbf{nn}_t^{-n}$ and $\mathbf{ns}_{t+1}^{-n}$, and intermediate particles. **nn** is the set of neighbors for each particle n, and **ns** is the set of n's speculative children. Note the similarity between this and Figure 19.

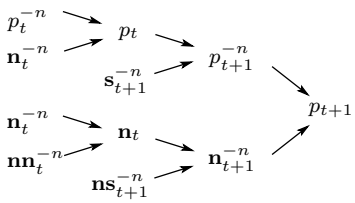

Fig. 21 The production of p_{t+1} from the original particle p_t^{-n}, messages $\mathbf{n}_t^{-n}$, $\mathbf{s}_{t+1}^{-n}$, $\mathbf{nn}_t^{-n}$, and $\mathbf{ns}_{t+1}^{-n}$, and intermediate particles. Note that this is just a combination of Figure 19 and Figure 20.

computes the motion equations for its neighbors it gets the same results as its neighboring processors.

With the results of evaluating p_t^{-e} and $\mathbf{s}_{t+1}^{-e}$, along with all of the required messages from neighboring particles, the goal is to produce p_{t+1} and output p_{t+2}^{-e} and $\mathbf{s}_{t+3}^{-e}$ ready to be evaluated for the next iteration. We first focus on the messages needed to produce p_{t+1}.

Upon evaluation, p_t^{-e} becomes p_t^{-n}, needing only to get its neighborhood best information from its neighbors. All of its neighbors, then, must send it a message, so that from their updated personal best at iteration t the particle becomes p_t. The work done with the messages received thus far is just as in regular PSO, and is graphically depicted in Figure 18.

With p_t we can select the correct speculative child as described above and produce p_{t+1}^{-n}. Again we show the use of messages thus far graphically, in Figure 19.

We then need the set of neighbors to p_{t+1}^{-n}, $\mathbf{n}_{t+1}^{-n}$, so we can update p_{t+1}^{-n}'s neighborhood best. To produce each neighbor n_{t+1}^{-n}, we need the same information for the neighboring particle that we needed to produce the original particle, p_{t+1}^{-n}; we need the original neighbor particle, its speculative children, and its neighbors. With that information, the set $\mathbf{n}_{t+1}^{-n}$ can be obtained by following the same process used to obtain p_{t+1}^{-n}. We graphically show the messages needed to produce $\mathbf{n}_{t+1}^{-n}$ in Figure 20. Note that it looks identical to Figure 19, just with different sets of particles.

With $\mathbf{n}_{t+1}^{-n}$ and p_{t+1}^{-n}, we can produce p_{t+1}. This is shown in Figure 21. Note that we just combined Figures 19 and 20, putting them together to make p_{t+1}, as all the particle needs is its neighborhood best to be updated.

In order to get p_{t+1}, then, a particle needs to receive messages from its neighbors, its neighbors' neighbors, its speculative children, and its neighbors' speculative children. The

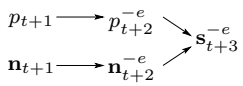

Fig. 22 The production of p^{-e}_{t+2} and $\mathbf{s}^{-e}_{t+3}$ from p_{t+1} and n_{t+1}, each of which are produced as in Figure 21.

particle p_{t+1} can be passed to some central machine to track the progress of the algorithm, and it can be moved to p^{-e}_{t+2} in order to start the next iteration.

The next goal is to produce the set $\mathbf{s}^{-e}_{t+3}$. As described above, the necessary components to produce $\mathbf{s}^{-e}_{t+3}$ are p^{-e}_{t+2} and the neighbors of p^{-e}_{t+2}, $\mathbf{n}^{-e}_{t+2}$. We already have p^{-e}_{t+2}, so what remains is to produce $\mathbf{n}^{-e}_{t+2}$. It is sufficient to obtain $\mathbf{n}_{t+1}$, as each neighbor particle n_{t+1} can be moved with the motion equations to n^{-e}_{t+2}.

We have already described how to use a set of messages to obtain p_{t+1}. The process is exactly the same to produce each n_{t+1}, requiring the same messages, only for the neighbor particles instead of the particle itself. Figure 22 shows graphically how $\mathbf{s}^{-e}_{t+3}$ is produced.

