Speculative Evaluation in Particle Swarm Optimization

Matthew Gardner, Andrew McNabb, and Kevin Seppi

Department of Computer Science, Brigham Young University

Abstract. Particle swarm optimization (PSO) has previously been parallelized only by adding more particles to the swarm or by parallelizing the evaluation of the objective function. However, some functions are more efficiently optimized with more iterations and fewer particles. Accordingly, we take inspiration from speculative execution performed in modern processors and propose speculative evaluation in PSO (SEPSO). Future positions of the particles are speculated and evaluated in parallel with current positions, performing two iterations of PSO at once.
We also propose another way of making use of these speculative particles, keeping the best position found instead of the position that PSO actually would have taken. We show that for a number of functions, speculative evaluation gives dramatic improvements over adding additional particles to the swarm.

1 Introduction

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

Within the last few years, researchers have begun to recognize the need to develop parallel implementations of PSO, publishing many papers on the subject. The methods they have used include various synchronous algorithms [1, 2] and asynchronous algorithms [3–5]. Parallelizing the evaluation of the objective function can also be done in some cases, though that is not an adaption of the PSO algorithm itself and thus is not the focus of this paper.

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

For many functions there comes a point of diminishing returns with respect to adding particles. Very small swarms do not produce enough exploration to consistently find good values, while large swarms result in more exploration than is necessary and waste computation. For this reason, previous work has recommended the use of a swarm size of 50 for PSO [6]. Thus, in at least some cases, adding particles indefinitely will not yield an efficient implementation.

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

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

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

The balance of this paper is organized as follows. Section 2 describes the particle swarm optimization algorithm. Section 3 describes how speculative evaluation can be done in parallel PSO to perform two iterations at once. In Section 4 and Section 5 we present our results and conclude.

2 Particle Swarm Optimization

Particle swarm optimization was proposed in 1995 by James Kennedy and Russell Eberhart [7]. This social algorithm, inspired by the flocking behavior of birds, is used to quickly and consistently find the global optimum of a given objective function in a multi-dimensional space.

The motion of particles through the search space has three components: an inertial component that gives particles momentum as they move, a cognitive component where particles remember the best solution they have found and are

attracted back to that place, and a social component by which particles are attracted to the best solution that any of their neighbors have found.

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

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

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \boldsymbol{v}_{t+1} \quad (2)$$

where U_t^P and U_t^N are vectors of independent random numbers drawn from a standard uniform distribution, the $\otimes$ operator is an element-wise vector multiplication, $\boldsymbol{x}^P$ (called personal best) is the best position the current particle has seen, and $\boldsymbol{x}^N$ (called neighborhood best) is the best position the neighbors of the current particle have seen. The parameters ϕ^N, ϕ^P, and χ are given prescribed values required to ensure convergence (2.05, 2.05, and .73, respectively) [8].

Changing the way neighbors are defined, usually called the "topology," has a significant effect on the performance of the algorithm. In the Ring topology, each particle has one neighbor to either side of it; in the Complete topology, every particle is a neighbor to every other particle [6]. 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, $\boldsymbol{x}^N$. Thus with p particles, using the Ring topology each particle with index i has three neighbors: $i-1$, i (itself), and $i+1$. With the Complete topology, each particle has p neighbors.

In this paper we use these topologies as well as a parallel adaptation of the Complete topology, called Random, that has been shown to approximate the behavior of Complete with far less communication [9]. 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.

3 Speculative Evaluation in PSO

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

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

The refactoring hinges on the idea that once the random coefficients U_t^P and U_t^P are determined, there are only a few possible updates to $\boldsymbol{x}^N$ and $\boldsymbol{x}^P$. 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 $\boldsymbol{x}^N$ coming from the left neighbor, R represents an update to $\boldsymbol{x}^N$ coming from the right neighbor, and S represents an update to either $\boldsymbol{x}^P$ or $\boldsymbol{x}^N$ coming from the particle itself. As an example, $(S,-)$ refers to the case that the particle finds a new personal best, but neither it nor its neighbors found a position that updated its neighborhood best. In the equations that follow, we refer to an unspecified update case as c, and to the set of cases collectively as $\mathcal{C}$.

