Int J Parallel Prog (2017) 45:595–611
DOI 10.1007/s10766-016-0434-5

Scalable Loop Self-Scheduling Schemes for Large-Scale Clusters and Cloud Systems

Yiming Han[1] · **Anthony T. Chronopoulos**[1]

Received: 28 June 2015 / Accepted: 26 April 2016 / Published online: 11 May 2016

Abstract Cloud systems have demonstrated the powerful computation and storage capability in many scientific applications. In this paper, we propose a class of scalable distributed loop self-scheduling schemes to achieve good load balancing and scalability. We implemented these schemes on a large-scale cluster and on a heterogeneous cloud system. The schemes consider the distribution of the output data, which can help reduce communication overhead and improve scalability. We evaluated the schemes using four scientific computations: Mandelbrot set, adjoint convolution, matrix multiplication and quick sort. The results show that the new schemes achieve better load balancing, better scalability and better overall performance than standard distributed loop self-scheduling schemes.

Keywords Self-scheduling · Distributed · Hierarchical · Scalable · Cloud system

1 Introduction

Scientific loops are usually computation-intensive which may take a long execution time. Distributed systems, such as cluster, grid and cloud, are widely used in many scientific loops. Thus, scientific loop parallelization, which schedules and assigns work to processors/workers, becomes an important issue. One of the difficult problems is load balancing. Efficient loop scheduling schemes can improve the utilization of resources and minimize the total execution time. Loop scheduling can be categorized into static and dynamic. Static scheduling schemes determine the task allocation to the processors prior to the execution of the application loop. It has lower scheduling

✉ Anthony T. Chronopoulos
Anthony.Chronopoulos@utsa.edu

1 Department of Computer Science, University of Texas at San Antonio, San Antonio, TX, USA

overhead, but suffers from load imbalance. Dynamic scheduling (or self-scheduling) is an automatic loop scheduling method in which idle processors request new loop iterations to be assigned to them during run time (the execution of the application). This approach has better load balancing with less scheduling overhead, which is popular in work distribution on distributed systems.

Cloud computing is emerging as a powerful technology to meet the requirements for high-performance computing and massive storage. It provides scalable, flexible, reliable and on demand computing and storage resources over a network. Many scientific computation-intensive and data-intensive applications are accomplished on cloud systems [1,2]. A cloud system could be considered as a dynamic heterogeneous distributed system. A cloud system may also provide a homogeneous computing environment at the start. However, it may be upgraded and replaced to exhibit more heterogeneity [3]. The availability and performance of virtual machines can change over time. Also a cloud system is transparent to cloud users, which means cloud users still perceive it as a homogeneous environment. Thus, it is likely to create load imbalance if we ignore the heterogeneity. Previous research reported some schemes on a heterogeneous cluster and grid systems. Also, [4] tested a distributed scheme for cloud systems.

In general, three aspects must be considered in scientific loops self-scheduling schemes on distributed systems: load imbalance, communication and synchronization overhead. In distributed computing environments, the communication often becomes the bottleneck. The reasons are: the computing nodes may be located on the same rack, may be part of the same cluster sharing a common LAN, or may be on separate clusters communicating through a slow network [5].

Many modern high performance computing platforms, such as clusters grids and clouds, can be scaled to thousands of parallel processors, servers and workstations. Thus, scalability becomes an important issue which should be taken into consideration. In the static scheduling of high performance computing application programs extra resources may be allocated which can cause load imbalance and low speedup. This is especially true for some nested loops when executed on large-scale clusters. Previous research, has developed some loop scheduling schemes to get good performance and load balancing for small-scale clusters with multi-core processors. A scalable two Masters model with small number of workers on a small size application is proposed in [4].

In this paper, we propose a hierarchical distributed loop self-scheduling scheme. We implemented this scheme on a homogeneous large-scale cluster and on a heterogeneous cloud system. The hierarchical scheme is based on a supermaster/master/worker model which can reduce the communication overhead and synchronization overhead. Preliminary results have been published in [6,7]. We implemented these schemes on a large-scale cluster of Texas Advanced Computing Center, University of Texas at Austin. We also implemented these schemes on a Joyent Cloud system [8]. Our experiments validate the scalability and the better overall improved performance of the proposed schemes.

