Parallel PSO Using MapReduce

Andrew W. McNabb, Christopher K. Monson, and Kevin D. Seppi

Abstract— **In optimization problems involving large amounts of data, such as web content, commercial transaction information, or bioinformatics data, individual function evaluations may take minutes or even hours. Particle Swarm Optimization (PSO) must be parallelized for such functions. However, large-scale parallel programs must communicate efficiently, balance work across all processors, and address problems such as failed nodes.**

We present MapReduce Particle Swarm Optimization (MRPSO), a PSO implementation based on the MapReduce parallel programming model. We describe MapReduce and show how PSO can be naturally expressed in this model, without explicitly addressing any of the details of parallelization. We present a benchmark function for evaluating MRPSO and note that MRPSO is not appropriate for optimizing easily evaluated functions. We demonstrate that MRPSO scales to 256 processors on moderately difficult problems and tolerates node failures.

I. Introduction

Particle Swarm Optimization (PSO) is an optimization algorithm that was inspired by experiments with simulated bird flocking [8]. This evolutionary algorithm has become popular because it is simple, requires little tuning, and has been found to be effective for a wide range of problems. Often a function that needs to be optimized takes a long time to evaluate. A problem using web content, commercial transaction information, or bioinformatics data, for example, may involve large amounts of data and require minutes or hours for each function evaluation. To optimize such functions, PSO must be parallelized.

Unfortunately, large-scale PSO, like all large-scale parallel programs, faces a wide range of problems. Inefficient communication or poor load balancing can keep a program from scaling to a large number of processors. Once a program successfully scales, it must still address the issue of failing nodes. For example, assuming that a node fails, on average, once a year, then the probability of at least one node failing during a 24-hour job on a 256-node cluster is $1 - (1 - 1/365)^{256} = 50.5\%$. On a 1000-node cluster, the probability of failure rises to 93.6%.

Google faced these same problems in large-scale parallelization, with hundreds of specialized parallel programs that performed web indexing, log analysis, and other operations on large datasets. A common system was created to simplify these programs. Google's MapReduce is a programming model and computation platform for parallel computing [4]. It allows simple programs to benefit from advanced mechanisms for communication, load balancing, and fault tolerance.

Andrew W. McNabb is with the Department of Computer Science, Brigham Young University, 3361 TMCB, Provo, UT 84602 (phone: 801-422-8717; email: a@cs.byu.edu).

Christopher K. Monson is with Google, Inc., 4720 Forbes Ave., Lower Level, Pittsburgh, PA 15213 (email: c@cs.byu.edu).

Kevin D. Seppi is with the Department of Computer Science, Brigham Young University, 3361 TMCB, Provo, UT 84602 (phone: 801-422-4619; email: k@cs.byu.edu).

MapReduce Particle Swarm Optimization (MRPSO) is a parallel implementation of Particle Swarm Optimization for computationally intensive functions. MRPSO is simple, flexible, scalable, and robust because it is designed in the MapReduce parallel programming model.

Since MRPSO is intended for computationally intensive functions, we use the problem of training a radial basis function (RBF) network as representative of optimization problems which use large amounts of data. An RBF network is a simple function approximator that can be trained by PSO, with difficulty proportional to the amount of training data.

Section II reviews standard PSO and prior work in parallel PSO. Section III discusses the MapReduce model in detail, including a simple example. Section IV describes how Particle Swarm Optimization can be cast in the MapReduce model, without any explicit reference to load balancing, fault tolerance, or any other problems associated with large-scale parallelization. Section V shows that MRPSO scales well through 256 processors on moderately difficult problems but should not be used to optimize trivially evaluated functions.

II. Particle Swarm Optimization

In Particle Swarm Optimization, a set of particles explores the input space of a function. Each particle has a position and velocity, which are updated during each iteration of the algorithm. Additionally, each particle remembers its own best position so far (personal best) and the best position found by any particle in the swarm (global best).

Initially, each particle has a random position and velocity drawn from a function-specific feasible region. The particle evaluates the function and updates its velocity such that it is drawn towards its personal best point and the global best point. This influence towards promising locations is strong enough that the particle eventually converges but weak enough that the particles explore a wide area.