Table 1. All possible updates for a particle with two neighbors

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

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

$$I^{(S,-)}_{t+1}\left(f(\boldsymbol{x}_t), f\left(\boldsymbol{x}^L_t\right), f\left(\boldsymbol{x}^R_t\right), f\left(\boldsymbol{x}^P_{t-1}\right), f\left(\boldsymbol{x}^N_{t-1}\right)\right) = \begin{cases} 1 & \text{if } f(\boldsymbol{x}_t) < f\left(\boldsymbol{x}^P_{t-1}\right) \text{ and } f\left(\boldsymbol{x}^N_{t-1}\right) < \min\left(f(\boldsymbol{x}_t), f\left(\boldsymbol{x}^L_t\right), f\left(\boldsymbol{x}^R_t\right)\right) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

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

$$\boldsymbol{V}^{(S,-)}_{t+1}\left(\boldsymbol{v}_t, \boldsymbol{x}_t, \boldsymbol{x}^L_t, \boldsymbol{x}^R_t, \boldsymbol{x}^P_{t-1}, \boldsymbol{x}^N_{t-1}, U^P_t, U^N_t\right) = \chi\left[\boldsymbol{v}_t + \phi^P U^P_t \otimes (\boldsymbol{x}_t - \boldsymbol{x}_t) + \phi^N U^N_t \otimes \left(\boldsymbol{x}^N_{t-1} - \boldsymbol{x}_t\right)\right] \tag{4}$$

In the same way we can create notation for the position update function by substituting into (2):

$$\begin{aligned}\boldsymbol{X}^c_{t+1}&\left(\boldsymbol{x}_t, \boldsymbol{v}_t, \boldsymbol{x}^L_t, \boldsymbol{x}^R_t, \boldsymbol{x}^P_{t-1}, \boldsymbol{x}^N_{t-1}, U^P_t, U^N_t\right)\\ &= \boldsymbol{x}_t + \boldsymbol{V}^c_{t+1}\left(\boldsymbol{v}_t, \boldsymbol{x}_t, \boldsymbol{x}^L_t, \boldsymbol{x}^R_t, \boldsymbol{x}^P_{t-1}, \boldsymbol{x}^N_{t-1}, U^P_t, U^N_t\right) \qquad (5)\end{aligned}$$

With this notation we can re-write the original PSO velocity equation (1), introducing our sum over cases with the indicator functions. The velocity (1) and position (2) equations become:

$$\begin{aligned}\boldsymbol{v}_{t+1} = \sum_{c\in\mathcal{C}} &\left[I^c_{t+1}\left(f(\boldsymbol{x}_t), f\left(\boldsymbol{x}^L_t\right), f\left(\boldsymbol{x}^R_t\right), f\left(\boldsymbol{x}^P_{t-1}\right), f\left(\boldsymbol{x}^N_{t-1}\right)\right)\right.\\ &\left.\boldsymbol{V}^c_{t+1}\left(\boldsymbol{x}_t, \boldsymbol{v}_t, \boldsymbol{x}^L_t, \boldsymbol{x}^R_t, \boldsymbol{x}^P_{t-1}, \boldsymbol{x}^N_{t-1}, U^P_t, U^N_t\right)\right] \qquad (6)\\ \boldsymbol{x}_{t+1} = \sum_{c\in\mathcal{C}} &\left[I^c_{t+1}\left(f(\boldsymbol{x}_t), f\left(\boldsymbol{x}^L_t\right), f\left(\boldsymbol{x}^R_t\right), f\left(\boldsymbol{x}^P_{t-1}\right), f\left(\boldsymbol{x}^N_{t-1}\right)\right)\right.\\ &\left.\boldsymbol{X}^c_{t+1}\left(\boldsymbol{x}_t, \boldsymbol{v}_t, \boldsymbol{x}^L_t, \boldsymbol{x}^R_t, \boldsymbol{x}^P_{t-1}, \boldsymbol{x}^N_{t-1}, U^P_t, U^N_t\right)\right] \qquad (7)\end{aligned}$$

