MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms

Chao Jin, Christian Vecchiola and Rajkumar Buyya

Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
Email: {chaojin, csve, raj}@csse.unimelb.edu.au

Abstract

The MapReduce programming model allows users to easily develop distributed applications in data centers. However, many applications cannot be exactly expressed with MapReduce due to their specific characteristics. For instance, Genetic Algorithms (GAs) naturally fit into an iterative style. That does not follow the two phase pattern of MapReduce. This paper presents an extension to the MapReduce model featuring a hierarchical reduction phase. This model is called MRPGA (MapReduce for Parallel GAs), which can automatically parallelize GAs. We describe the design and implementation of the extended MapReduce model on a .NET-based enterprise Grid system in detail. The evaluation of this model with its runtime system is presented using example applications.

1. Introduction

Genetic Algorithms (GAs) are a class of evolutionary algorithms [16], which are widely used in many domains, such as CAD/CAM, scheduling, chemistry, and biology [7]. In particular, GAs are used to search for an nearly optimal solution in complex spaces with a computationally intensive solution [19]. Normally, GAs abstract the problem space as a population of individuals and deploy a randomized optimization method to generate offspring for searching near-optimal individuals. Generally, this process takes a long time for large problem sizes. To improve efficiency, Parallel Genetic Algorithms (PGAs) have been adopted. PGAs [8] normally split a problem space into a number of smaller sub-spaces, then explore sub-optimal solutions for each sub-space, and finally find out a set of optimal solutions based on the sub-optimal solutions. PGAs can not only reduce the execution time, but also can tackle more complex problems by taking advantage of distributed computing systems. Furthermore, PGAs are more versatile than their corresponding sequential version as they have a less possibility of getting stuck in local optima.

However, parallelized GAs still face the common development difficulties in distributed environments, such as communication and synchronization between distributed components. Furthermore, due to the increase of cloud computing [3][13], PGAs have to solve more challenging problems common in data centers, such as heterogeneity and frequent failures. Many existing models for PGAs are based on message passing interface (MPI), which is not designed for cloud computing. Thus, it is important to explore a more suitable solution for performing distributed GAs in data centers.

The MapReduce model [10] provides a parallel design pattern for simplifying application developments in distributed environments. This model can split a large problem space into small pieces and automatically parallelize the execution of small tasks on the smaller space. It was proposed by Google for easily harnessing a large number of resources in data centers to process data-intensive applications and has been proposed to form the basis of a "data center computer" [5]. This model allows users to benefit from advanced features of distributed computing without worrying about the difficulty of coordinating the execution of parallel tasks in distributed environments.

This paper explores how to extend MapReduce to support PGAs. The iteration style of GAs cannot be expressed directly with MapReduce. We extend MapReduce by adding a phase for global selection at the end of every iteration of PGAs. Furthermore, we expose a coordinator client for coordinating the execution of PGAs iterations. The contribution of this paper includes:

1. An extension to MapReduce for automatically parallelizing GAs with sequential programming through 3 components: Map, Reduce, and Reduce, called MRPGA.
2. A runtime system on the .NET platform for MRPGA to coordinate the parallel execution of GA applications within a distributed environment.
3. Evaluation on the runtime system and programming model through real world experiments over an enterprise Grid.

The remainder of the paper is organized as follows. Section 2 reviews the MapReduce programming model. Section 3 presents the extension to MapRedce through a distributed model of PGAs. Section 4 presents the architecture of MRPGA. Section 5 describes the implementation. Section 6 discusses the experimental evaluation of the system. Section 7 concludes the paper with pointers to future work.

2. MapReduce Overview

MapReduce is triggered by *map* and *reduce* operations in functional languages, such as Lisp. This model abstracts computation problems through two functions: map and reduce. All problems formulated in this way can be parallelized automatically.

Essentially, the MapReduce model allows users to write Map/Reduce components with functional-style code. These components are then composed as a dataflow graph with fixed dependency relationship to explicitly specify its parallelism. Finally, the MapReduce runtime system can transparently explore the parallelism and schedule these components to distributed resources for execution.