The following equations are used in constricted PSO to update the position $\mathbf{x}$ and velocity $\mathbf{v}$ of a particle with personal best $\mathbf{p}$ and global best $\mathbf{g}$:

$$\mathbf{v}_{t+1} = \chi \left[\mathbf{v}_t + \phi_1 \operatorname{U}() \otimes (\mathbf{p} - \mathbf{x}_t) + \phi_2 \operatorname{U}() \otimes (\mathbf{g} - \mathbf{x}_t)\right] \tag{1}$$

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

where $\phi_1 = \phi_2 = 2.05$, $\operatorname{U}()$ is a vector of samples drawn from a standard uniform distribution, and $\otimes$ represents element-wise multiplication. The constriction χ is defined to be:

$$\chi = \frac{2\kappa}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|}$$

where $\kappa = 1.0$ and $\phi = \phi_1 + \phi_2$ [3].

There are several parallel adaptations of Particle Swarm Optimization. Synchronous PSO, like MRPSO, preserves the exact semantics of serial PSO [12]. In contrast, asynchronous variants do not preserve the exact semantics of serial PSO, but instead focus on better load balancing [9], [13]. Other variants propose different topologies to limit communication among particles and between groups of particles [1], [10]. Parallel PSO has been applied to applications including antenna design [6] and biomechanics [9] and adapted to solve multiobjective optimization problems [10], [11].

III. MapReduce

MapReduce is a functional programming model that is well suited to parallel computation. In the model, a program consists of a high-level map function and reduce function which meet a few simple requirements. If a problem is formulated in this way, it can be parallelized automatically.

In MapReduce, all data are in the form of keys with associated values. For example, in a program that counts the frequency of occurrences for various words, the key would be a word and the value would be its frequency.

A MapReduce operation takes place in two main stages. In the first stage, the map function is called once for each input record. At each call, it may produce any number of output records. In the second stage, this intermediate output is sorted and grouped by key, and the reduce function is called once for each key. The reduce function is given all associated values for the key and outputs a new list of values (often "reduced" in length from the original list of values).

The following notation and example are based on the original presentation [4].

A. Map Function

A map function is defined as a function that takes a single key-value pair and outputs a list of new key-value pairs. The input key may be of a different type than the output keys, and the input value may be of a different type than the output values:

$$\mathsf{map} : (K_1, V_1) \rightarrow \mathsf{list}((K_2, V_2))$$

Since the map function only takes a single record, all map operations are independent of each other and fully parallelizable.

B. Reduce Function

A reduce function is a function that reads a key and a corresponding list of values and outputs a new list of values for that key. The input and output values are of the same type. Mathematically, this would be written:

$$\mathsf{reduce} : (K_2, \mathsf{list}(V_2)) \rightarrow \mathsf{list}(V_2)$$

Function 1 WordCount Map

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

Function 2 WordCount Reduce

```
def reducer(key, value_list):
    total = sum(value_list)
    emit(total)
```

A reduce operation may depend on the output from any number of map calls, so no reduce operation can begin until all map operations have completed. However, the reduce operations are independent of each other and may be run in parallel.

Although the formal definition of map and reduce functions would indicate building up a list of outputs and then returning the list at the end, it is more convenient in practice to emit one element of the list at a time and return nothing. Conceptually, these emitted elements still constitute a list.

C. Example: WordCount

The classic MapReduce example is WordCount, a program which counts the number of occurrences of each word in a document or set of documents. For this program, the input and output sets are:

$$\begin{aligned} K_1 &: \mathbb{N} \\ V_1 &: \text{set of all strings} \\ K_2 &: \text{set of all strings} \\ V_2 &: \mathbb{N} \end{aligned}$$

In WordCount, the input value is a line of text. The input key is ignored but arbitrarily set to be the line number for the input value. The output key is a word, and the output value is its count.

The map function, shown as Function 1, splits the input line into individual words. For each word, it emits the key-value pair formed by the word and the value 1.

The reduce function, shown as Function 2, takes a word and list of counts, performs a sum reduction, and emits the result. This is the only element emitted, so the output of the reduce function is a list of size 1.

These map and reduce functions are deceptively simple. The problem itself is inherently difficult—implementing a scalable distributed word count system with fault-tolerance and load-balancing is not easy. However all of the complexity is found in the surrounding MapReduce infrastructure rather than in the map and reduce functions. Note that the reduce function does not even output a key, since the MapReduce system already knows what key it passed in.