In this form the important point to notice is that there are only 7 values (for this Ring topology) in the set $\{\boldsymbol{X}^c_{t+1} : c \in \mathcal{C}\}$ and that none of them depend upon $f(\boldsymbol{x}_t)$ 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* $f(\boldsymbol{x}_t)$ is computed. Once the evaluation of $f(\boldsymbol{x}_t)$ 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 $\boldsymbol{X}^c_{t+1}$ will be kept.

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

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

In order to do two iterations at once, we use 8 times as many processors as there are particles in the swarm. If these processors were not performing speculative evaluation, they might instead be used for function evaluation needed to support a large swarm. This raises the question of whether a swarm of 100 particles doing twice as many iterations outperforms a swarm of 800 particles. We show in the rest of this paper that in many instances a smaller swarm performing

more iterations does in fact outperform a larger swarm. We acknowledge that both intuition and prior research [9] indicate that the optimization of deceptive functions benefits greatly from large and even very large swarm sizes. Thus this work will focus on less deceptive functions.

3.1 Implementation

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

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

A particle at iteration $t - 1$ that has been moved to iteration t using (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.

The outline of the speculative evaluation in PSO (SEPSO) algorithm is given in Algorithm 1.

Algorithm 1 Speculative Evaluation PSO (SEPSO)

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 (7).
8: Pass along personal and neighborhood best values obtained by p_t, making p_{t+1}^{-n}
9: **end for**
10: Update neighborhood best for each p_{t+1}^{-n}, creating $\mathbf{p}_{t+1}$
11: Repeat from Step 1 until finished

3.2 Using All Speculative Evaluations

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

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

4 Results

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

In each set of experiments we keep the number of processors for each algorithm constant. In all of our experiments SEPSO uses topologies in which each particle has two neighbors in addition to itself. As shown in Table 1, this results in 7 speculative evaluations per particle. With one evaluation needed for the original, non-speculative particle, we have a total of $8p$ evaluations for every two iterations, where p is the number of particles in the speculative swarm. In order to use the same resources for each algorithm, we compare swarms of size p in the speculative algorithms with swarms of size $8p$ in standard PSO.

As discussed in Section 3.1, where the Complete topology would normally be used, we use a Random topology in our speculative algorithm, as Complete leads to an explosion of speculative evaluations. For the standard PSO baseline we have included experiments with the Ring and the Random topologies, both with two neighbors, as well as for the Complete topology. It is important to note however, that in many cases the Complete topology is not a practical alternative in the context of large swarms on large clusters where where the messaging complexity is $O(p^2)$ and can overwhelm the system.

We follow the experimental setup used in [6]. All functions have 20 dimensions, and all results shown are averages over 20 runs. For ease of labeling, we call

our speculative algorithm SEPSO, the variant SEPSO Pick Best, and standard PSO just PSO and identify the topology in a suffix, "PSO Ring" for example.

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

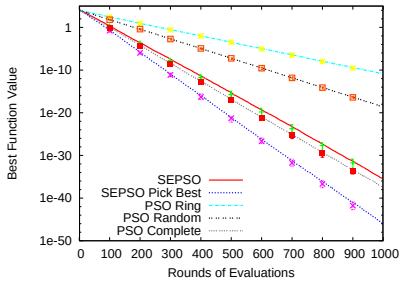

Fig. 1. Function Sphere with a swarm that uses 240 processors per round of evaluations. We show 10th and 90th percentiles every 100 iterations. Note that PSO Complete requires $O(p^2)$ messaging and may not be practical in many cases.