All data processed by MapReduce are in the form of key/value pairs. The execution happens in two phases. In the first phase, a map function is invoked once for each input key/value pair and it can generate output key/value pairs as intermediate results. In the second one, all the intermediate results are merged and grouped by keys. The reduce function is called once for each key with associated values and produces output values as final results.

2.1. MapReduce Model

A map function takes a key/value pair as input and produces a list of key/value pairs as output. The type of output key and value can be different from input key and value:

$$map :: (key_1, value_1) \Rightarrow list(key_2, value_2)$$

A reduce function takes a key and an associated value list as input and generates a list of new values as output:

$$reduce :: (key_2, list(value_2)) \Rightarrow list(value_3)$$

2.2. MapReduce Execution

A MapReduce application is executed in a parallel manner through two phases. In the first phase, all map operations can be executed independently with each other. In the second phase, each reduce operation may depend on the outputs generated by any number of map operations. However, similar to map operations, all reduce operations can be executed independently.

From the perspective of dataflow, MapReduce execution consists of *m* independent map tasks and *r* independent reduce tasks, each of which may be dependent on *m* map tasks. Generally the intermediate results are partitioned into *r* pieces for *r* reduce tasks.

The MapReduce runtime system schedules map and reduce tasks to distributed resources, which handles many tough problems, including parallelization, concurrency control, network communication and fault tolerance. Furthermore, it performs several optimizations to decrease overhead involved in the scheduling, network communication and intermediate grouping of results.

As MapReduce can be easily understood, its abstraction considerably improves the productivity of parallel/distributed development, especially for novice programmers.

3. MapReduce for PGA

The MapReduce model cannot be used to express PGAs directly. This section describes an extension to MapReduce through adding an additional reduce phase for population selection after analyzing the general requirements of PGAs.

3.1. Overview of PGA

Genetic algorithms abstract the problem space as a population of individuals, and explore the optimum individual through a loop of operations. Usually the individual is represented by a string of symbols, and each step of the loop produces a new generation with *reproduction*, *mutation*, *evaluation* and *selection* operations. Given a generation of individuals as ancestors, the reproduction operation generates their offspring by combining several ancestors and the mutation operation performs simple stochastic variations on each offspring to generate a new version of it. The evaluation operation evaluates the offspring according to an objective function and the selection operation chooses the best one from the population for next generation. This process repeats until the optimum individual is found.

Among these operations, the evaluation and selection operation consumes most of the time and has been estimated to take more than 1 CPU year for the problems in complex domains [15].

```
function Distributed_GA()
    t = 0                          /* index of generation */
    P[0] = a1[0], …, au[0]         /* initialization */
    Evaluation(a1[0],…,an[0])
    while not T(P[t]) do
        P'[t] = Mutation(Crossover(P[t]))
        Evaluation(a'1[0],…,a'n[0])
        <Communication>
        P[t+1] = Selection(P'[t])
        t = t + 1
    endwhile
return Optimum(P[t])
```

Fig. 1 Parallel Genetic Algorithm.

There are several models for PGAs. We choose the distributed model as a general presentation of the principle of PGA. With this model, there exist many elementary *worker* GA working on separate populations. Each worker performs same computations as the rest. Fig. 1 illustrates the principle of PGAs with a distributed model. Essentially, after mutation and crossover operations, there exists a communication phase, where each worker communicates with neighbors for exchanging a set of individuals or statistics. After this communication phase, the offspring will be selected as the starting point to evolve next generation.