The data given to map and reduce functions, as in this example, are generally as fine-grained as possible. This ensures that the implementation can split up and distribute tasks. The MapReduce system consolidates the intermediate output from all of the map tasks. These records are sorted and grouped by key before being sent to the reduce tasks.

If the map tasks emit a large number of records, the sort phase can take a long time. MapReduce addresses this potential problem by introducing the concept of a combiner function. If a combiner is available, the MapReduce system will locally sort the output from several map calls on the same machine and perform a "local reduce" using the combiner function. This reduces the amount of data that must be sent over the network for the main sort leading to the reduce phase. In WordCount, the reduce function would work as a combiner without any modifications.

D. Benefits of MapReduce

Although not all algorithms can be efficiently formulated in terms of map and reduce functions, MapReduce provides many benefits over other parallel processing models. In this model, a program consists of only a map function and a reduce function. Everything else is common to all programs. The infrastructure provided by a MapReduce implementation manages all of the details of communication, load balancing, fault tolerance, resource allocation, job startup, and file distribution. This runtime system is written and maintained by parallel programming specialists, who can ensure that the system is robust and optimized, while those who write mappers and reducers can focus on the problem at hand without worrying about implementation details.

A MapReduce system determines task granularity at runtime and distributes tasks to compute nodes as processors become available. If some nodes are faster than others, they will be given more tasks, and if a node fails, the system automatically reassigns the interrupted task.

E. MapReduce Implementations

Google has described its MapReduce implementation in published papers and slides, but it has not released the system to the public. Presumably the implementation is highly optimized because Google uses it to produce its web index.

The Apache Lucene project has developed Hadoop, an Java-based open-source clone of Google's closed MapReduce platform. The platform is relatively new but rapidly maturing. At this time, Hadoop overhead is significant but not overwhelming and is expected to decrease with further development.

IV. MapReduce PSO (MRPSO)

In an iteration of Particle Swarm Optimization, each particle in the swarm moves to a new position, updates its velocity, evaluates the function at the new point, updates its personal best if this value is the best seen so far, and updates its global best after comparison with its neighbors. Except for updating its global best, each particle updates independently of the rest of the swarm.

Due to the limited communication among particles, updating a swarm can be formulated as a MapReduce operation. As a particle is mapped, it receives a new position, velocity, value, and personal best. In the reduce phase, it incorporates information from other particles in the swarm to update its global best. The MRPSO implementation conforms to the MapReduce model while performing the same calculations as standard Particle Swarm Optimization.

A. Particle Representation and Messages

In MRPSO, the input and output sets are:

$$\begin{aligned} K_1 &: \mathbb{N} \\ V_1 &: \text{set of all strings} \\ K_2 &: \mathbb{N} \\ V_2 &: \text{set of all strings} \end{aligned}$$

Each particle is identified by a numerical `id` key, and particle state is represented as a string. The state of a particle consists of its dependents list (neighbors' `ids`), position, velocity, value, personal best position, personal best value, global best position, and global best value. The state string is a semicolon-separated list of fields. If a field is vector valued, its individual elements are comma-separated. The state string is of the form:

$$\text{deps; pos; vel; val; pbpos; pbval; gbpos; gbval}$$

A typical particle is shown in Figure 1. This particle is exploring the function $f(\mathbf{x}) = x_1^2 + x_2^2$. Its components are interpreted as follows:

3	particle id
$1, 2, 3, 4$	dependents (neighbors)
$1.7639, 2.5271$	current position (x_1, x_2)
$52.558, 50.444$	velocity (x_1, x_2)
9.4976	value of $f(\mathbf{x})$ at the current position $(1.7639^2 + 2.5271^2)$
$1.7639, 2.5271$	personal best position (x_1, x_2)
9.4976	personal best value
$-1.0151, -2.0254$	global best position (x_1, x_2)
5.1325	global best value

MRPSO also creates messages, which are like particles except that they have empty dependents lists. A message is sent from one particle to another as part of the MapReduce operation. In the reduce phase, the recipient reads the personal best from the message and updates its global best accordingly.

