2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

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

Andrew McNabb and Kevin Seppi

*Abstract*— **From multi-core processors to parallel GPUs to computing clusters, computing resources are increasingly parallel. These parallel resources are being used to address increasingly challenging applications. This presents an opportunity to design optimization algorithms that use parallel processors efficiently. In spite of the intuitively parallel nature of Particle Swarm Optimization (PSO), most PSO variants are not evaluated from a parallel perspective and introduce extra communication and bottlenecks that are inefficient in a parallel environment. We argue that the standard practice of evaluating a PSO variant by reporting function values with respect to the number of function evaluations is inadequate for evaluating PSO in a parallel environment. Evaluating the parallel performance of a PSO variant instead requires reporting function values with respect to the number of iterations to show how the algorithm scales with the number of processors, along with an implementation-independent description of task interactions and communication. Furthermore, it is important to acknowledge the dependence of performance on specific properties of the objective function and computational resources. We discuss parallel evaluation of PSO, and we review approaches for increasing concurrency and for reducing communication which should be considered when discussing the scalability of a PSO variant. This discussion is essential both for designers who are defending the performance of an algorithm and for practitioners who are determining how to apply PSO for a given objective function and parallel environment.**

## I. Introduction

Despite the overwhelming parallel nature of modern hardware, contributions to Particle Swarm Optimization (PSO) are still evaluated from a purely serial perspective. Variants to PSO may improve performance in a serial environment but worsen performance in a parallel environment. For example, adaptive topologies such as ARPSO [1] ensure population diversity, but adaptation usually requires global information about the swarm and either reduces concurrency or increases communication costs. Researchers must understand these issues to evaluate PSO parameters and variants, and practitioners must understand them to apply PSO effectively.

For the case of serial PSO, Bratton and Kennedy [2] addressed common issues such as swarm size and motion equations in order to establish a common starting point for PSO, but these conclusions do not directly apply to parallel PSO. While practical approaches to parallel PSO are available, most PSO research does not consider issues raised by parallel computation, such as communication and scaling the number of processors. We are not aware of any general attempts to address the implementation-independent consequences of parallelization. Now that computing clusters, multicore processors, and powerful GPUs are commonplace, the behavior of parallel PSO must be a primary rather than a secondary concern.

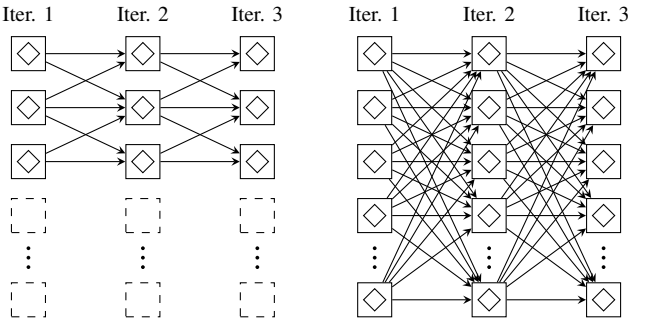
Andrew McNabb and Kevin Seppi are with the Department of Computer Science, Brigham Young University, 3361 TMCB, Provo, UT 84602 (phone: 801-422-8717; email: {a,k}@cs.byu.edu).

We organize the issues raised by PSO in a parallel environment into two categories, processor scaling and task interaction, which both limit the degree of concurrency, the number of tasks that can be computed simultaneously [3]. First, a PSO variant may scale poorly as the number of processors increases, independent of communication. For example, if the maximum degree of concurrency is less than the total number of processors, then the average degree of concurrency is limited because the processors are never all active. Different ways to increase the number of concurrent tasks, such as increasing the swarm size, are not necessarily equally effective. Second, interaction between tasks limits the average degree of concurrency because tasks remain idle while waiting for communication. Additionally, communication can affect performance if non-local information, if available, would make the work more effective. Issues such as motion equations and topology determine the dependencies and communication between tasks. Figure 1 illustrates how maximum degree of concurrency and task interaction affect performance. In Figure 1a, the number of tasks is smaller than the number of processors, so several processors are unused. In Figure 1b, every task in one iteration interacts with every task in the next iteration, requiring a high level of communication. In a task dependency diagram, the maximum degree of concurrency is given by the number of tasks in a column, and the task interactions are given by the arrows.