Each individual, P, consists of u elements, $a_1,\ldots,a_u$. T is the function that determines whether the optimum value has been generated.

$$P_x > P_y \Leftrightarrow a_{xi} > a_{yi}, i \in (1..u)$$

$$P_x < P_y \Leftrightarrow a_{xi} < a_{yi}, i \in (1..u)$$

$$P_x = P_y \Leftrightarrow \begin{cases} (\exists i, a_{xi} > a_{yi}) \wedge (\exists j, a_{xj} < a_{yj}), i, j \in (1..u) \\ \textbf{\textit{OR}} \\ a_{xi} == a_{yi}, i \in (1..u) \end{cases}$$

Fig. 2 Comparison Rule of Individuals.

To perform the selection operation, the individual with the best value of evaluation are chosen. Each individual after evaluation still consists of an array of u elements: $a_1,\ldots,a_u$. Given any two individuals, P_x and P_y, P_x is bigger than P_y only if every element of P_x is bigger than the corresponding element of P_y, as illustrated in Fig. 2.

3.2. MRPGA: MapReduce for PGA

We deploy MapReduce to parallelize those parts of a PGA that are the most time-consuming. Essentially, a map operation can be used to express the phase of local evaluation. The communication phase can be achieved by collecting dependent inputs for the reduce operation through the runtime system. However, the execution of selection cannot be achieved by one reduce operation, because after the local selection, a global selection is required. Therefore, we have to express the selection phase through two phases of reduce operations. Thus the whole execution model consists of three phases: map, reduce and reduce.

3.2.1. Key/Value Pairs for MRPGA. The types for input key/value pairs of MRPGA are illustrated as follows:

Key1	:	***Integer***
Value1	:	***Individual***
Key2	:	***Integer***
Value2	:	***Set of Individuals***
Key3	:	***Individual***
Vaue3	:	***Integer***

At first, each individual is identified by a numerical key. After being evaluated in the map phase, each individual will be associated with a common key as the result. This intermediate result is kept on the local machine. In the standard MapReduce, with this common key, all mutated individuals will be associated together without any partition. However, MRPGA deploys a different policy. Essentially, the set of individuals associated with the common key is partitioned according to their locations.

Each reduce function will be called for each partition, which is actually taken as the input list of value. As a result, a set of sub-optimal individuals is produced with the selection algorithm implemented by users in the 1st phase of reduce operation.

In the final reduce phase, all sets of sub-optimal individuals are collected and then merged and sorted. Only the best individuals are selected by the system as the input of final reduce function.

3.2.2. Map Phase. The map operation is for every individual and is called once for each of the individuals in each of the steps of the loop. As an input fed into the map function, *key* is the index of the individual, while *value* is the individual. The map operation extracts the individual from *value*, performs evaluation, and then submits the result as an intermediate output, as shown in Fig 3. In the figure, ***Emit*** is used to submit results.

```
function mapper(key, value)
    /* translation */
   P= a1[0], …, au[0]= Individual(value)
   /* perform evaluation */
   P'=Evaluation(a'1[0],…,a'n[0])
   /* Submit intermediate results */
   Emit(default_key, P')
```

Fig. 3 Map Operation for Parallel GA.

The results generated by the map phase are kept in a persistent database on the local machine. All the results are associated with same key, *default_key*. We adopt a partition policy different from the standard implementation of MapReduce. The intermediate results generated by Map functions are not partitioned by key. However, they are automatically split into pieces according to their locations. This partition policy allows each of the reduce tasks to collect dependent input just from the local machine without fetching data from a remote machine.

Intermediate results produced by map operations on the same node will be merged by key as the input for the 1st phase of reduce operation.

```
function reducer(key, value_list)
   i = 0          /* index variable */
   foreach value in value_list
      P[i]= a1[i], …, au[i]= Individual(value)
      i++
   /* perform local selection */
   P' = Selection(P)
   /* submit local optimum individuals*/
   foreach individual in P'
      Emit(individual, 1)
```

Fig. 4 The First Phase of Reduce Operation.

3.2.3. The 1st Phase of Reduce. The 1st phase of reduce operation is for each of the partition groups generated by the map phase. As illustrated in Fig.4, the reduce operation extracts populations from *value_list*, and performs selection operation on those populations to choose local optimum individuals. Finally, it submits the selection result as input for *final_reducer*.