Having obtained both p^{-e}_{t+2} and $\mathbf{s}^{-e}_{t+3}$ from the messages received, the algorithm then moves to the evaluation phase, and the process repeats itself. The particles are evaluated, send their messages, and produce the next set of particles to be evaluated from the messages received.

To perform the entire process, at each message passing round a particle must receive messages from its neighbors, its neighbors' neighbors, its neighbors' neighbors' neighbors, its speculative children, its neighbors' speculative children, and its neighbors' neighbors' speculative children. With the Ring topology, that looks like more messages than it really is, as many of the neighbors' neighbors are duplicates. With the Random topology, however, the list of necessary messages could be rather large.

One more issue arises when dealing with dynamic topologies. With neighbors changing each iteration, messages that processors pass to their neighbors need to be sent to the correct neighbors for each iteration. A particle cannot simply send messages to its neighbors' neighbors' neighbors—it needs to send messages to its iteration t neighbors' iteration $t+1$ neighbors, and so on. For every neighbor outward information is sent, the iteration also needs to be incremented, as information about neighbors' neighbors is used during iteration $t+1$, and information about neighbors' neighbors' neighbors is used to reconstruct information about iteration $t+2$. Also, this method of message passing again requires the use of random seeds if the topology is random, so that each processor computes the same neighbors for a particle as all other processors.

This may seem like an inordinate amount of work, and with some distributed PSO frameworks it is. However, other parallel frameworks necessitate this type of message passing, so we have described how speculative evaluation can be performed in those circumstances.

References

Belal, M., & El-Ghazawi, T. (2004). Parallel models for particle swarm optimizers. *International Journal of Intelligent Computing and Information Sciences*, *4*(1), 100–111.

Beyer, H., & Schwefel, H. (2002). Evolution strategies—A comprehensive introduction. *Natural computing*, *1*(1), 3–52.

Bratton, D., & Kennedy, J. (2007). Defining a standard for particle swarm optimization. In *Proceedings of the IEEE Swarm Intelligence Symposium*, (pp. 120–127). Piscataway, NJ, USA: IEEE Press.

Chang, J.-F., Chu, S.-C., Roddick, J. F., & Pan, J.-S. (2005). A parallel particle swarm optimization algorithm with communication strategies. *Journal of Information Science and Engineering*, *21*, 809–818.

Chu, S.-C., & Pan, J.-S. (2006). Intelligent parallel particle swarm optimization algorithms. *Parallel Evolutionary Computations*, (pp. 159–175).

Clerc, M., & Kennedy, J. (2002). The particle swarm—explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, *6*(1), 58–73.

Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to Parallel Computing*. Harlow, England: Addison-Wesley, second ed.

Herrera, F., Lozano, M., & Molina, D. (2010). Test suite for the special issue of soft computing on scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems. Tech. rep., University of Granada.

Hsieh, S.-T., Sun, T.-Y., Liu, C.-C., & Tsai, S.-J. (2009). Efficient population utilization strategy for particle swarm optimizer. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, *39*(2), 444 –456.

Jin, N., & Rahmat-Samii, Y. (2005). Parallel particle swarm optimization and finite-difference time-domain (PSO/FDTD) algorithm for multiband and wide-band patch antenna designs. *IEEE Transactions on Antennas and Propogation*, *53*(11), 3459–3468.

Kennedy, J. (2003). Bare bones particle swarms. In *Proceedings of the IEEE Swarm Intelligence Symposium, 2003*, (pp. 80 – 87). Piscataway, NJ, USA: IEEE Press.

Kennedy, J., & Eberhart, R. C. (1995). Particle swarm optimization. In *International Conference on Neural Networks IV*, (pp. 1942–1948). Piscataway, NJ, USA: IEEE Press.

Koh, B.-I., George, A. D., Haftka, R. T., & Fregly, B. J. (2006). Parallel asynchronous particle swarm optimization. *International Journal of Numerical Methods in Engineering*, *67*, 578–595.