B. MRPSO Map Function

The MRPSO map function, shown as Function 3, is called once for each particle in the population. The key is the `id` of the particle, and the value is its state string representation. The PSO mapper finds the new position and velocity of the particle and evaluates the function at the new point. It then calls the update method of the particle with this information. In addition to modifying the particle's state to reflect the new position, velocity, and value, this method replaces the personal best if a more fit position has been found.

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

Fig. 1. A particle as a key-value pair

Function 3 MRPSO Map

```
def mapper(key, value):
    particle = Particle(value)

    # Update the particle:
    new_position, new_velocity = pso_motion(particle)
    y = evaluate_function(new_position)
    particle.update(new_position, new_velocity, y)

    # Emit a message for each dependent particle:
    message = particle.make_message()
    for dependent_id in particle.dependent_list:
        if dependent_id == key:
            particle.gbest_candidate(particle.pbest_position, particle.pbest_value)
        else:
            emit((dependent_id, repr(message)))

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

Function 4 MRPSO Reduce

```
def reducer(key, value_list):
    particle = None
    best = None

    # Of all of the inputs, find the record with the best gbest_value:
    for value in value_list:
        record = Particle(value)
        if (best is None) or (record.gbest_value <= best.gbest_value):
            best = record
        if not record.is_message():
            particle = record