The rest of the paper is organized as follows. In Sect. 2, we present the related work. In Sect. 3, we review simple loop self-scheduling schemes. In Sect. 4, we describe the hierarchical distributed schemes. In Sects. 5 and 6, experiments and results are presented. Section 7 contains conclusions and future work.

2 Related Work

Previous research [9–12] (and references therein) proposed some well-known loop scheduling schemes to assign varying task chunks to each processor. These loop scheduling schemes can be categorized into static and dynamic. Static scheduling schemes determine the task allocation to the processors prior to the execution of the application. Dynamic scheduling (or self-scheduling) is an automatic loop scheduling method in which idle processors request new tasks to be assigned to them during run time of the execution of the application. An adaptive chunk self-scheduling scheme is proposed (in [13]) to reduce the scheduling overhead. In [14,15], the authors have proposed new improved self-scheduling schemes named NGSS and ANGSS that are well-suited for grids. A two-phase scheme is proposed to solve parallel regular loop scheduling problem in heterogeneous grid computing environments in [16]. In [17–19] new results are presented for loops with dependencies. Recent research results [20,21] have been reported for designing loop self-scheduling methods for grids. In [10,22,23], the heterogeneity of different cluster systems was considered, in order to get better load balancing. Multi-dimensional loop scheduling schemes were studied in [24,25].

Presently, cloud computing platforms are growing in popularity. They provide scalable, flexible, reliable and on-demand computing and storage resources over a network. There are some commercial cloud providers, such as Amazon EC2, Microsoft Azure, Salesforce Service Cloud and Google Cloud. Some open source cloud projects for research and development also exist, for example, OpenStack, Eucalyptus, CloudStack and Ganeti [26] and references there in. There is also much ongoing research for cloud systems. Energy consumption of large scale data centers cloud systems has become a prominent problem and receives much attention. A hierarchical scheduling algorithm for applications, to minimize the energy consumption of both servers and network devices is proposed in [27]. Cloud computing can be used for solving some computational intensive jobs in high performance computing research area. Clouds are becoming an alternative to clusters, grids, and parallel production environments for scientific computing applications [28–30]. However, virtualization and resource time-sharing may introduce performance overheads for the demanding scientific computing workloads. The performance of cloud computing services for scientific computing workloads is studied in [28]

For high performance computing applications, we can use cloud to virtualize clusters on cloud systems. These virtual machines can share the same physical hardware or different physical hardware with various system load and user load and cloud system use a fair-share balancing algorithm that gives equal time to each virtual machine. However, because of limited resources, the virtualized cluster is not private and the resources are shared by many users, which means the virtualized cluster may act as a heterogeneous computing environment at running time. Thus, the heterogeneity should be taken into account to improve resource utilization and reduce load imbalance. MapReduce [31] is a general concurrent programming framework for scheduling job-tasks on cloud systems. Previous research [29,30] reported that the performance on virtual machines is lower than the physical system. They analyzed message passing

(MPI) parallel applications on different cloud systems and reported that communication overhead is a substantial slowdown factor for cloud systems.

3 Loop Self-Scheduling Schemes for Distributed Systems

Load balancing in distributed systems is a very important factor in achieving near optimal execution time. To obtain load balancing, loop scheduling schemes must take into account the processing speeds of the workers or processors forming the system. The processors' speeds are not precise, since memory, cache structure and even the program type may affect the performance of processors. However, one could run experiments to obtain estimates of the throughputs and one could show that these schemes are quite effective in practice.

We present the distributed versions Distributed Factoring Self-Scheduling Scheme (DFSS), Distributed Guided Self-Scheduling Scheme (DGSS) , Distributed Trapezoid Self-Scheduling Scheme (DTSS) of the following well known scheduling schemes Factoring (FSS), Guided (GSS) and Trapezoid (TSS) (see [10]). A distributed scheme is obtained by modifying the chunk allocation mechanism of a standard scheme in order to take into account the computing speed of each worker in the system. The distributed algorithms work use the same allocation mechanism as the corresponding simple schemes (i.e.FSS, GSS, TSS) but are designed to take into account the variable computing speeds of the computers or nodes of the cluster system. At first we introduce some notation and terminology that will be used in the rest of the paper.

