2014 IEEE 13th International Symposium on Network Computing and Applications

A Resilient Hierarchical Distributed Loop Self-Scheduling Scheme for Cloud Systems

Yiming Han and Anthony T. Chronopoulos

Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, USA
Email: yimingmining@gmail.com, antony.tc@gmail.com

***Abstract*—In heterogeneous distributed cloud systems, load balance, communication and synchronization overhead must be taken considered. A hierarchical distributed loop self-scheduling scheme is effective and efficient for scientific loop applications. In this paper, we propose a resilient hierarchical distributed loop self-scheduling algorithm suitable for heterogeneous cloud systems. This algorithm is intended to enable the algorithm to continue to work in the event that some virtual machines (VMs) are too slow or cease to respond. We tested our algorithm in a heterogeneous cloud system. The results show that our algorithm can achieve normal operation and good performance.**

***Keywords*-Self-Scheduling; Distributed; Hierarchical; Resilient; Cloud System.**

I. Introduction

Scientific applications usually contain compute-intensive loops. Self-scheduling schemes were proposed in the past that partition independent loop iterations into chunks and allocate them to processors in order to reduce the overall computing time. Dynamic self-scheduling algorithms (or schemes) assign various chunks to each processor and they are well discussed to achieve good load balance on distributed systems such as clusters, grids (see e.g. [1], [2], [3], [4], [5] and refs therein).

Fault tolerance in conjunction with loop self-scheduling has been studied in the past (see e.g. [6], [7] and refs therein. MPI is a middleware of choice for distributed systems with fault tolerant capabilities (see e.g. [8] and refs therein).

Cloud systems are based on a large number of computers connected through a communication network, with additional capability to provision resources (such as VMs, memory, network) to users requests in a flexible and automated manner. They provide scalable, flexible, reliable and on demand computing and storage resources over a network. There are many scientific computation-intensive and data-intensive applications running on cloud systems [9]. The availability and performance of virtual machines in cloud systems can change over time. Thus, cloud systems are considered as a dynamic heterogeneous distributed system. Load balance, communication and synchronization overhead are important factors. Some studies of fault tolerance in clouds exist (e.g. see [10] and refs therein).

The hierarchical distributed self-scheduling schemes work well in cloud system [11] [12]. However, we should also consider fault tolerance. When a worker node fails to respond in the event of an unexpected problem or error, a fault tolerant scheme may be notified of the problem and prepare to take correcting actions to survive from faults. In current cloud systems, single point or multiple points failure are a common occurrence. Although there is hardware level fault tolerance, the program without fault tolerance may not continue to process. In this paper, we propose a resilient hierarchical distributed loop self-scheduling scheme for cloud systems. This scheme continues to work when the computing environment is dynamic and the nodes are to slow due to heavy load or cease to respond. We make an experimental study of our proposed algorithm in a heterogeneous cloud system.

The rest of the paper is organized as follows. In Section II, we review hierarchical distributed loop scheduling schemes. In Section III, we describe new resilient hierarchical distributed schemes. In Section IV, experiments and results are presented. Section V contains conclusions and future work.

II. Hierarchical distributed schemes

In the past, we proposed a hierarchical model in order to obtain scalability in distributed loop scheduling schemes. Figure 1 shows a two-level of hierarchical distributed scheme, one *supermaster* and two *master* nodes. The task scheduler resides in the *supermaster* and it uses distributed scheduling schemes (DTSS/DFSS/DGSS) [12] to compute coarse-grained chunks for each *master* and send to master nodes' Task Pools. When the Task Pool of a *master* node is empty, it asks for more work (from the *supermaster*) in order to fill the Task Pool until there is no more work. The *master* accepts a *worker* request, places it into the request queue and gets a fine-grained chunk from the Task Pool and serves the top request from Request Queue. Also, *master* nodes collect the computed results from *workers*. There are