The key of intermediate result is individual and the value is just a number. All the intermediate results generated by the 1st phase of reduce operations are collected as the input for the 2nd phase of reduce operation.

3.2.4. The 2nd Phase of Reduce. The 2nd phase of reduce operation is for the global selection, called once at the end of each iteration of the loop. Essentially, there is only one operation in the 2nd reduce phase. The final reducer takes the intermediate result generated by the reducer in the first phase and produces the final selection results for the current generation. This result will be taken as the input for the next round of MRPGA operations.

```
function final_reducer(key, value)
   /* translation */
   P= a1[0], …, au[0]= Individual(key)
   /* submit global optimum individuals*/
   Emit(P, 1)
```

Fig. 5 The Second Phase of Reduce Operation.

Local optimum individuals selected by the operation in the 1st phase of reduce are merged and sorted to select the global optimum individuals. The merging and sorting are performed by the runtime system. Through a special optimization [described in Section 4], only the best individuals are fed to the final reducer as input. Therefore, the final reducer just extracts each optimum individual from *key* and submits it as the final results, as illustrated in Fig. 5.

```
procedure MapReduce_GA()
   t = 0                    /* index of generation */
   P[0] = a1[0], …, au[0]   /* initialization */
   Evaluation (a1[0],…,an[0])
   while not T(P[t]) do
     P'[t] = Mutation(Crossover(P[t]))
     SendToScheduler(P'[t])
     P[t+1] = ReceiveFromScheduler(t)
     t = t + 1
   endwhile
return P[t]
```

Fig. 6 Coordinator.

To achieve the iterations for the population evolution, a coordinator is adopted. As illustrated in Fig. 6, the coordinator works on the reproduction, mutation, and submission of offspring to the scheduler of MRPGA and on the collection optimum individuals for each of the rounds of the evolution. Users do not have to face the difficulties of distributed computing. Instead, they only need to work on sequential programming for all the components, including one map function, two reduce functions and one coordinator. The runtime system coordinates the parallel execution of map and reduce tasks.

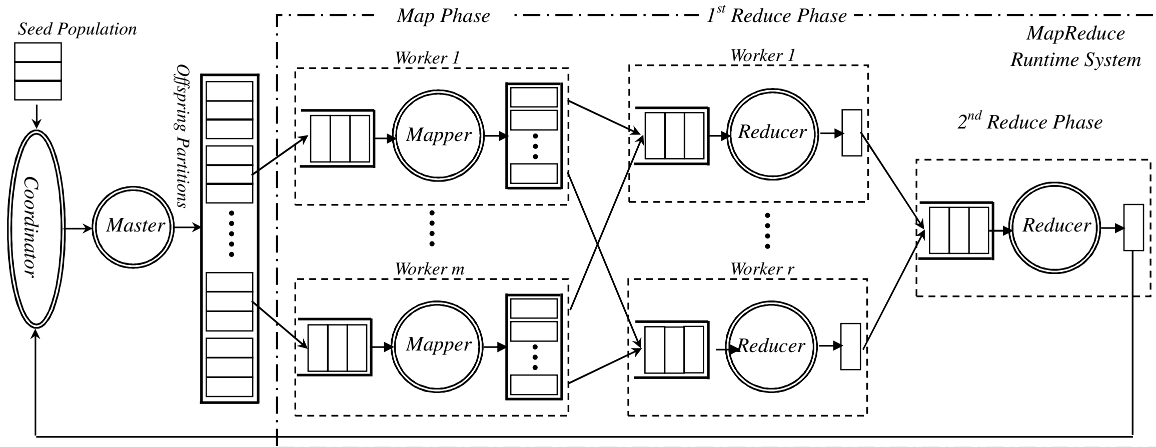

Fig. 7 Architecture of MRPGA.