Li, J., Wang, X., He, R., & Chi, Z. (2007). An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, (pp. 855–862). Piscataway, NJ, USA: IEEE Press.

McNabb, A., Gardner, M., & Seppi, K. (2009). An exploration of topologies and communication in large particle swarms. In *Proceedings of the IEEE Congress on Evolutionary Computation*, (pp. 712–719). Piscataway, NJ, USA: IEEE Press.

McNabb, A. W., Monson, C. K., & Seppi, K. D. (2007). Parallel PSO using MapReduce. In *Proceedings of the IEEE Congress on Evolutionary Computation*, (pp. 7–14). Piscataway, NJ, USA: IEEE Press.

Mendes, R. (2004). *Population Topologies and Their Influence in Particle Swarm Performance*. Ph.D. thesis, Escola de Engenharia, Universidade do Minho, Guimaraes, Portugal.

Mendes, R., Kennedy, J., & Neves, J. (2004). The fully informed particle swarm: Simpler, maybe better. *IEEE Transactions on Evolutionary Computation*, *8*(3), 204–210.

Monson, C. K., & Seppi, K. D. (2005). Exposing origin-seeking bias in pso. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, vol. 1, (pp. 241–248). New York, NY, USA: ACM Press.

Montes de Oca, M. A., Stützle, T., Van den Enden, K., & Dorigo, M. (2010). Incremental social learning in particle swarms. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, *41*(2), 368–384.

Mostaghim, S., Branke, J., & Schmeck, H. (2006). Multi-objective particle swarm optimization on computer grids. Technical Report 502, AIFB Institute, Karlsruhe, Germany.

Parsopoulos, K. E., Tasoulis, D. K., & Vrahatis, M. N. (2004). Multiobjective optimization using parallel vector evaluated particle swarm optimization. In *Proceedings of the IASTED International Conference on Artificial Intelligence and Applications*, (pp. 823–828). Calgary, AB, Canada: IASTED/ACTA Press.

Poli, R. (2008a). Analysis of the publications on the applications of particle swarm optimisation. *Journal of Artificial Evolution and Applications*, *2008*, 3:1–3:10.

Poli, R. (2008b). Dynamics and stability of the sampling distribution of particle swarm optimisers via moment analysis. *Journal of Artificial Evolution and Applications*, *8*(2), 1–10.

Poli, R., Kennedy, J., & Blackwell, T. (2007). Particle swarm optimization. *Swarm intelligence*, *1*(1), 33–57.

Rudolph, G. (1991). Global optimization by means of distributed evolution strategies. In *Parallel Problem Solving from Nature*, (pp. 209–213). Berlin, Germany: Springer.

Schutte, J. F., Reinbolt, J. A., Fregly, B. J., Haftka, R. T., & George, A. D. (2004). Parallel global optimization with the particle swarm algorithm. *International Journal for Numerical Methods in Engineering*, *61*(13), 2296–2315.

Scriven, I., Ireland, D., Lewis, A., Mostaghim, S., & Branke, J. (2008a). Asynchronous multiple objective particle swarm optimisation in unreliable distributed environments. In *Proceedings of the IEEE Congress on Evolutionary Computation*, (pp. 2481–2486). Piscataway, NJ, USA: IEEE Press.

Scriven, I., Lewis, A., Ireland, D., & Lu, J. (2008b). Decentralised distributed multiple objective particle swarm optimisation using peer to peer networks. In *Proceedings of the IEEE Congress on Evolutionary Computation*, (pp. 2925–2928). Piscataway, NJ, USA: IEEE Press.

Venter, G., & Sobieszczanski-Sobieski, J. (2005). A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. In *Proceedings of the 6th World Congresses of Structural and Multidisciplinary Optimization*. Berlin, Germany: Springer.

Witte, E., Chamberlain, R., & Franklin, M. (1991). Parallel simulated annealing using speculative computation. *IEEE Transactions on Parallel and Distributed Systems*, *2*(4), 483–494.

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, *1*(1), 67–82.

Zhou, Y., & Tan, Y. (2009). GPU-based parallel particle swarm optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation*, (pp. 1493–1500). Piscataway, NJ, USA: IEEE Press.