978-1-4799-5393-6/14 $31.00 © 2014 IEEE
DOI 10.1109/NCA.2014.18

IEEE computer society

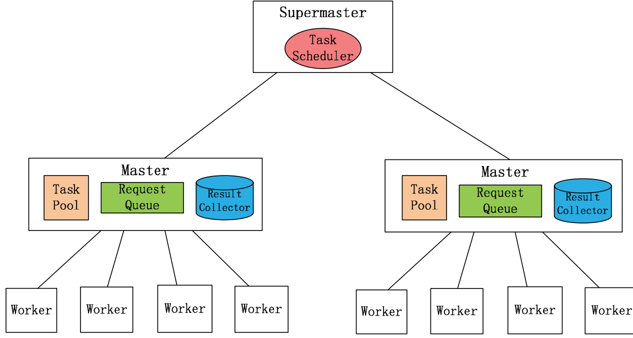


Figure 1. Hierarchical Distributed Model

multiple Request Queues and Result Collectors distributed in different *master* nodes, which can share the responsibilities.

III. RESILIENT HIERARCHICAL DISTRIBUTED SCHEMES

In this section we propose a new resilient hierarchical distributed self-scheduling algorithm. The pseudocode is shown in Algorithm 1, 2 and 3. The algorithm is inspired from past results on an adaptive self-scheduling algorithm proposed for grid systems for simple master-worker models in [6]. We also take advantage of the fault-tolerance approach for MPI middleware based on the master-worker MPI intercommunicators error-handling [8]. This error-handling is based on the continuous monitoring of the correct function of the MPI intercommunicators by the communication subsystem of the distributed system.

We assume that the supermaster is fault-free because modern cloud systems have single point failure solutions. The masters or workers can fail their work is assigned to other masters or workers. So the computation will be completed as long as one master and some of its workers do not fail till the end. We do not consider spawning new workers to replace failed ones.

At the supermaster side, in Algorithm 1, the first step generates the simulated execution environment, a heterogeneous computing environment. Supermaster nodes accept the registrations of master nodes and collect their current available computing powers. With the available powers and input computing job, the supermaster node generates large scheduling chunks for these master nodes. While the computing job is not finished, the supermaster node picks up one master node from the request queue and applies a small benchmark to measure the available computing power. We chose matrix multiplication with small input size as a benchmark. If the master node's available computing power is zero or the worker nodes managed by the master node are unresponsive (or faulty), the supermaster nodes saves the current status and chooses another master node in the request queue. If instead a master node can do some work, the supermaster node may assign a "large" scheduling ("master-level") chunk to the master node. A master-level chunk is a chunk assigned by the supermaster to the masters. It is sufficiently large to be subdivided into several "worker-level" chunks to be assigned to by the master to its workers.

Algorithm 2 shows the master node side. At first, a master node initializes a simulated execution environment. Then the master node accepts registrations from worker nodes and collects their available computing powers to report to supermaster nodes for computing the scheduling chunks. While the input job that is being computed is not finished, the master node requests a scheduling chunk from the supermaster node with some key parameters. These key parameters are the current available computing powers and the number of remaining tasks. These parameters are needed in computing the new chunks. After that, the master node computes "small" scheduling (worker-level) chunks for worker nodes. The master node picks up one requesting worker from the queue and applies a benchmark (a small size matrix multiplication) to measure the worker's available computing power. If the computing environment is changed or the worker is dead, the master node saves and updates the current status. Then the master node reschedules the remaining chunks with new available computing powers and generates new scheduling chunks. This way the load balance is improved. If the worker's available computing power is not changed, the master node may assign a scheduling chunk to the worker node.

A worker node starts with initialization of a heterogeneous environment, in Algorithm 3. Then a worker node registers at a master node with its current available computing power. When a worker node is idle, it requests more work with its current available computing power measured by a small benchmark. After getting a chunk, it completes the assigned computing task and returns results to master node.