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

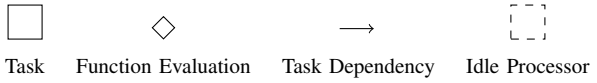
The paper proceeds as follows. Section II introduces parallel Particles Swarm Optimization by describing PSO, considering the relationship between its performance and the objective function, and reviewing approaches for the parallelization of PSO. The next two sections discuss the issues that limit the efficiency of parallel PSO. Section III

978-1-4799-1488-3/14/$31.00 ©2014 IEEE




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

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




(c) Legend

Fig. 1: Task dependencies for parallel PSO in two simple inefficient cases.

considers scaling of processors independent of communication and ways to improve it. Section IV considers the task interactions and communication introduced by a PSO variant in parallel and ways to reduce bottlenecks and overhead. Finally, Section V concludes with a plea to consider this parallel perspective when evaluating all variants of PSO.

## II. Parallel PSO

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

### A. Particle Swarm Optimization

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

Constricted PSO is generally considered the standard variant [2]. Each particle's position $\boldsymbol{x}_0$ and velocity $\boldsymbol{v}_0$ are initialized to random values based on a function-specific feasible region. During iteration $t$, the following equations update the $i^{\text{th}}$ component of a particle's position $\boldsymbol{x}_t$ and velocity $\boldsymbol{v}_t$ with respect to the personal best $\boldsymbol{x}^P_{t-1}$ and neighborhood best $\boldsymbol{x}^N_{t-1}$ from the preceding iteration:

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

$$x_{t,i} = x_{t-1,i} + v_{t,i} \quad (2)$$

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

The neighborhoods within a particle swarm are defined by the swarm topology, also known as the sociometry. The choice of topology can have a significant effect on the performance of PSO [5]. Topologies also determine the amount of communication between particles, which is especially important for parallel implementations of PSO.

### B. Objective Functions

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

The ideal motion equations, topology, and swarm size for PSO depend on the objective function. Under various benchmark functions, the ideal topology for one function may perform very poorly for another function. The No Free Lunch Theorems for Optimization show that this is true in general—if an algorithm performs well on average for one class of functions then it must do poorly on average for other problems [6].

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

As there are inexpensive functions with complex landscapes and expensive functions with simple landscapes, the behavior

of PSO with respect to the function is the main issue. For this purpose, benchmark functions are a useful and efficient tool for understanding the effects of PSO with expensive objective functions even though the benchmark functions are themselves inexpensive. For example, a plot of performance with respect to iterations for a smooth unimodal function with five minutes per evaluation would be similar to that of Sphere.

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

### C. Parallelization of PSO

In order for an algorithm to be parallelized, its operations must be decomposed into tasks [3]. The most fine-grained decompositions that are generally possible with PSO correspond to a task for each evaluation of the objective function. In specific cases, it may be possible to decompose the objective function itself, but we limit the discussion and the term "parallel PSO" to approaches that work for arbitrary objective functions. Even in the case of a parallelized objective function, it may be beneficial to also use parallel PSO.

Particle Swarm Optimization is usually decomposed into a task for each function evaluation, with one task per particle per iteration [7, 8, 9]. However, this is not always the case, and parallel PSO may take several different forms depending on what work constitutes each task, which mapping technique is used, and how the tasks interact. Some of these choices may significantly affect the behavior of PSO, while others are implementation details that may affect parallel performance and scalability but do not require rethinking at the PSO level. For example, tasks may include the work of a single particle or multiple particles; combining several particles into a single task can reduce communication given an appropriate choice of swarm topology. Likewise, tasks may include only the function evaluation, with each particle's position, velocity, and neighborhood best updated in serial on a centralized master [9], or tasks may include the full particle update, with position, velocity, and neighborhood best updates performed in parallel [8].