    # Update the gbest of the particle and emit:
    if particle is not None:
        particle.gbest_candidate(best.gbest_position, best_value)
        emit(repr(particle))
    else:
        emit(repr(best))
```

The key to implementing PSO in MapReduce is communication between particles. Each particle maintains a dependents list containing the `ids` of all neighbors that need information from the particle to update their own global bests. After the map function updates the state of a particle, it emits messages to all dependent particles. When a message is emitted, its corresponding key is the `id` of the destination particle, and its value is the string representation, which includes the position, velocity, value, and personal best of the source particle. The global best of the message is also set to the personal best.

If the particle is a dependent of itself, as is usually the case, the map function updates the global best of the particle if the personal best is an improvement. Finally, the map function emits the updated particle and terminates.

C. MRPSO Reduce Function

The MRPSO reduce function, shown as Function 4, receives a key and a list of all associated values. The key is the `id` of a particular particle in the population. The values list contains the newly updated particle and a message from each neighbor. The PSO reducer combines information from all of these messages to update the global best position and global best value of the particle. The reducer emits only the updated particle.

Function 4 also works as a combiner. If no particle is found in the input value list, the function combines the list by emitting only the best message. This message is then sent to the reducer.

D. Map and Reduce in Context

When a particle is emitted by a reducer, it is fully updated. In the map phase, it updates, moves, and evaluates, and then updates its personal best. In the reduce phase, it updates its global best after receiving messages from all of its neighbors in the swarm. A map phase followed by a reduce phase performs an iteration of the swarm that is exactly equivalent to an iteration in single-processor PSO.

Observe that the MRPSO implementation does not explicitly deal with communication across nodes, load balancing, or node failures. The MapReduce formulation of the problem allows the work to be divided in small enough pieces that the MapReduce system can balance work across processors and deal with failed nodes.

V. Results and Remarks

A. Implementation

Our experiments involve both a serial and a parallel implementation of Particle Swarm Optimization. The two Python programs share code for particle motion and for performing evaluations of the objective function. Particle motion is a straightforward implementation of (1) and (2).

The serial PSO program creates a swarm, or list of particle objects. During each iteration, it updates the velocity, position, value, and personal best of all of the particles. It then finds the global best, updates all of the particles, and continues to the next iteration.

The MRPSO program performs the same operations as the sequential code. However, instead of performing PSO iterations internally, it delegates this work to the Hadoop MapReduce system. After creating the initial swarm, it saves the particles to a file as a list of key-value pairs, as in Figure 1. This file is the input for the first MapReduce operation. Hadoop performs a sequence of MapReduce operations, each of which evaluates a single iteration of the particle swarm. The output of each MapReduce operation represents the state of the swarm after the iteration of PSO, and this output is used as the input for the following iteration. In each MapReduce operation, Hadoop calls map (Function 3) and reduce (Function 4) in parallel to update particles in the swarm. Note that the code that we have shown for these two functions is the actual implementation.

B. Environment

Performance experiments were run on Brigham Young University's Marylou4 supercomputer. Marylou4 is a cluster of Dell 1955 blade servers. Each node has four 2.6 GHz Xeon cores and 8 GB of memory. In serial experiments, we reserved one processor per node, and in parallel experiments, we used all four processors on each machine. Hadoop version 0.10 in Java 1.6 was used as the MapReduce system for all experiments. Both MRPSO and serial PSO were run in a Python 2.5 interpreter. Hadoop's streaming system provided the interface between Java and Python code.

C. Methodology