3.1 Terminology

At first, we present the notation for the simple schemes (i.e.FSS, GSS,TSS).

- I is the total number of iterations or tasks of a parallel loop;
- p is the number of workers (i.e. processors) in the parallel or distributed system which execute the computational tasks;
- P_1, P_2,..., P_p represent the p workers in the system;
- A few consecutive iterations are called a *chunk*. C_i is the chunk-size at the i-th scheduling step (where: $i = 1, 2, \ldots$);
- N is the number of scheduling steps;
- t_j, $j = 1, \ldots, p$, is the execution time of P_j to complete all its tasks assigned to it by the scheduling scheme;
- $T_p = \max_{j=1,\ldots,p} (t_j)$, is the parallel execution time of the loop on all p workers;

In a generic self-scheduling scheme, at the i-th scheduling step, the master computes the chunk-size C_i and the remaining number of tasks R_i:

$$R_0 = I, \qquad C_i = f(R_{i-1}, p), \qquad R_i = R_{i-1} - C_i \tag{1}$$

where $f(.,.)$ is a function possibly of more inputs than just R_{i-1} and p. Then the master assigns to a worker processor C_i tasks. For the simple schemes FSS, GSS,TSS the function "f" can be found in [10]. The same function "f" is used in the distributed schemes DFSS, DGSS,DTSS but the workers variable computing speeds are used. We next show how this is done.

- $V_j = Speed(P_j)/min_{1 \leq i \leq p}\{Speed(P_i)\}$, $j = 1, \ldots, p$, is the virtual power of P_j (computed by the master), where Speed(P_j) is the processing speed of P_j. That is a standardized computing power in the current cluster.
- $V = \sum_{j=1}^{p} V_j$ is the total virtual computing power of the cluster.
- DC is the distributed chunk size for one worker request, in a single scheduling step of distributed self-scheduling scheme.

3.2 Algorithm

We next present the distributed algorithm that takes into account the virtual computing power of the workers.
Master:

(1) Compute V_j for each worker
  (a) Receive $Speed(P_j)$;
  (b) Compute all V_j;
  (c) Send all V_j ;
(2) Assign work and get the results
  (a) While there are unassigned tasks, if a request arrives, put it in the Request Queue.
  (b) Pick a request from the queue and get its virtual power V_j. If there are computed results in this request, Result Collector receives them first. Then Task Scheduler compute the next chunk size DC to assign. The followings are the DTSS, DFSS and DGSS algorithms to compute the next chunk DC:

DTSS:
$Current$ is chunk size in the current step of TSS.
Initialization: $F = \left\lfloor \frac{I}{2V} \right\rfloor$, $L = 1, N = \left\lceil \frac{2*I}{(F+L)} \right\rceil$,
$D = \left\lfloor \frac{(F-L)}{(N-1)} \right\rfloor$, $Current = F$

Algorithm 1 Calculate DC

```
$DC = 0$;
for $k = 1 \rightarrow V_j$ do
    $DC = DC + Current$;
    $Current = Current - D$;
end for
return $DC$;
```

DFSS:
DC_{sum} is the assigned work in the current stage.
Initialization: $R = I, \alpha = 2.0, DC_{sum} = 0$

Algorithm 2 Calculate DC

```
$DC = \lceil R/(\alpha V) \rceil * V_j$;
$DC_{sum} = DC_{sum} + DC$;
**if** (Master has assigned all the work in the current stage) **then**
    { Goto next stage and update the remaining work. }
    $R = R - DC_{sum}$;
    $DC_{sum} = 0$;
**end if**
**return** $DC$;
```

DGSS:
Initialization: $R = I$

Algorithm 3 Calculate DC

```
$DC = \lceil R/(A) \rceil * V_j$;
$R = R - DC$;
**return** $DC$;
```

Worker :

```
(1) Send $Speed(P_j)$;
(2) Send a request;
(3) Wait for a reply;

IF (There is unassigned work)
{
    Compute the new work;
    Return the results and send another request;
    Go back to (2);
}
ELSE
    Terminate;
```

4 Hierarchical Distributed Self-Scheduling Schemes

When considering a scheduling scheme using the Master–Worker model for concurrent computing, several issues must be considered: the load balancing, the communication and synchronization overhead.