4. Architecture

The architecture of the runtime system that supports MRPGA is shown in Fig. 7. The runtime system consists of one *master*, and multiple *mapper* and *reducer* workers. Mapper workers are responsible for executing the map function defined by users and reducer workers execute the reduce function, while the master schedules the execution of parallel tasks.

The control flow of execution consists of the following stages:

1) The coordinator generates offspring and performs mutation. Then, it sends the offspring to the master for evaluation and selection.

2) The master splits the offspring into *m* pieces respectively for *m* map tasks. The value of *m* is chosen so as to maximize parallelism for map tasks. Generally this value is larger than the number of machines.

3) Each piece of offspring is sent to a machine with a mapper worker. The mapper worker iterates over the individuals in the piece of input to execute the map function for each individual. Intermediate results generated by the map function are kept locally.

4) Each reducer worker is assigned with reduce tasks for the 1st phase of reduce operations. Normally the input is taken from the local machine. In case of heterogeneity, to make uniformly distribute loads over all workers, some reduce workers fetch intermediate results from neighboring machines.

5) The reduce function is invoked to select local optimum individuals that are then stored on the local machine.

6) A reducer worker is assigned to execute the final reduce function. This worker collects all the results generated in the 1st phase of reduce operation.

7) The final reduce function is invoked to produce the global optimum individuals as final results.

8) The final results are sent to the client for the next round of the evolutionary algorithm.

The above stages are repeated until the optimum individuals meet the specified requirements.

Different from the standard implementation of the MapReduce runtime system, an additional support is added for the 2nd reduce phase, including a special optimization for selecting the global optimum individuals. Since there is only one reduce task in the final phase, normally the master selects the most powerful machine from all the available resources to execute the *final_reduce* function.

Usually in the reduce phase, all inputs are collected, merged through sorting before feeding to the reduce function. MRPGA adds a policy support in the merging phase. That allows users to specify the order for the sorting and the number for the top elements which users want to process. For instance, to meet the requirements of the final reducer, users specify an ascending order, just to process the individuals with the biggest ranking value. From Fig.2, we can know that the best offspring means a set of individuals, not just one individual. For those individuals with biggest rank value, they will be fed to the final reducer at any order.

To simplify handling faults during execution, the master replicates the optimum individuals selected by MRPGA for each round of evolvement. If some machines become un-available during the execution, we just restart the execution from the last round. This fault tolerance mechanism is different from standard MapReduce implementation and therefore we do not need a complex distributed file system for reliability purpose.

5. Implementation

We have implemented MRPGA on the .NET platform using the C# language. The deployment of MRPGA is simplified by using Aneka [18]. Aneka is a .NET-based enterprise Grid software platform. It allows the creation of enterprise Grid environments. Each Aneka node consists of a configurable container hosting mandatory and optional services. The mandatory services provide the basic capabilities required in a distributed system, such as communications between Aneka nodes, security, and membership. Optional services can be installed to support the implementation of different programming models in Grid environments. The *master*, *mapper* worker and *reducer* worker of MRPGA are all implemented as optional services and can be loaded into the Aneka container according to their configurations. Fig. 8 illustrates a configuration of deployment scenario of MRPGA within Aneka. Representatively, one machine is configured with a master service, while other machines are equipped with worker services.

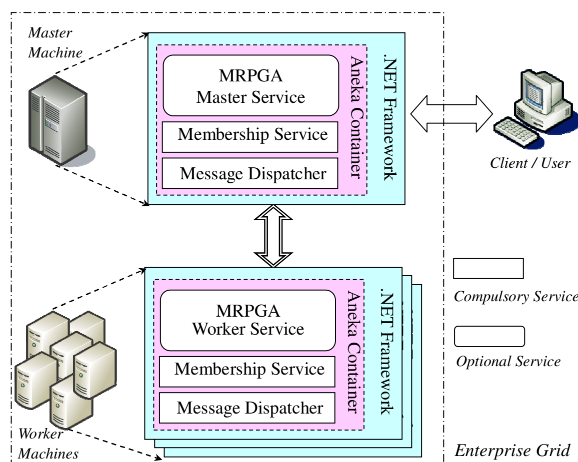

Fig. 8 Implementation of MRPGA over Aneka.