MRPSO performs the same calculations as a serial implementation of PSO. With the same number of particles and iterations, MRPSO and serial PSO will achieve the same level of accuracy. Comparing the quality of solutions is useful only to verify correctness. However, the average execution time per iteration is important because it shows whether the parallel implementation is an improvement. Unless noted otherwise, the first iteration of each run was excluded from averages because they often ran slightly faster or slower than the rest of the runs.

Each swarm consists of 1,000 fully connected particles. Since each particle has a 1,000 neighbors, the dependents list is quite large. Since the sociometry is static in this case, an explicit list is not necessary. To save space, we replaced the full dependents list field in the string representation with the special string "all-1000." When a particle saw this string, it emitted messages for all 1,000 particles in the swarm as if the dependents list were included.

MapReduce has many parameters involving issues such as how to partition the input and how many tasks to run concurrently on each machine. We decided how to set these parameters after performing some initial experiments with n nodes and p processors for various values of n and p. The number of tasks per node, which indicates how many total map and reduce functions can be executing concurrently on one physical computer, was set to 4 (the number of processors per node). The number of map tasks per job, which determines how finely to partition the input, was set to p, the total number of processors. We used $\log_2 n$, but not more than 4, reduce tasks per job. We also configured the MapReduce system to use the reduce function as a combiner.

We used speedup as a measure of scalability. However, there are enough variants of speedup that the measure is worse than useless without precise clarification. Speedup is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processing elements [5]. Thus the speedup with p processors is:

$$S_p = \frac{t_1}{t_p} \tag{3}$$

The definition of speedup is ambiguous as to what constitutes the best sequential algorithm. Since MRPSO is a reformulation of PSO that performs the same operations, we use our standard single-processor PSO implementation as the best sequential algorithm. This implementation and the MRPSO implementation are written in the same language, share common code, and run on the same hardware.

D. *RBF Network Training*

We used a RBF network training function as our primary test function because it is representative of functions that use large amounts of data. An RBF network is a sum of radial basis functions and is used as a function approximator [2]. The following equation describes the activation function of an RBF network:

$$f(\mathbf{x}) = \sum_{i=1}^{n_{\text{bases}}} \frac{s_i}{\sqrt{2\pi}} \exp\left[\left(\sum_{j=1}^{d_{\text{input}}} \frac{w_{ij}}{625}(x_i - c_{ij})^2\right)^{\frac{1}{2}}\right] \tag{4}$$

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

Given a set of training points with known values, a particle swarm can minimize the sum square error function to find the parameters of the RBF network that best fits the data. Thus, the training problem becomes an optimization problem of the error function:

$$g(\mathbf{X}, \mathbf{y}) = \sum_{i=1}^{n_{\text{points}}} (f(\mathbf{X_i}) - y_i)^2 \tag{5}$$

where $\mathbf{X}$ are the training points and y are the corresponding values. A particle swarm finds parameters for the RBF network f in (4) that minimize the error function g shown in (5).

To a particle swarm, an RBF network with one-dimensional input and four basis functions is represented as a 12-dimensional vector with 3 parameters for each of the 4 bases, for example:

$$\begin{aligned}&(s_1, w_{11}, c_{11}, s_2, w_{21}, c_{21}, s_3, w_{31}, c_{31}, s_4, w_{41}, c_{41})\\ &= (32, 1.3, -22, 11, 37, 18, 45, 4.3, -7.8, 1.4, 11, 0.53)\end{aligned} \tag{6}$$

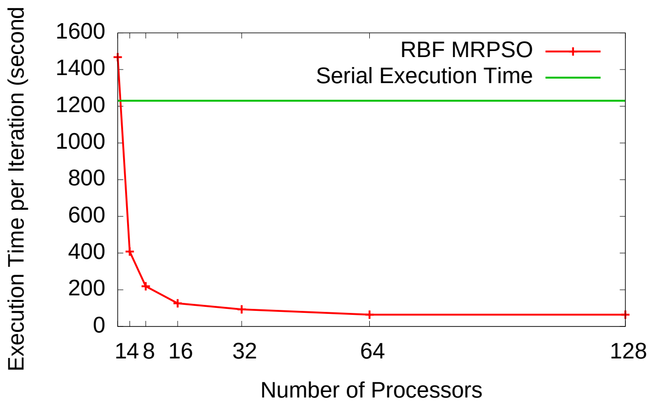

Fig. 2. Execution times per iteration for RBF with 10,000 points

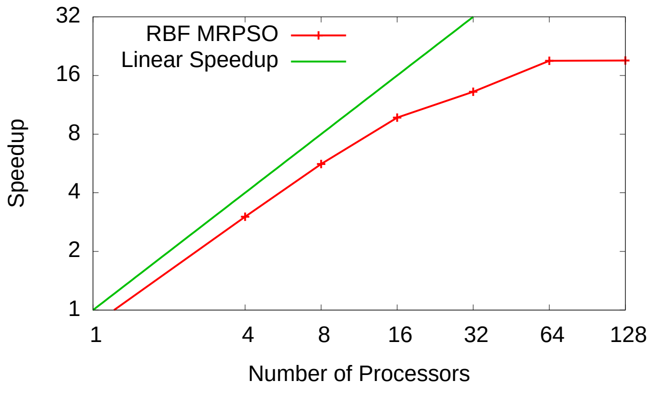

Fig. 3. Speedup for RBF with 10,000 data points

E. *RBF Results*

We used PSO for the 12-dimensional problem of optimizing weights for an RBF network. The function minimized by PSO was g in (5); in this problem, a particle's position is a vector of weights for an RBF network. The training data was a set of 10,000 samples from the RBF network of (6).

We ran PSO for the serial implementation and for MRPSO with 1, 4, 8, 16, 32, 64, and 128 processors. In each case, we report the average execution time of at least 70 iterations of PSO. For MRPSO, the estimated standard deviation of execution times ranged from 2.3 seconds for 8 processors to 6.7 seconds for 128 processors. For serial PSO, the estimated standard deviation was 34 seconds (2.8% of the average time). The execution times are shown in Figure 2.