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

For extremely inexpensive objective functions, graphics processing units (GPUs) are an attractive platform for parallelization, but they are not applicable in all situations. GPUs are extremely fast at performing floating point operations, and GPU-based implementations of PSO can improve performance by an order of magnitude compared to a single processor [10]. GPUs perform well for objective functions that are floating point heavy, but they are less effective for functions relying on integer operations or large amounts of data. Implementations of PSO that use texture mapping on GPUs additionally require independence between variables [11]. General-purpose parallel architectures, such as CUDA and OpenCL, offer a more flexible approach to GPU-based parallel PSO [10].

In parallel PSO, tasks can be distributed to processors with either dynamic mapping or static mapping techniques. With dynamic mapping, tasks are distributed at runtime, often with a centralized scheduler. With static mapping, each particle is fixed to a specific processor, which performs all updates to its state. Fully distributed implementations may even use peer-to-peer networks for communication between particles [12]. Assuming that position, velocity, and neighborhood best updates all occur in serial on a centralized processor gives a poor understanding of behavior for a more distributed implementation. In general, it is safe to make a simplifying assumption that parallel PSO is fully distributed.

## III. Processor Scaling Independent of Communication

The ideal case where communication is free separates the issue of how an algorithm takes advantage of processors from the issue of task interaction. As the number of processors increases, the maximum degree of concurrency—the maximum number of tasks that can be computed simultaneously—must increase accordingly, or the additional processors are never utilized. In a task-dependency graph such as noted earlier in Figure 1, the maximum degree of concurrency is represented by the number of rows. While it is important to be able to increase the maximum degree of concurrency as the number of processors increases, it is also important to do so in the most effective way possible. A significant challenge for parallel PSO is to have as high of a marginal improvement in performance as possible as the number of processors increases.

Evaluating PSO algorithms as the number of processors increases or comparing PSO variants at a fixed number of processors is not possible from the traditional serial perspective. PSO algorithms are usually evaluated on their performance with respect to the number of function evaluations, but this is not appropriate for parallel PSO, where function evaluations are performed concurrently. Furthermore, criteria such as speedup or wallclock time per iteration are only appropriate for evaluating a specific implementation of parallel PSO because these measures are highly implementation-dependent. In a parallel context, the number of function evaluations divided by the number of processors is a much better

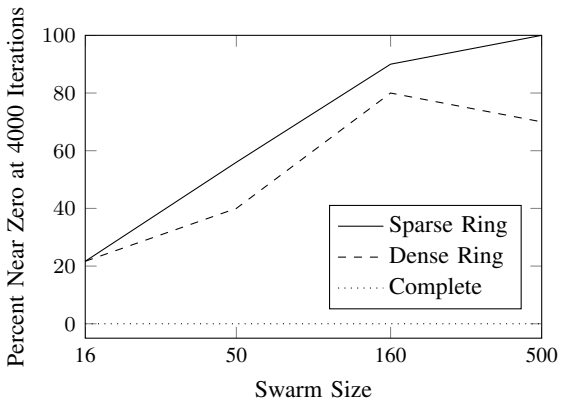



Fig. 2: Percent of swarms attaining a value near the global optimum of the Griewank function at a fixed number of iterations. In the sparse ring topologies, each particle is connected to 2 neighbors; in the dense ring topologies, each particle is connected to 20% of the swarm; and in the complete topologies,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

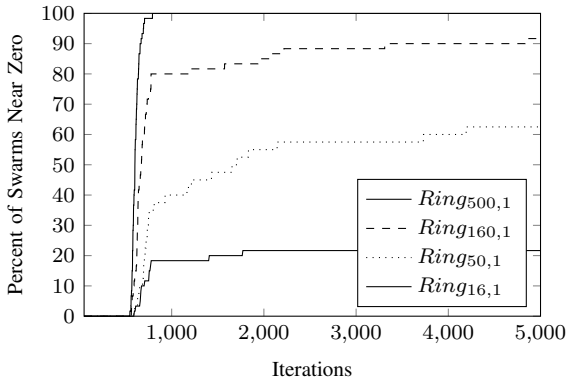



Fig. 3: Success rate on Griewank with respect to iterations for $Ring_{n,1}$ with various swarm sizes.

implementation-independent scale with which to measure performance. This is simply the number of iterations when there is one particle per processor, and in many other realistic cases, it is a multiple of the number of iterations. Comparing the performance of PSO with respect to the number of iterations as the number of processors increases shows the scaling behavior, which may in turn depend on the specific strategy for employing additional processors.