The master service utilizes the membership service supported in Aneka to monitor the status of each worker. In case of failures, the master adopts policy described in Section 4 to continue the execution. The message dispatcher service is adopted by master and worker services to perform the transfer of intermediate results between machines.

Users can define map and reduce functions with any language supported by .NET, such as C++, C# and Visual Basic. To start execution, the client should submit map and reduce functions, and initial populations to the master service. The map and reduce functions are serialized into binary format through the object reflection supported by .NET. The master service first selects available workers by querying the membership service and splits the initial offspring according to the settings of worker services. Then, the master service sends the serialized map function with pieces of initial population to each worker service. After one worker completes the execution of the map function, the master sends the serialized reduce function to the worker. When all reduce workers finish the execution for the 1st phase of reduce operation, one worker is assigned to execute the final reduce operation. Finally, the optimum individuals generated by the final reducer are collected by the master service and then forwarded to the client, which produces a new generation for next round of evolution.

6. Performance Evaluation

We have developed the MRPGA runtime system and deployed it with Aneka in several student laboratories at the University of Melbourne. The experiments were performed in an enterprise Grid consisting of 33 nodes drawn from three student laboratories. Each machine has a single Pentium 4 processor, 1GB of memory, 160GB IDE disk, 1 Gbps Ethernet and runs Windows XP operating system.

We conducted 2 experiments to evaluate the overhead and performance of our approach. This section first evaluates the overhead of MRPGA for typical GA applications, and then illustrates the scalability of the system with one example application.

6.1. Overhead of MRPGA

The MapReduce model trades performance for simplicity of programming. For instance, applications have to follow the interface of key/value pairs. However, its overhead cannot be overwhelming. Otherwise, its performance might not be acceptable. We use multiobjective evolutionary algorithms (MOEA) [12], a solution of multiple objective optimizations as the benchmark for evaluating the overhead of MRPGA by sequential execution. MOEA are an effective tool for solving search and optimization problems containing several incommensurable and possibly conflicting objectives. MOEA is reported to be used as a framework to support various applications in the real word [12].

We have integrated MOEA with MRPGA. This section presents the overhead introduced by MRPGA to MOEA. Two benchmark multiobjective problems were adopted in the experiments: DLTZ4 and DLTZ5. A more comprehensive description of these two problems can be found in [11]. In the following, we list the actual implementation for the evaluation operation:

DTLZ4 *is for minimizing*:

$$f_1(x) = (1 + g(X_M))\prod_{j=1}^{M-1}\cos(x_j^{\alpha}\pi/2)$$

$$f_k(x) = (1 + g(X_M))\sin(x^{\alpha}_{M-k+1}\pi/2)\prod_{j=1}^{M-1}\cos(x_j^{\alpha}\pi/2)$$

for k = 2, 3, …,*M*

where:

$$g(XM) = \sum_{x_i \in X_M}(x_i - 0.5)^2, 0 \le x_i \le 1, i = 1,2,...,P.$$

DTLZ5 *is for minimizing*:

$$f_1(x) = (1 + g(X_M))\prod_{j=1}^{M-1}\cos(\theta_j)$$

$$f_k(x) = (1 + g(X_M))\sin(\theta_{M-k+1})\prod_{j=1}^{M-1}\cos(\theta_j)$$

for k = 2, 3, …,*M*

where:

$$g(XM) = \sum_{x_i \in X_M}(x_i - 0.5)^2, 0 \le x_i \le 1, i = 1,2,...,P.$$

$$\theta_1 = x_1\pi/2$$

$$\theta_q = \frac{\pi}{4(1 + g(X_M))}(1 + 2g(X_M)x_q), q = 2,3,...,(M-1)$$

In the experiments, we executed the application in a sequential manner. We chose to produce 300 individuals for every iteration of the evolutionary algorithm and repeated for 100 generations. The result is illustrated in Table 1. We executed the sequential MOAE with MRPGA and without MRPGA. Then the difference between the two execution times is the overhead of the sequential MRPGA. Actually, this overhead is comparably small, less than 1% of the whole execution time for both DLTZ4 and DLTZ5. In the next section, it is shown that the overhead is amortized by the benefits of parallel execution.