With 10,000 training points, each evaluation of g took about 1.2 seconds. Although this is not a very long time, it is much longer than common PSO benchmark functions. In fact, a single iteration in the serial implementation took 1,230 seconds (20.5 minutes) to complete. A training set of 10,000 data points is not a large, and a function that takes 1.2 seconds to compute is not slow. However, even with this function, parallelization made a huge difference: with 64 processors, each iteration took 65 seconds.

The speedup, as calculated using (3), is shown in Figure 3.

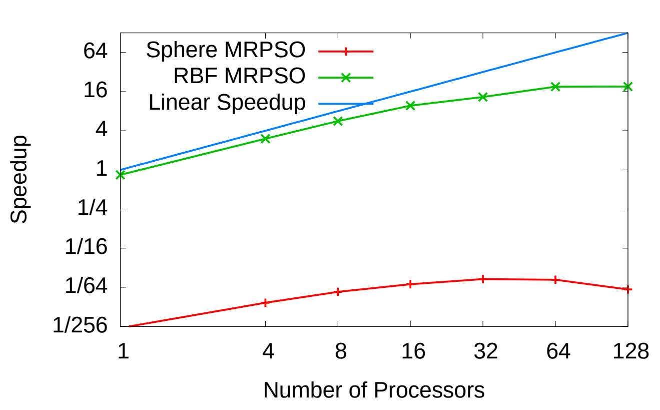

Fig. 4. Speedup for the sphere function compared to the speedup for RBF. Using MRPSO for sphere would not be appropriate.

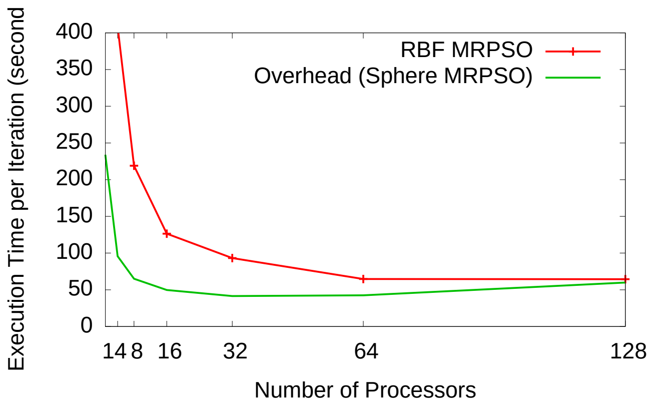

Fig. 5. Execution times per iteration for RBF with 10,000 data points and implementation overhead as measured by sphere

Improvement was dramatic until 64 processors, but beyond this point, implementation and communication overhead hindered further improvement. For each iteration with 128 processors, the amount of computation per processor was only 9.6 seconds.

F. *Sphere*

A MapReduce runtime system introduces overhead due to job startup, communication, and sorting. Additional overhead is incurred by MRPSO's inter-particle messages. If the function being optimized is simple enough that particle communication takes longer than function evaluation, then MRPSO should not be used.

The sphere function is: $f(\mathbf{x}) = x_1^2 + x_2^2 + \cdots + x_n^2$. Like all of the standard benchmarks, it is easily evaluated. In our serial PSO, it took less than a millisecond per evaluation on 12-dimensional sphere with 1,000 particles. For parallelization to be useful, there would have to be almost no additional overhead. In MapReduce, the overhead to process each particle would certainly be expected to take much longer than a millisecond (on the cluster, round trip time alone is 0.084 milliseconds). In Hadoop 0.10, the amount of time to evaluate sphere was dominated by the time needed by the MapReduce runtime system.

Figure 4 shows the speedup of MRPSO with 1,000 particles on 12-dimensional sphere. The baseline is a standard serial implementation which completed each iteration in 0.867 seconds on average. Even at its best, MRPSO took 41.5 seconds per iteration, which is many times slower than a millisecond. MRPSO should not be used for easily evaluated functions.

The sphere function serves another useful purpose by measuring the amount of MRPSO overhead. Since each evaluation of sphere takes less than a millisecond to compute, the function is essentially a null operation compared to the total execution time. Overlaying the graph of RBF execution times with the graph of sphere execution times shows how much of the time was spent on function evaluation and how much was spent on overhead. In Figure 5, the time between the two curves approximates the amount of RBF computation time, while the line beneath the lower curve shows the amount of overhead. As the number of processors increases to 128, the two curves nearly meet. After this point, each additional processor increases overhead more than it contributes to computation. Some of the overhead represented by this curve is unavoidable, but much of it will decrease as Hadoop continues to improve.

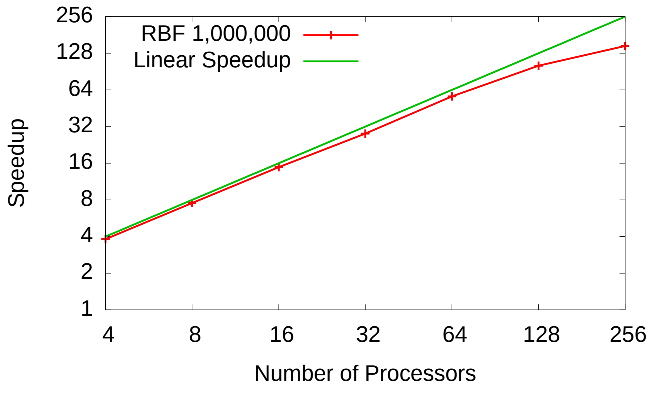

Fig. 6. Speedup for RBF with 1,000,000 data points

G. *RBF With 1,000,000 Points*

The earlier RBF experiments used 10,000 training points and took 1.2 seconds to compute one function evaluation. Although MRPSO scaled well for this function, it was not particularly long-running function. However, with 100 times as many data points, the RBF network error function from (5) takes 100 times longer to run. At 120 seconds per function evaluation, training an RBF network with 1,000,000 training points is noticeably slow. Over 10 serial PSO experiments, the average time per iteration was 120,000 seconds (33 hours), with an estimated standard deviation of 710 seconds (12 minutes).

MRPSO experiments were similar to the previous experiments. However, the first iteration was not dropped because of the sparsity of data. Also, the 256-processor experiments were run with 500 map tasks instead of 256 because of the need for load balancing, as discussed below.

Figure 6 shows the speedup of RBF network training in MRPSO. Note that the RBF nearly matches linear speedup through 128 processors. The speedup with 16 processors is 14.9, and the speedup with 128 processors is 101.

H. Load Balancing

In each experiment with up to 128 processors, the number of map tasks was equal to the total number of processors. In these experiments, the MapReduce system performed static load balancing. It split the input into similarly sized tasks and assigned a task to each processor.