There are multiple ways to employ additional processors, and their effectiveness depends on the objective function and computational resources. For example, Figure 2 shows the success rate of three different topologies as the number of processors increases. For this particular objective function, the more densely connected swarms scale poorly with the number of processors compared to the more sparsely connected swarms. In the remainder of this section, we consider three approaches for increasing the maximum degree of concurrency of parallel PSO: independent runs, increased swarm size, and speculative evaluation.

### A. Independent Runs

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

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

### B. Swarm Size

Increasing the swarm size is generally a much more effective use of parallel resources than simply running multiple serial copies of PSO. Unfortunately, many PSO variants have not been tested at multiple swarm sizes to determine how well they scale. Serial PSO is typically used with small swarms of about 50 particles, with slightly larger or smaller swarm sizes as indicated by customized testing for the specific objective function [2]. We review the general effects of increasing the swarm size and encourage testing the effects of swarm size in all evaluation of PSO variants.

All else equal, an increase in swarm size increases the exploration of PSO and decreases its variability between runs. If a particularly multimodal objective function requires more particles for adequate exploration than the number of available processors, then this simultaneously increases the maximum degree of concurrency. Small swarms converge prematurely for the Griewank and Rastrigin functions. In parallel PSO, where performance is plotted against iterations instead of function evaluations, the benefit of large swarms is even more pronounced. Figure 3 and Figure 4 show that for Griewank and Rastrigin respectively, large swarms are not only less prone to premature convergence, but in parallel PSO they also attain comparable values in fewer iterations. Results are similar for other extremely multimodal functions but are omitted for space.

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

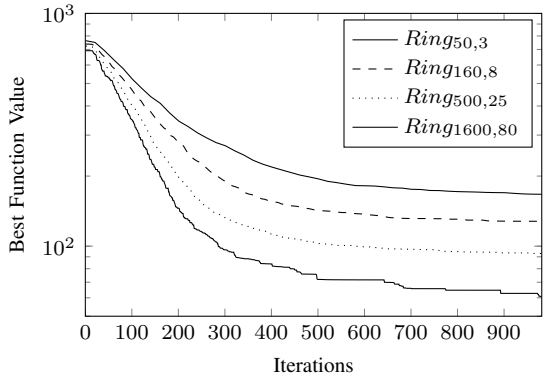



Fig. 4: PSO performance on Rastrigin with respect to iterations with various topologies.

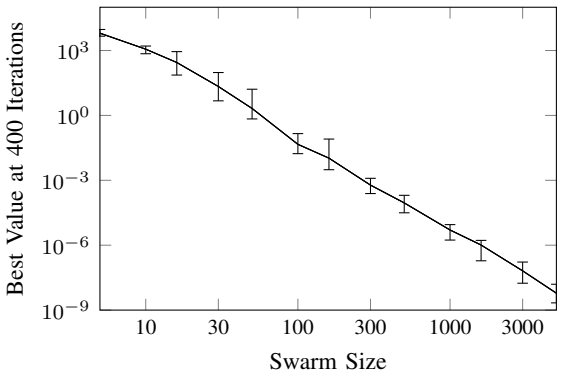



Fig. 5: Performance of PSO for the Sphere function with a complete topology at various swarm sizes. If communication is free and additional processors are available, increasing the swarm size is always beneficial, even in the extreme case of a unimodal function.