All the policies, where a single node (the master) is in charge with the work distribution and collecting the results, may cause degradation in performance as the problem size increases. This means that for a large size problem (and for a large number of processors) the master could become a bottleneck. There are two major kinds of overheads in simple Master–Worker architectures. The first one is: if workers send back the computed results, it may take a long time to gather the computed results. The communication overhead is expensive in a distributed memory system, where long communication latency can be encountered. Another kind of overhead occurs when

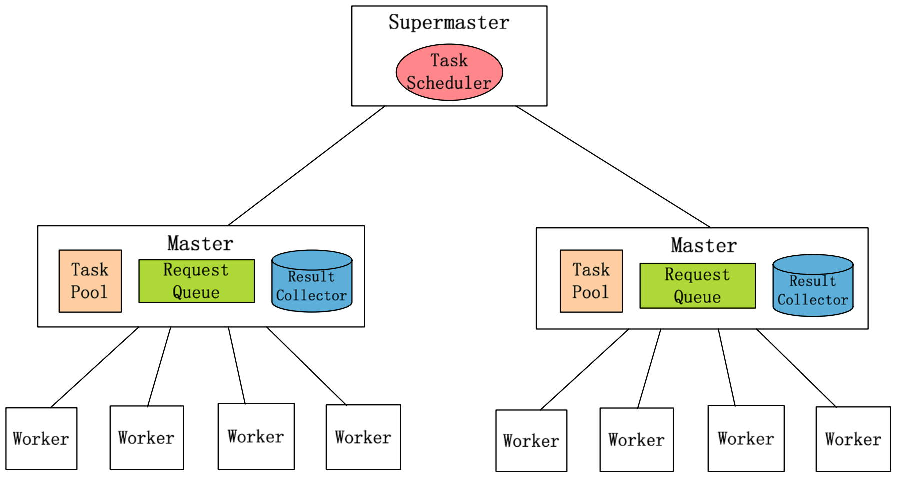

Fig. 1 Hierarchical architecture

many workers send work requests at the same time and only one worker can be served from the request queue and the others have to wait. This is time consuming, especially in the case of a single request queue, when the task scheduler is slow or the scheduling schemes are complicated.

It is known that distributed policies usually do not perform as well as the simple Master-Worker policies (i.e. using a single master), for small problem sizes and small number of workers. This is because the algorithm and the implementation of distributed schemes usually add a non-trivial overhead.

We consider a logical hierarchical architecture as a good model for scalable systems and we propose a new hierarchical approach for addressing the bottleneck problems in the Master-Worker schemes.

Instead of making one master process responsible for all the workload distribution, several master processes are introduced. Thus, the hierarchical structure contains a lower level, consisting of worker processes, and several superior levels, of master processes. On top, the hierarchy has an overall *supermaster*. The workers' role is to perform the computations following a Master-Worker self-scheduling method for the problem that is to be solved. This scheme is called a *Hierarchical Distributed Scheme*.

Figure 1 shows this design for two levels of master processes, one supermaster and two master nodes. The task scheduler resides in the supermaster and it uses distributed scheduling schemes (DTSS/DFSS/DGSS) [4] to compute small scheduled chunks for each master node 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 node accepts a worker request, places it into the request queue and gets a scheduled chunk from the Task Pool and serves the top request from Request Queue. Also, the master node is in charge of gathering the computed results from workers. There are multiple Request Queues and Result Collectors distributed in different master nodes, which can share the responsibilities.

The hierarchical distributed scheduling scheme is described as follows:

- $V_j = Speed(P_j)/min_{1 \leq i \leq p}\{Speed(P_i)\}$, $j = 1, \ldots, p$, is the virtual power of P_j (computed by the master), where Speed(P_j) is the processing speed of P_j. That is a standardized computing power in the current cluster.
- $V = \sum_{j=1}^{p} V_j$ is the total virtual computing power of the cluster.

Supermaster:

(1) Compute V_j for each Worker
  (a) Receive Workers' $Speed(P_j)$ from Masters;
  (b) Compute all V_j;
(2) Assign work to Masters
  (a) While there are unassigned tasks, if a Master request arrives, put it in the queue;
  (b) Pick a request from the queue and get the Workers virtual power V_j under the requesting Master. Using distributed self-scheduling schemes (i.e. DTSS, DFSS and DGSS), compute small scheduled chunks for each Worker under the requesting Master.