Algorithm 1 Supermaster side

- *Initialize simulated execution environment*;
- *Accept master nodes registrationsand available powers*
- *Generate large scheduling chunks*;

while *there are unassigned large chunks* **do**
  - *Pick up one master from requesting queue*;
  - *Get updated on available power of the master*;
  if *the master node's available power is zero or it has failed* **then**
    - *Save current status*;
    - *Pick up the next master*;
  else
    - *Assign chunk to the requesting master*;
  end if
end while

Algorithm 2 Master side

- *Initialize simulated execution environment*;
- *Accept worker nodes registrationsand available powers*
while *there are unassigned large chunks* **do**
  - *Master node calculates total available powers of workers*;
  - *Master node requests a large chunk from supermaster*;
  - *Master node generates small scheduling chunks (FSS/GSS/TSS)*;
  while *there are unassigned small chunks andnot−failed workers exist* **do**
    - *Pick up one worker from requesting queue*;
    - *Check if worker is not failed*;
    if *the worker is unresponsive or the environment is changed* **then**
      - *Save current status*;
      - *Master node regenerates the scheduling chunks for unassigned work (FSS/GSS/TSS)*;
      - *Pick up the next worker*;
    else
      - *Assign chunk to the requesting worker*;
    end if
  end while
end while

Algorithm 3 Worker side

- *Initialize simulated execution environment*;
- *Register at supermaster node*;
while *there are unassigned small chunks* **do**
  - *Apply a small benchmark to measure the current computing power*;
  - *Worker node requests a small chunk from master node*;
  if *the worker gets a chunk from its master node* **then**
    - *Compute and return results*;
  end if
end while

IV. Experimental Evaluation

A. Cloud Environment

FlexCloud is our cloud system platform. FlexCloud is a private cloud platform for academic research in Institute for Cyber Security (ICS), University of Texas at San Antonio. It is capable of meeting the demands of world-class academic research. It offers fast provisioning, reliability, data security and observability, good balance between performance and security, and easy scalability. FlexCloud includes:

- 5 Racks of Dell R410, R610, R710, and R910s consisting of 748 processing cores, 3.44TB of RAM, and 144TB of total storage.
- Redundant 10GB network connectivity provides high performance access between all nodes. The Network Switch is a Dell Switch and it is connected via a High Speed (greater than 1GB/s) Fiber Optic link to the Main ICS Juniper (MX-80) Router.

B. Experiments

112 single core, 512MB virtual machines (VM) are initialized and they are loaded with Ubuntu Linux 12.04 image. In order to build a heterogeneous computing environment, we use Stress [13] to add additional random load on each VM. Stress is a deliberately simple workload generator. Stress was developed by University of Oklahoma. It imposes a configurable amount of CPU, memory, I/O, and disk stress on the system. The application is Mandelbrot Set with size of 20K * 20K.

In the experiments, we applied additional (i.e. outside the simulation) computing loads (Stress [13]) to make the computing environment "heterogeneous". We also applied "heavier" computing loads in some worker nodes to simulate an environment with too slow (or unresponsive) nodes in order to test the resilient function of the schemes. We set a threshold equal to 0.15, in the experiments. This means that if the current computing power of a worker node is less than 15% of its original computing power, it is regarded to be unresponsive and can not receive workload until its computing power is more than 15%. The other point is that if we continue to assign workload to such a slow worker node, it will be the bottleneck in the future and the other nodes will wait for its completion of assigned tasks. Thus, we treat these worker nodes as unresponsive (or faulty). When there are faulty nodes in the computing environment, we collect the parameters and regenerate scheduling chunks.

For better testing and understanding the resilient function of the new schemes, we choose different numbers of faulty worker nodes by different fault rates, 12.5% and 25%.

All the schemes are implemented by C++ and MPI. All timings are in seconds.