Although the ideal swarm size for the Sphere function in serial PSO is about 30 particles, Figure 5 shows that a greater swarm size always gives improvements in parallel PSO if additional processors are available. However, the marginal improvement diminishes as the swarm size increases (note the log scale), so it is important to find PSO variants that scale as well as possible. For example, organizing a large swarm of particles into subswarms is generally more effective than organizing the swarm into a large ring [13].

### C. Speculative Evaluation

It seems intuitively obvious that if the number of particles is less than the number of processors, then some of those processors would be unused at each iteration. However, the work associated with a particle can be split into multiple tasks by reorganizing the work performed in consecutive iterations of PSO. In this manner, speculative evaluation allows PSO to perform two iterations concurrently [14]. Figure 6 depicts the task dependencies of PSO with speculative evaluation. At the cost of using multiple processors for each particle, and thus requiring the number of particles to be fewer than the number of processors, PSO with speculative evaluation can reach

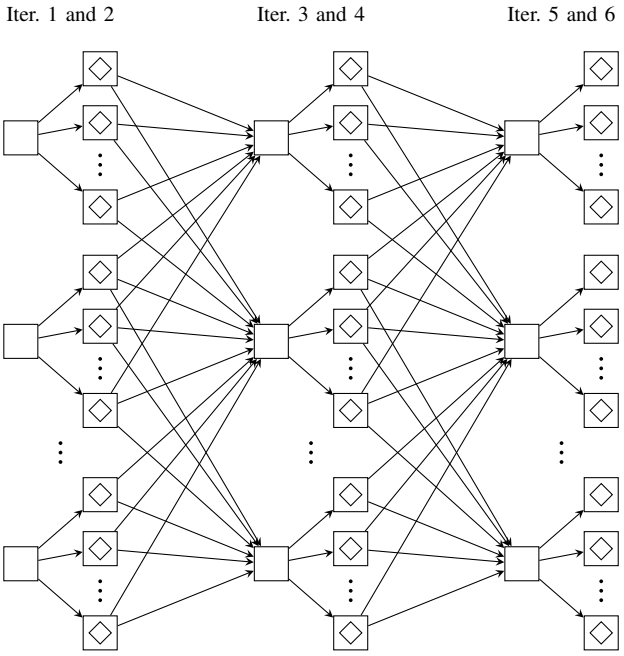



Fig. 6: Task dependency diagram for parallel PSO with speculative evaluation.

values much more quickly for appropriate objective functions. Where premature convergence is a concern, increasing swarm size may be a more valuable use of resources, but if exploration is adequate, speculative evaluation can halve the runtime of parallel PSO.

## IV. Task Interaction and Communication

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

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

### A. Sparse Topologies

The swarm topology determines which particles communicate at each iteration. Sparse topologies require less communication than dense topologies. PSO variants and topologies should be evaluated by the average number and size of messages per iteration that must be sent by each particle. For example, in a swarm of $n$ particles, a ring

topology requires 2 messages per particle per iteration, while a complete topology requires $n-1$ messages per particle per iteration. For standard PSO, the cost of messages is dominated by their number rather than size. Motion equations or dynamic topologies which require global information about the swarm introduce task interactions that are equivalent to using a complete topology, as in Figure 1b. Variants of PSO that require centralized coordination, or equivalently, communication between every pair of particles at each iteration, are not generally practical for parallel computation.

In some cases where heavy interaction shows improved results for serial PSO, it may be possible to make adaptations to be more appropriate for parallel PSO. For example, for some functions, serial PSO performs better with a fully connected topology than with a sparse topology such as a ring. By changing the motion equations to pass along the best value from any neighbor rather than the best value seen by the particle itself, sparse topologies behave like dense topologies by spreading information quickly through the swarm. In this case, communicating with 2 neighbors chosen randomly at each iteration requires only 2 messages per particle per iteration and gives almost the same performance as the complete topology, which requires a much higher overhead of $n-1$ messages per particle per iteration [15].