Master:

(1) Compute V_j for each Worker
  (a) Receive $Speed(P_j)$ from its Workers;
  (b) Send these $Speed(P_j)$ to Super Master;
(2) Send work request work to Super Master;
(3) Assign work to Workers;
  (a) If there are unassigned tasks, if a Worker request arrives, put it into the Request Queue. Pick a request from the Request Queue, Result Collector receives computing results first. Then get a chunk from Task Pool and send this chunk to requesting worker;
  (b) If there are not unassigned tasks, request more work to Supermaster;
  (c) If there is no work left, go back to (2);

Worker :

```
(1) Send Speed(P_j) to its Master;
(2) Send a request to its Master.
(3) Wait for a reply;

IF (There is unassigned work)
{
    Compute the new work;
    Return the results and send another request;
    Go back to (2);
}
ELSE
    Terminate;
```

5 Description of the Large-Scale Cluster and Cloud System platforms

5.1 Large Scale Cluster

We used as our platform the Ranger supercomputer cluster system located at TACC (Texas Advanced Computing Center) in University of Texas at Austin. The cluster nodes' Operating System is Linux and the nodes are managed by Rocks 4.1 cluster toolkit. Each node has four AMD Opteron Quad-Core 64-bit processors and 16 cores total. The memory limit is 32 GB per node. The nodes are interconnected by InfiniBand technology in a full-CLOS topology which provides a 1GB/sec point to point bandwidth.

5.2 Cloud Environment Platform

We used FlexCloud of Joyent corporation [8] located at Institute for Cyber Security(ICS) at University of Texas at San Antonio. The ICS FlexCloud is one of the first dedicated Cloud Computing academic research environments. It offers significant capacity and similar design features found in Cloud Computing providers, including robust compute capability and elastic infrastructure design. FlexCloud currently includes the following hardware components:

- 5 Racks of Dell R410, R610, R710, and R910s consisting of 748 processing cores, 3.44TB of RAM, and 144TB of total storage.
- Redundant 10 GB 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 1 GB/s) Fiber Optic link to the Main ICS Juniper (MX-80) Router.
- The FlexCloud is powered by Joyent SmartDataCenter [8], which provides the highest performance virtualization and analytics. The Joyent SmartOS includes the following features:
  - Joyent uses the HPC model of management: one headnode PXE boots compute nodes
  - SmartOS is a RAM disk based image (nothing stored on disk) which makes it very easy to upgrade head node and compute nodes
  - SmartOS uses the disks on the local nodes to back the SmartMachines and Virtual Machines using ZFS
  - SmartOS has DTrace which allows for monitoring all VMs with little overhead for maximum observability
  - SmartOS has the best multitenant defenses to prevent one tenant from affecting others on the box

We use five different physical machines on FlexCloud. We created 16 VMs on each physical machine sharing the same LAN. Each VM corresponds to a separate core. The purpose is to show the network heterogeneity in the experiment. The communication overhead between VMs in the same physical machine (intra-node shared memory communication) is lower than in different physical machines (inter-node distributed memory communication). For 64 workers using 4 masters hierarchical distributed scheme, each master has 16 workers. Master VM and its 15 worker VMs are in the same physical machine and the other worker VM is in another physical machine. Most of the communication for the work distribution and the results collection is intra-node shared memory communication, instead of communication across nodes. The result collection communication work is distributed in masters, instead of in a single master node by standard scheme. Each VM is loaded with Ubuntu Linux 12.04 image. Stress [32], a work generator, is used to create a heterogeneous computing environment. 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.

Each worker can get work proportional to its available computing power. The super-master VM resides on the 6th physical machine from the masters and workers. This machine has a large memory and we used no 'Stress' load because we want to minimize the scheduling overhead.

6 Experimental Results

In this section, we compare the performance of the distributed loop self-scheduling schemes using a single master versus the same schemes using the hierarchical model approach. We present the results for the large scale cluster and the cloud system in separate subsections.