C. Results

Figure 2 illustrates the total execution time of HDTSS with 16 workers using the standard algorithm and the new algorithm. We can observe that when there are no faulty nodes, the total execution time with different number of master nodes is the same. However, the new algorithm are much better when the fault rate is 12.5% and 25%. This may happen when the load balancer of the current computing system doesn't work well and make some computing nodes are heavily loaded. These slow nodes may cause bottleneck in the system and they can create load imbalance when we are applying regular scheduling schemes. The new algorithm avoids the bottlenecks and completes the work in a reasonable amount of time.

Figure 3 shows the total execution time comparison of our algorithm between the faulty and the non-faulty environment. There are 2 faulty workers in 16 workers hierarchical distributed trapezoid self-scheduling scheme (HDTSS) experiment and there are 4 faulty workers in 32 workers experiment. The fault rates are 12.5% and 25%. We observe that the resilient scheme can survive and continue to finish the computing work when some problems happened. When 12.5% workers nodes are down, the total execution time increase is between 13% to 17%. And for 25% fault rate, the increase is between 28% to 36%. The reason why the increase is higher is due to the rescheduling overhead. When there are faulty nodes, the scheduling schemes will collect the current parameters and generate new scheduling chunks for the remaining workload. The total increase is similar to the fault rate and is not much higher. Thus, the new schemes do not introduce much rescheduling overhead into the final results and they are effective.

We also carried out experiments on different schemes, the comparison between HDFSS, HDGSS and HDTSS. Figure 4 shows the total execution time between the new schemes HDFSS, HDGSS and HDTSS with 16, 32 and 64 worker nodes. From the results, we observe that all the new schemes work when there are faulty nodes when faulty nodes appear in the middle of the computation. Moreover, the new schemes can work well when more worker nodes are added into system. HDTSS is the best because it generates fewer chunks to reduce the synchronization overhead. HDFSS is better than HDGSS due to the smaller early chunks. However, when the fault rate increased, HDGSS became better than HDFSS. The reason is that after rescheduling the rest of workload, the early chunks are not too large to harm the load balance and HDGSS generates a smaller number of scheduling chunks than HDFSS.

Finally, we tested the scalability of the new schemes. Figure 5 presents the total execution time of the resilient HDTSS with two master nodes, when the number of workers scale from 16 to 64. It is observed that the overall performance and the scalability became better when we increase the number of worker nodes. The 4 Masters' model's improvement is the best and 2 Masters' model is better than the simple Master-Worker model. The hierarchical distributed scheme introduce multiple master nodes and it reduces centrality of traditional master worker model. Thus, the scalability is good. Finally, the resilient hierarchical distributed scheme doesn't introduce too much rescheduling and synchronization overhead.

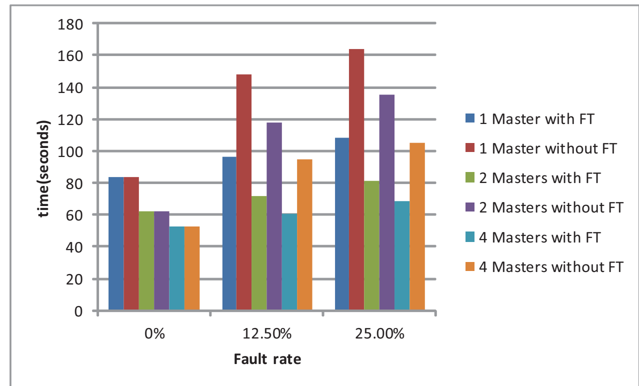


Figure 2. The total execution time of HDTSS with 16 workers using standard algorithm and resilient algorithm.

V. Conclusions and Future Work

In this paper, we propose a resilient hierarchical distributed scheme. We simulated a heterogeneous computing environment on cloud systems. We assume that some nodes are faulty, if they are too slow. The new schemes can survive from partial faulty nodes and reschedule the rest of workload to finish the job. The experimental results show that the performance and scalability of the new schemes is still good because it introduce few rescheduling and synchronization overhead. We also compared different loop self scheduling schemes to show their effectiveness. In future, we plan to extend our schemes to tolerate real faults of machines.