### B. Subswarms

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

Most attempts at subswarms for PSO have introduced sophisticated procedures for migrating particles between subswarms as with islands in genetic algorithms [16, 17], but these approaches require centralized coordination that increases communication and idleness. In contrast, the apiary topology [13] achieves similar or better results using the standard mechanism of swarm topology. Thus, communication follows the same structure as ordinary parallel PSO, as shown in the task-dependency graph in Figure 7. Likewise, cooperative micro-PSO is well-suited to parallel swarms with a similar communication structure [18]. Note that the number of particles per subswarm and the number of iterations per task determine the task granularity, so it is straightforward to adapt these parameters according to the relative cost of communication. If the topology between subswarms requires $k$ messages per subswarm, and if communication between subswarms occurs every $m$ iterations, then the average number of messages per subswarm per iteration is $k/m$.

### C. Synchronous and Asynchronous Parallel PSO

Asynchronous parallel PSO [19, 20] is a modification to the standard algorithm which removes the synchronization point at the end of each iteration. At each iteration of PSO, each particle must update its neighborhood best. This calculation requires the position and the result of the function evaluation from each of its neighbors. Synchronous parallel PSO, which exactly reproduces the computations of serial PSO, requires that computation associated with a particle wait until the results from the previous iteration are available from all of its neighbors. However, in asynchronous parallel PSO, particles iterate independently and communicate asynchronously. If a particle is ready to update its neighborhood best but has not received information about all of its neighbors, it may use information from the previous iteration.[1] Figure 8 shows task dependency diagrams for synchronous and asynchronous parallel PSO. Asynchronous iteration is particularly beneficial in situations such as a cluster with heterogeneous processors, an objective function with varying evaluation times, or a cluster with a large number of processors.

There are a few slightly different variants of asynchronous parallel PSO. In a partially asynchronous implementation, particles might wait for some but not all neighbors to complete before proceeding [21]. In some master-slave implementations, particles never get more than one iteration ahead of others [19,

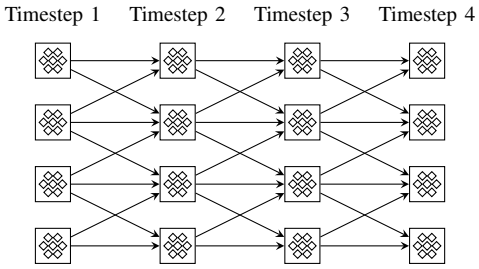
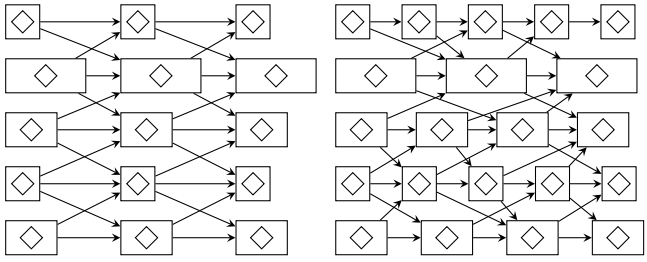


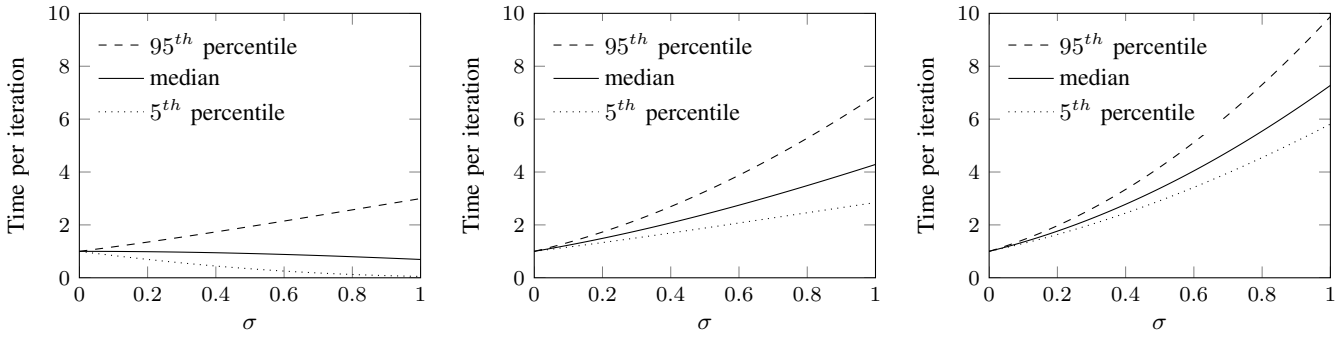

Fig. 7: Task-dependency graphs for parallel PSO with subswarms The squares represent tasks, and the diamonds represent function evaluations (with multiple particles and iterations in each timestep).