6.1 Large Scale Cluster

In this section, we compare the performance of the various schemes, non-hierarchical (single master) and hierarchical (2 masters, 4 masters, 8 masters, 16 masters) and with a number of workers (processors) from 256 to 8,192.

We use the following two applications in this implementation [7]. The outer loops in these applications are partitioned using scheduling and the tasks are assigned to workers. The output results are collected by the masters and can be stored in the file system.

(1) The Mandelbrot set: It is a doubly nested loop without dependencies. The computation of one column of the Mandelbrot matrix is considered the smallest schedulable unit. The Mandelbrot set loop is an irregular loop in terms of unpredictable iteration task sizes. Thus this kind of loop causes load imbalance in the parallel computation. The following loops are used for computing the Mandelbrot set. The Mandelbrot set computation domain is $[-2.0, 2.0] \times [-2.0, 2.0]$ and its size is $200\,\text{K} \times 200\,\text{K}$.

(2) Adjoint convolution: This application involves computation of decreasing task sizes. Thus, it can cause load imbalance in the parallel computation. The adjoint convolution has a size of 800×800 and the arrays are generated randomly.

Unlike TSS, the FSS and GSS, are known to have a number of their (last) scheduling steps of chunk size $= 1$ ([25]). This number of steps equals the number of workers, which is considerable for massive number of workers. So, HDFSS and HDGSS are

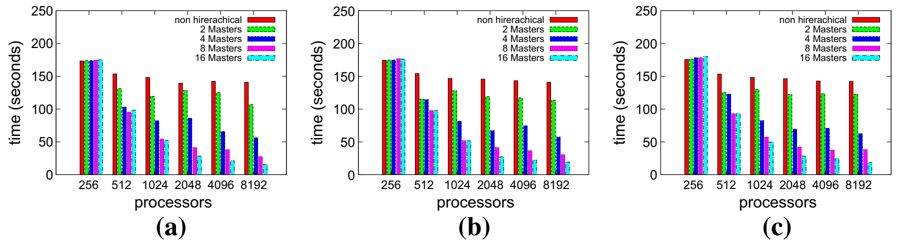

Fig. 2 The performance of Mandelbrot set using hierarchical distributed schemes. **a** HDTSS, **b** HDFSS, **c** HDGSS

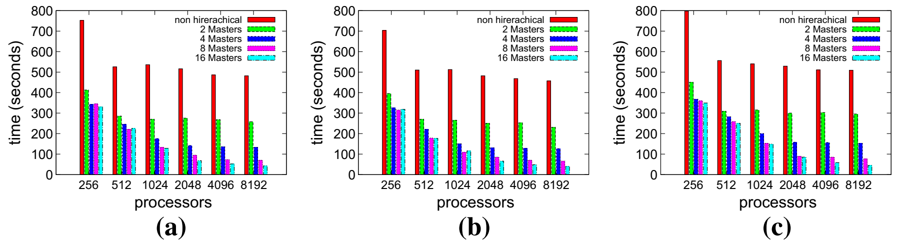

Fig. 3 The performance of adjoint convolution using hierarchical distributed schemes. **a** HDTSS, **b** HDFSS, **c** HDGSS

expected to have similar behavior. In order to avoid this unnecessary synchronization overhead we chose a threshold (>1) for all the methods. We did not try to optimize the selection of threshold. In our experiment, the threshold equals 5, which means the master can not assign a chunk with size <5, except possibly the last chunk.

We test the HDTSS, HDFSS and HDGSS schemes discussed in Sect. 4. All workers are treated (by the schemes) as having the same computing power. The execution time is measured in seconds.

The performance plots presented in Figs. 2 and 3 are organized from left to right in doubling numbers of workers using HDTSS, HDFSS, HDGSS schemes for Mandelbrot set and adjoint convolution. It can be observed that the hierarchical distributed scheme with more master nodes can achieve better performance improvement. The 2-Masters' model scales well upto 512 workers, however past this point the execution time does not decrease as the number of workers increases. The 16-Masters shows the best scalability because when the number of workers doubles, the execution time is halved. For small number of workers (e.g. 256, 512 in Fig. 2a–c) all the hierarchical schemes (HDTSS, HDFSS, HDGSS) have a small discrepancy in the performance between the 4,8 or 16 Masters. For example, one can observe that in the case of 256 workers (Fig. 2a–c) the 2 Masters is the best. This is expected due to the overhead of the hierarchical schemes. The fact that the loop tasks have irregular computational cost makes the discrepancy to appear sometimes. However, once the overhead is overcome the hierarchical schemes show their potential. The load balancing issue can be solved by the original self-scheduling schemes (TSS, FSS and GSS), which have been demonstrated to be effective scheduling schemes in both shared memory systems and distributed memory systems. In our experiments, the performance of HDFSS and