Acknowledgement

We gratefully acknowledge the following: (i) support by NSF grant (HRD-0932339) to the University of Texas at San Antonio; and (ii) time grants to access the facilities of Institute for Cyber Security(ICS) of University of Texas at San Antonio.

References

[1] C.-C. Wu, C.-T. Yang, K.-C. Lai, and P.-H. Chiu, "Designing parallel loop self-scheduling schemes using the hybrid MPI and openMP programming model for multi-core grid systems," *The Journal of Supercomputing*, vol. 59, pp. 42–60, 2012.

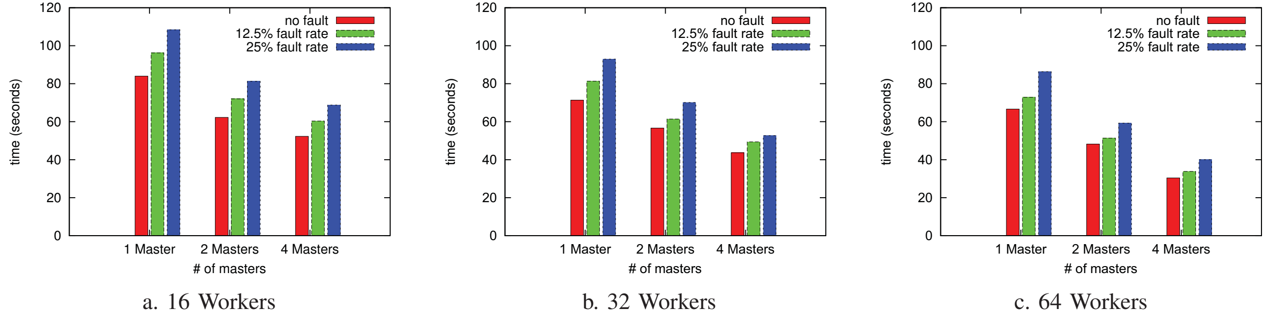
[2] Y. He, J. Liu, and H. Sun, "Scheduling functionally heterogeneous systems with utilization balancing," *IEEE International Parallel Distributed Processing Symposium, IPDPS'11*, pp. 1187–1198, 2011.

[3] S. Penmatsa, A. Chronopoulos, N. Karonis, and B. Toonen, "Implementation of distributed loop scheduling schemes on the teragrid," *IEEE International Parallel and Distributed Processing Symposium, IPDPS'07*, pp. 1–8, 2007.

[4] A. T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali, "Distributed loop-scheduling schemes for heterogeneous computer systems," *Concurrency and Computation:Practice and Experience*, pp. 771–785, 2006.

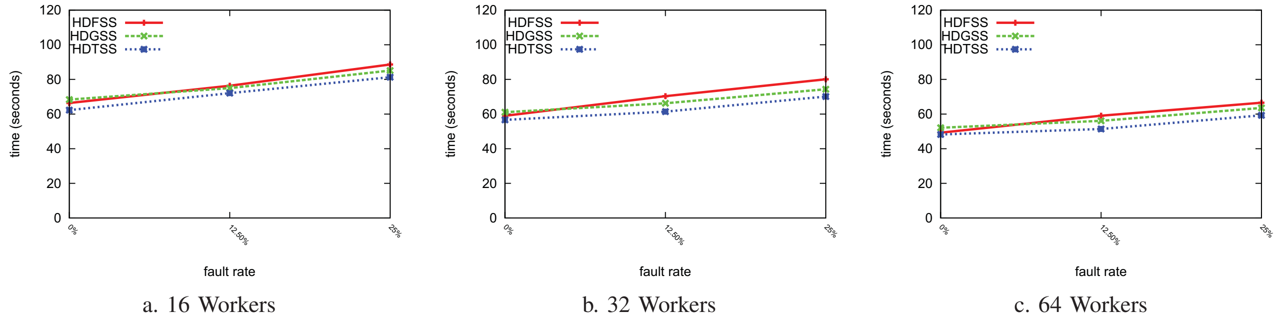
[5] Y.-M. Wang, H.-H. Wang, and R.-C. Chang, "Hierarchical loop scheduling for clustered numa machines," *Journal of Systems and Software*, pp. 33–44, 2000.