Table 1 Overhead of Sequential MRPGA.

	Overhead(s)	% Execution Time
DLTZ4	56	<1%
DLTZ5	53	<1%

6.2. Scalability

For the scalability experiments, we used the framework of MOEA to simulate a real application: aerodynamic airfoil design [9]. The evaluation costs for aerodynamic airfoil design are extracted from what they claim in their respective research paper. The costs were scaled according to different machine configurations. We choose this example because its evaluation phase is one of the most time-consuming applications as we know.

A cost simulation function is used in the scalability experiment for realizing the execution time for evaluation in the application. The simulated cost follows a normal distribution. The average evaluation cost for aerodynamic airfoil design it is 10 seconds. The standard deviation, σ^2, is configured to be 0.2.

The experiments generated 500 individuals for each generation and repeated 10 times. We compare the time consumed by the parallel execution of MRPGA with that by its sequential version. The parallel execution was performed on various numbers of machines, from 4 to 32. The scalability results are illustrated in Fig. 9.

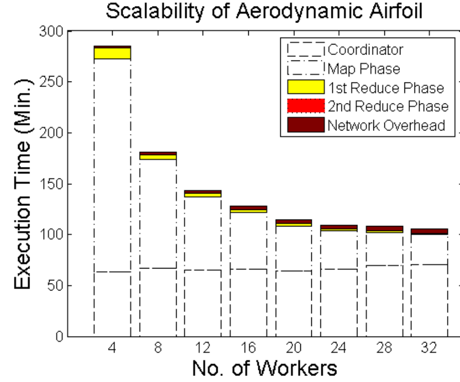

Fig. 9 Scalability Experiments.

From the results, we can see that MRPGA supports scalable performance for embarrassingly parallel tasks, including the map and reduce phases. The reason is that the parallel tasks are time-consuming and network overhead is comparably small. However, the time consumed by the sequential part of coordinator cannot be neglected. The reason is MOEA utilizes a complex algorithm to choose individuals for crossover with the complexity of $O(n^2)$. It is our future research to parallelize the crossover and reproduction operations.

7. Related Work

Parallelizing genetic algorithms have received much attention from researchers. Many models have been proposed to meet the challenges of implementing PGAs. These include the distributed [6], coarse grained [14] and fine grained models [17]. Many of the existing approaches try to achieve their targets using MPI. However, MPI is not flexible enough for handling heterogeneity and failures, which are common features of data centers, the increasing popular computation platform.

MapReduce is a simple programming model for developing distributed data intensive applications in data centers. Since it was proposed by Google for cluster of commodity machines, there have been many following projects. For instance, Phoenix [4] is a MapReduce model designed for the shared memory architecture, while Hadoop [1] is an open source implementation of MapReduce designed as a general

distributed computing platform. MRPSO [2] utilizes the MapReduce model to parallelize a computing-intensive application, Particle Swarm Optimization.

Compared with the above work, we are not aiming to invent new parallel genetic algorithms. Instead, we try to simplify the difficulties of developing distributed genetic algorithms. To the best of our knowledge, MRPGA is the first work on using MapReduce to parallelize GAs. Similar to MRPSO, MRPGA is also targeting computationally intensive problems. However, GA applications cannot be directly expressed by MapReduce. Therefore, MRPGA makes an extension to the MapReduce model, which can naturally express GA applications and automatically parallelize the execution.