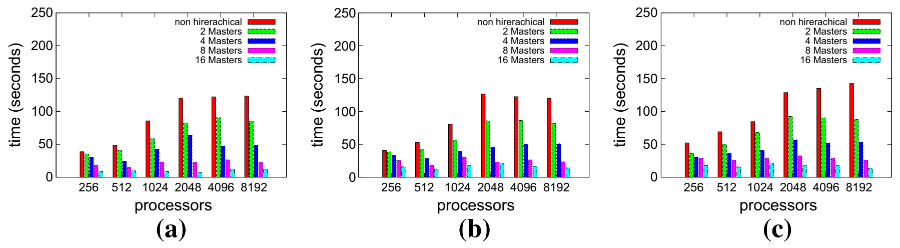

Fig. 4 The non-overlapped communication and synchronization overhead $T'_{overhead}$ of Mandelbrot set. **a** HDTSS, **b** HDFSS, **c** HDGSS

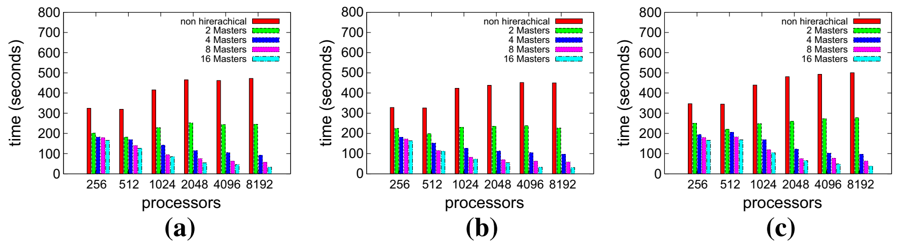

Fig. 5 The non-overlapped communication and synchronization overhead $T'_{overhead}$ of adjoint convolution. **a** HDTSS, **b** HDFSS, **c** HDGSS

HDGSS are a little better than HDTSS because HDFSS and HDGSS may generate more small chunks at the end to balance the workload across the computation. These two schemes introduce more synchronization problems (i.e. more chunks and more work requests). However, hierarchical distributed schemes have distributed queues and these synchronization points take really little time, which can lead to good load balancing.

In Figs. 4 and 5, we show the non-overlapped communication and synchronization overhead ($T'_{overhead}$) with increasing number of workers, $T'_{overhead} = T_{total} - T_{computation}$. In our experiments, there are some overlapping between computation and communication for efficient computing. The computation time can be measured exactly but the total communication overhead is difficult to capture. So we use $T'_{overhead}$ to represent the sum of non-overlapped communication and synchronization overhead. In our results, when more masters are used, $T'_{overhead}$ are smaller. The 16-masters' model has the best performance for our results, because it has more result collectors and distributed task queues residing on master nodes. This helps to reduce the synchronization overhead and especially the communication overhead, which may be the slowest part for large problems in distributed memory systems.

Figures 6 and 7 shows the speedup of the three hierarchical distributed schemes for Mandelbrot set and adjoint convolution. The x-axis represents $\log_2(p)$. The speedup is computed by $S_p = \frac{\hat{T}_1}{T_p}$, $\hat{T}_1$ is the execution time for the non hierarchical distributed scheme with 256 workers, where T_p is the execution time with p workers. It can be observed that as the number of workers increases, the 16-masters' hierarchical distributed scheme scales well upto 8,192 workers. The non hierarchical distributed scheme's scalability is the worst.