[6] J. Díaz, C. Muñoz Caro, and A. Niño, "An adaptive approach to task scheduling optimization in dynamic grid environments," *In GCA*, pp. 23–29, 2009.



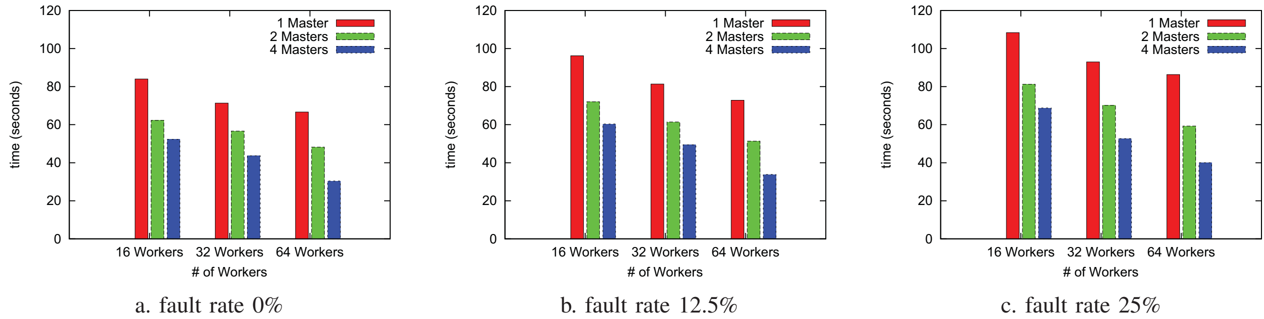
a. 16 Workers b. 32 Workers c. 64 Workers

Figure 3. The total execution time of HDTSS on faulty environment and non-faulty environment.



a. 16 Workers b. 32 Workers c. 64 Workers

Figure 4. The total execution time between resilient HDFSS, HDGSS and HDTSS with 16, 32 and 64 worker nodes.



a. fault rate 0% b. fault rate 12.5% c. fault rate 25%

Figure 5. The total execution time between the resilient HDFSS, HDGSS and HDTSS with 16, 32 and 64 worker nodes.

[7] R. C. Wang Yizhuo, Alexandru Nicolau and A. V. Veidenbaum, "A fault tolerant self-scheduling scheme for parallel loops on shared memory systems," *2012 19th International Conference on High Performance Computing, HiPC'12*, pp. 1–10, 2012.

[8] G. William and E. Lusk, "Fault tolerance in message passing interface programs," *International Journal of High Performance Computing Applications*, pp. 363–372, 2004.

[9] L. Wang, J. Tao, M. Kunze, A. Castellanos, D. Kramer, and W. Karl, "Scientific cloud computing: Early definition and experience," *10th IEEE International Conference on High Performance Computing and Communications, HPCC'08*, pp. 825–830, 2008.

[10] R. Jhawar, V. Piuri, and M. Santambrogio, "Fault tolerance management in cloud computing: A system-level perspective," *IEEE Systems Journal*, pp. 288–297, 2013.

[11] Y. Han and A. T. Chronopoulos, "Distributed loop scheduling schemes for cloud systems," *27th IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum, IPDPSW'13*, pp. 20–26, 2013.

[12] ——, "A hierarchical distributed loop self-scheduling scheme for cloud systems," *The 12th IEEE International Symposium on Network Computing and Applications, NCA'13*, pp. 7–10, 2013.

[13] Stress, http://people.seas.harvard.edu/~apw/stress/.