8. Conclusions

This paper mainly addresses the challenge of using the MapReduce model to parallelize GAs. As a stateless programming model, MapReduce cannot directly express GAs. To achieve our target, we have extended MapReduce by adding a second reduce phase and a special optimization on the merge phase for the added reduce operation. This extension makes PGAs can benefit from the MapReduce model on handling heterogeneity and failures. The evaluation on the runtime system was conducted and the benefits are presented. Based on our successful start of MapReduce-based PGAs, in the future, we endeavor to explore ways of easily parallelizing other population-based solutions.

Acknowledgements

This work is partially supported by research grants from the Australian Research Council (ARC) and Australian Department of Industry, Innovation, Science and Research (DIISR). We thank Srikumar Venugopal, Mukaddim Pathan, Alexandre di Costanzo, James Andrew Broberg and Michael Kirley for their comments on improving the quality of the paper.

References

[1] Apache. Hadoop. *http://lucene.apache.org/hadoop/*.

[2] A. W. McNabb, C. K. Monson, and K. D. Seppi, *Parallel PSO Using MapReduce*, Proc. of the Congress on Evolutionary Computation, Singapore, 2007.

[3] A. Weiss. *Computing in the Clouds*. netWorker, 11(4):16-25, Dec. 2007.

[4] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, *Evaluating MapReduce for Multi-core and Multiprocessor Systems*, Proc. of the 13th Intl. Symposium on High-Performance Computer Architecture, USA, 2007.

[5] D. A. Patterson, *Technical perspective: the data center is the computer*, Communications of the ACM, 51-1, 105, January 2008.

[6] D. Lim, Y.S. Ong, Y. Jin, B. Sendhoff, and B.S. Lee, *Efficient Hierarchical Parallel Genetic Algorithms using Grid computing*, Future Generation Computer Systems, Vol. 23, No. 4, pp 658-670 , Elsevier Science Publishers, 2007.

[7] E. Alba, C. Cotta, *The on-line tutorial on evolutionary computation*, http://www.lcc.uma.es/~ccottap/semEC/.

[8] E. Alba and J. M. Troya, *A Survey of Parallel Distributed Genetic Algorithms*, Complexity 4(1999), 31-52.

[9] H. K. Ng, D. Lim, Y. S. Ong, B. S. Lee, L. Freund, S. Parvez and B. Sendhoff, *A Multi-cluster Grid Enabled Evolution Framework for Aerodynamic Airfoil Design Optimization*, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, Vol. 3611, 2005.

[10] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, Proc. of the 6th Symposium on Operating System Design and Implementation, USA, 2004.

[11] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, *Scalable multiobjective optimization test problems*. Proc. of Congress on Evolutionary Computation, 2002.

[12] M. Kirley, R. Stewart, *An analysis of the effects of population structure on scalable multiobjective optimization problems*, Proc. of SIGEVO Genetic and Evolutionary Computation Conference, UK, 2007.

[13] R. Buyya, C. S. Yeo, and S. Venugopal, *Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities*, Proc. of the 10th IEEE International Conference on High Performance Computing and Communications, China, 2008.

[14] S. C. Lin, W. F. Punch, and E. D. Goodman, *Coarse-grain parallel genetic algorithms: Categorization and new approach*, Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing, 1994.

[15] S. Luke, *Genetic programming produced competitive soccer softbot teams for robocup97*, Proc. of the 3rd Annual Genetic Programming Conference, USA, 1998.

[16] T. Back, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford Univ. Press, New York, 1996.

[17] T. Maruyama, T. Hirose, and A. Konagaya, *A Fine-grained Parallel Genetic Algorithm for Distributed Parallel Systems*, Proc. of the 5th International Conference on Genetic Algorithms, USA, 1993.

[18] X. Chu, K. Nadiminti, J. Chao, S. Venugopal, and R. Buyya, *Aneka: Next-Generation Enterprise Grid Platform for e-Science and e-Business Applications*, Proc. of the 3rd IEEE International Conference and Grid Computing, India, 2007.

[19] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer, Germany, 1996.