Alternatively, the number of map tasks can be set to the total number of particles. In this case, there would be 1,000 map tasks, with exactly one particle in each map task. However, experimentation showed that this increased overhead.

With 256, we found that it was relatively common for a machine to experience a network error and lose a map task. When this happened, the MapReduce system recognized the fault and restarted the task. The iteration completed successfully, but the reduce tasks could not begin until the last map task completed. With 256 processors, iterations finished in less than 600 seconds in the normal case, but took more than 1,000 seconds in the event of a failure.

To reduce the variance, the number of map tasks was set to 500 instead of 256. Since there were more map tasks than processors, Hadoop performed dynamic load balancing, and if a task failed, the reassigned map task would complete more quickly because it included 2 particles instead of 4. After making this change, the slowest iteration time was 865 seconds rather than 1,078 seconds. The more processors in use, the greater the need for dynamic load balancing.

VI. Future Work and Conclusions

If an MRPSO swarm has fewer particles than the number of available processors, then the extra processors are idle. With more particles than the number of processors, the MapReduce system can dynamically balance the load. This suggests that with thousands of processors, MRPSO would perform best with a very large number of particles.

The number of messages emitted by the map function is proportional to the size of the particle's dependents list. Because of this, each particle should have a limited number of neighbors. MRPSO makes it easy to control the swarm sociometry, but it is still not clear which sociometries work best in which contexts, and very little work has been done on large swarms. Experiments with more sparsely connected sociometries such as rings, directed rings, and tribes in MRPSO might show how to reduce communication and improve optimization. [7]

Since a particle's dependents list is part of its state, it can be updated during either the map or the reduce phase. Dynamic changes to the dependents list might affect the performance of PSO.

MRPSO makes no assumptions about whether the sociometry is static or dynamic. If the sociometry is assumed to be static, then the map function could refrain from emitting unnecessary messages. In this case, a message would only be emitted in iterations where a particle updates its personal best. This might reduce the average communication overhead.

In summary, we have shown that Particle Swarm Optimization can be naturally adapted to the MapReduce programming model. With a function that took 2 minutes to evaluate, MapReduce Particle Swarm Optimization scaled well through 256 processors. MRPSO addresses the problems that face highly parallel programs because it builds on a system that is specifically designed be robust.

References

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

[2] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, November 1995.

[3] 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, February 2002.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Sixth Symposium on Operating System Design and Implementation*, November 2004.

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

[6] Nanbo Jin and Yahya Rahmat-Samii. Parallel particle swarm optimization and finite-difference time-domain (pso/fdtd) algorithm for multiband and wide-band patch antenna designs. *IEEE Transactions on Antennas and Propogation*, 53(11):3459–3468, 2005.

[7] James Kennedy. Small worlds and mega-minds: Effects of neighborhood topology on particle swarm performance. In *Proceedings of the Congress of Evolutionary Computation*, volume 3, pages 1931–1938. IEEE Press, 1999.

[8] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *International Conference on Neural Networks IV*, pages 1942–1948, Piscataway, NJ, 1995. IEEE Service Center.

[9] 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.

[10] Sanaz Mostaghim, Jürgen Branke, and Hartmut Schmeck. Multi-objective particle swarm optimization on computer grids. Technical Report 502, AIFB Institute, DEC 2006.

[11] K. Parsopoulos, D. Tasoulis, and M. Vrahatis. Multiobjective optimization using parallel vector evaluated particle swarm optimization. In *Proceedings of the IASTED International Conference on Artificial Intelligence and Applications*, 2004.

[12] J. Schutte, J. Reinbolt, B. Fregly, R. Haftka, and A. George. Parallel global optimization with the particle swarm algorithm. *International Journal of Numerical Methods in Engineering*, 61:2296–2315, 2004.

[13] 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.