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

Fig. 8: Task dependency diagrams for synchronous and asynchronous parallel PSO with heterogeneous processors. In this particular example, asynchronous parallel PSO performs 21 function evaluations in the same time that synchronous parallel PSO performs 15 evaluations.

[1] Asynchronous parallel PSO has been compared to the "asynchronous updates" variant of serial PSO [20]. However, serial PSO with asynchronous updates differs from standard PSO in that particles use newer information, but asynchronous parallel PSO differs from standard PSO in that particles use older information.




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

Fig. 9: Probability distributions of the time per iteration (i.e., maximum task time) for synchronous parallel PSO. Individual task times are i.i.d. with a Gamma distribution of mean 1 and varying standard deviations.

20]. However, in a fully distributed implementation, particles might never wait for information, and one particle could complete many more iterations than another particle [12].

Synchronous and asynchronous parallel PSO are both valuable approaches. The benefits of the synchronous PSO include its simplicity, repeatability, and comparability with standard PSO, which may be essential in research applications. If the evaluation time varies significantly or if processors are heterogeneous, then asynchronous parallel PSO may provide a significant performance improvement over synchronous parallel PSO [19, 20]. However, its slower communication can make asynchronous parallel PSO require more iterations to converge. When evaluation times are consistent and processors are homogeneous, synchronous and asynchronous parallel PSO are comparable with respect to time [21]. Choosing between synchronous and asynchronous parallel PSO is a tradeoff between maximizing the number of function evaluations and having the location of the particles better informed.

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

This statistical result shows how the cost of synchronous parallel PSO increases with the number of processors. We illustrate this with the case where task times are Gamma distributed with an expected value of 1 second and a standard deviation of $\sigma$. Thus, using the inverse-scale parameterization of the Gamma, $X_i \sim \text{Gamma}(\frac{1}{\sigma^2}, \frac{1}{\sigma^2})$. This distribution is shown for varying values of $\sigma$ in Figure 9a. For asynchronous parallel PSO, this represents the time for each task and has an expected value of 1. For synchronous parallel PSO, the time required for each task is effectively lengthened by the need to wait for all other tasks in the same iteration. The distribution over the slowest task time, $\max X_i$, is shown for 50 processors in Figure 9b and for 1000 processors in Figure 9c with varying values of $\sigma$. Similar plots can be made for any distribution with a known c.d.f. and can even be approximated from a set of empirical samples.

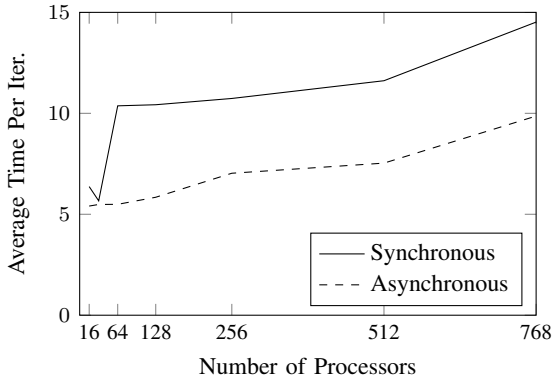



Fig. 10: Average time per iteration as the number of processors changes. Each function evaluation takes about 5 seconds with very little variance in task times apart from communication.

In practice, there may be variance in task times even if function evaluation times are homogeneous, and task times can be longer for synchronous than for asynchronous parallel PSO because communication is more expensive when all processors are communicating at the same time. Figure 10 shows the average time per iteration of synchronous and asynchronous parallel PSO for tasks with very little variance in function evaluation times. At least for this particular implementation of parallel PSO, the benefit of asynchronous parallel PSO is even greater than might be expected.