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

PSO and SEPSO get caught in local minima with small swarm sizes so we show results in Figure 2 for swarms of size 100 (SEPSO) and 800 (standard) using the Ring topology. Figure 2 shows that SEPSO quickly finds the global optimum, between two and three times faster than running standard PSO.

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

when using SEPSO Pick Best with the same swarm size, the algorithm finds the optimum every time.

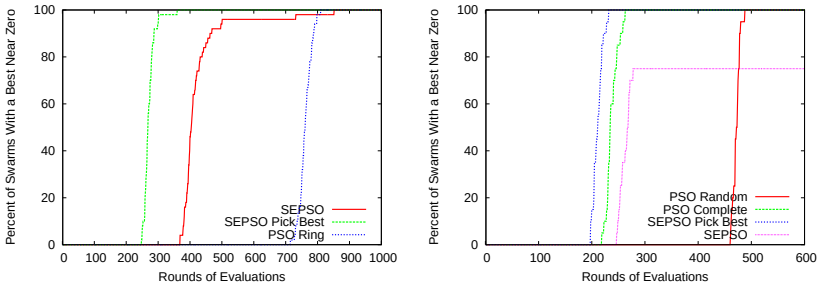

Fig. 2. Function Griewank with a swarm that uses 800 processors per round of evaluations, and function Bohachevsky with a swarm that uses 480 processors per round of evaluations.

Previous work has shown that the optimization of highly deceptive functions like Rastrigin ($f(\boldsymbol{x}) = \sum_{i=1}^{D} \left(x_i^2 - 10\cos\left(2\pi x_i\right) + 10\right)$) benefit greatly from the addition of particles. Smaller swarms get caught in local optima, up to swarms of at least 4000 particles [9]. Because our speculative algorithms have a significantly smaller swarm size, they get stuck at higher values while the larger swarms performing regular PSO continue to improve the best value found. Our experiments with SEPSO on Rastrigin were predicably lack luster, yielding an average value of 31 after 1000 evaluations, as compared to 10 for standard PSO.

5 Conclusions

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

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

There are other functions for which it seems there is never enough exploration, such as the Rastrigin function. It has been shown that up to 4000 particles there is no point at which "enough" exploration has been done [9]. With

such functions, the smaller swarm size required by speculative evaluation is not able to produce enough exploration to perform better than standard PSO.

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

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

References

1. M. Belal and T. El-Ghazawi. Parallel models for particle swarm optimizers. *Intl. Journal of Intelligent Computing and Information Sciences*, 4(1):100–111, 2004.
2. J. F. Schutte, J. A. Reinbolt, B. J. Fregly, R. T. Haftka, and A. D. George. Parallel global optimization with the particle swarm algorithm. *International Journal for Numerical Methods in Engineering*, 61(13):2296–2315, December 2004.
3. Sanaz Mostaghim, Jürgen Branke, and Hartmut Schmeck. Multi-objective particle swarm optimization on computer grids. Technical Report 502, AIFB Institute, DEC 2006.
4. Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. In *Proceedings of the 6th World Congresses of Structural and Multidisciplinary Optimization*, 2005.
5. Byung-Il Koh, Alan D. George, Raphael T. Haftka, and Benjamin J. Fregly. Parallel asynchronous particle swarm optimization. *International Journal of Numerical Methods in Engineering*, 67:578–595, 2006.
6. D. Bratton and J. Kennedy. Defining a standard for particle swarm optimization. In *Proceedings of the IEEE Swarm Intelligence Symposium*, pages 120–127, 2007.
7. James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *International Conference on Neural Networks IV*, pages 1942–1948, Piscataway, NJ, 1995.
8. Maurice Clerc and James Kennedy. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.
9. Andrew McNabb, Matthew Gardner, and Kevin Seppi. An exploration of topologies and communication in large particle swarms. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 712–719, May 2009.