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

## V. Conclusion

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

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

- First, the performance of PSO per iteration at different numbers of processors indicates how well the algorithms use function evaluations as the number of processors scales.
- Second, the number and size of messages per particle per iteration indicates the amount of communication required. Furthermore, any communication beyond that required by the PSO topology, such as centralized coordination, must be identified.

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

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

## References


[1] J. Riget and J. S. Vesterstrøm, "A diversity-guided particle swarm optimizer—the ARPSO," EVALife, Dept. of Computer Science, University of Aarhus, Denmark, Tech. Rep., 2002.

[2] D. Bratton and J. Kennedy, "Defining a standard for particle swarm optimization," in *Proc. IEEE Swarm Intelligence Symposium*, 2007.

[3] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Harlow, England: Addison-Wesley, 2003.

[4] M. Clerc and J. Kennedy, "The particle swarm—explosion, stability, and convergence in a multidimensional complex space," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 1, 2002.

[5] R. Mendes, "Population topologies and their influence in particle swarm performance," Ph.D. dissertation, Universidade do Minho, Guimaraes, Portugal, 2004.

[6] D. H. Wolpert and W. G. Macready, "No Free Lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, 1997.

[7] D. Gies and Y. Rahmat-Samii, "Particle swarm optimization for reconfigurable phase-differentiated array design," *Microwave and Optical Technology Letters*, vol. 38, no. 3, 2003.

[8] M. Belal and T. El-Ghazawi, "Parallel models for particle swarm optimizers," *International Journal of Intelligent Computing and Information Sciences*, vol. 4, no. 1, 2004.

[9] J. Schutte, J. Reinbolt, B. Fregly, R. Haftka, and A. George, "Parallel global optimization with the particle swarm algorithm," *International Journal for Numerical Methods in Engineering*, vol. 61, no. 13, 2004.

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

[11] J. Li, X. Wang, R. He, and Z. Chi, "An efficient fine-grained parallel genetic algorithm based on GPU-accelerated," in *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, 2007.

[12] I. Scriven, A. Lewis, D. Ireland, and J. Lu, "Decentralised distributed multiple objective particle swarm optimisation using peer to peer networks," in *Proc. IEEE Congress on Evolutionary Computation*, 2008.

[13] A. McNabb and K. Seppi, "The apiary topology: Emergent behavior in communities of particle swarms," in *Proc. Parallel Problem Solving from Nature*, 2012.

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

[15] A. McNabb, M. Gardner, and K. Seppi, "An exploration of topologies and communication in large particle swarms," in *Proc. IEEE Congress on Evolutionary Computation*, 2009, pp. 712–719.

[16] J. Liang and P. Suganthan, "Dynamic multi-swarm particle swarm optimizer," in *Proc. IEEE Swarm Intelligence Symposium*, 2005.

[17] J. Jordan, S. Helwig, and R. Wanka, "Social interaction in particle swarm optimization, the ranked FIPS, and adaptive multi-swarms," in *Proc. Conference on Genetic and Evolutionary Computation*. ACM, 2008.

[18] K. E. Parsopoulos, "Parallel cooperative micro-particle swarm optimization: A master–slave model," *Applied Soft Computing*, vol. 12, no. 11, 2012.

[19] G. Venter and J. Sobieszczanski-Sobieski, "A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations," in *Proc. World Congress on Structural and Multidisciplinary Optimization*, 2005.

[20] B.-I. Koh, A. George, R. Haftka, and B. Fregly, "Parallel asynchronous particle swarm optimization," *International Journal of Numerical Methods in Engineering*, vol. 67, 2006.

[21] I. Scriven, D. Ireland, A. Lewis, S. Mostaghim, and J. Branke, "Asynchronous multiple objective particle swarm optimisation in unreliable distributed environments," in *Proc. IEEE Congress on Evolutionary Computation*, 2008.