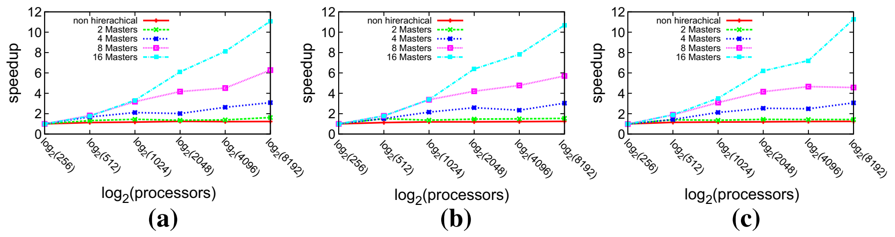

Fig. 6 The speedup of Mandelbrot set using hierarchical distributed schemes. **a** HDTSS, **b** HDFSS, **c** HDGSS

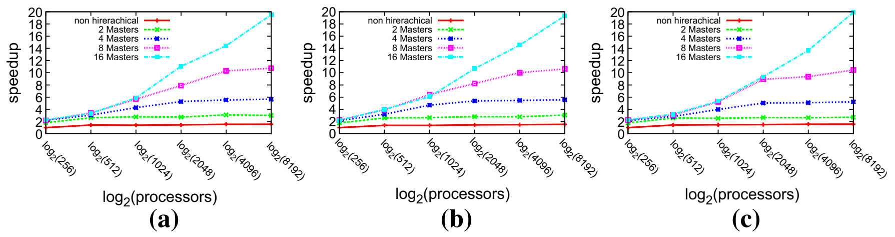

Fig. 7 The speedup of adjoint convolution using hierarchical distributed schemes. **a** HDTSS, **b** HDFSS, **c** HDGSS

6.2 Cloud System

The following loop scheduling schemes are implemented. distributed schemes: DTSS, DFSS, DGSS; hierarchical distributed schemes: HDTSS, HDFSS, HDGSS. All the schemes are implemented by C++ and MPI. All timings are in seconds.

Two applications, quick sort and matrix multiplication are used to evaluate the overall performance. The quick sort problem consists of sorting $N = 20,000$ rows of random arrays (each array is of size $N = 20K$). For the matrix multiplication problem the dimension of the matrices are $N = 15,000$. The computations serial complexity is $O(NlogN)$ (on average)and $O(N^3)$ respectively.

We next analyze the total execution time in terms of the master time, the communication time and overhead time. Let T_{master} denote the total execution time of a master, which means that the workers managed by this master have finished all the work assigned to them and the results have been returned to the master. We note that, $T_{exec} = max\{T_{master_1}, T_{master_2}, \ldots, T_{master_m}\} + T^{'}$, where T_{exec} denotes the total execution time (measured by the supermaster) for m masters hierarchical distributed scheme and where $T^{'}$ is the time for scheduling, work distribution, start up and termination overheads in the supermaster. We note that we measure directly T_{exec} and T_{master_i} for $i = 1, \ldots, m$. Thus, T_{master} represents most of the work execution time in the experiment, because the scheduling overhead in the supermaster is low. Thus, the load balancing depends on both computation in workers and the communication time to return the results to the masters. We use the maximum master times difference, $T_{diff} = max\{T_{master_1}, T_{master_2}, \ldots, T_{master_m}\} - min\{T_{master_1}, T_{master_2}, \ldots, T_{master_m}\}$, to measure the work load balancing in the experiment. If T_{diff} is small, the major work is distributed evenly and the uti-

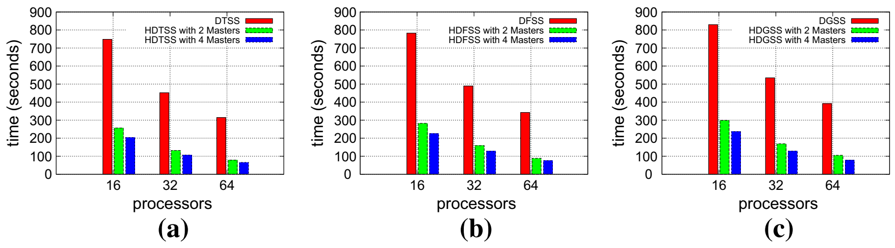

Fig. 8 The maximum difference in masters execution time for quick sort using non-hierarchical distributed and hierarchical distributed schemes. **a** DTSS and HDTSS, **b** DFSS and HDFSS, **c** DGSS and HDGSS

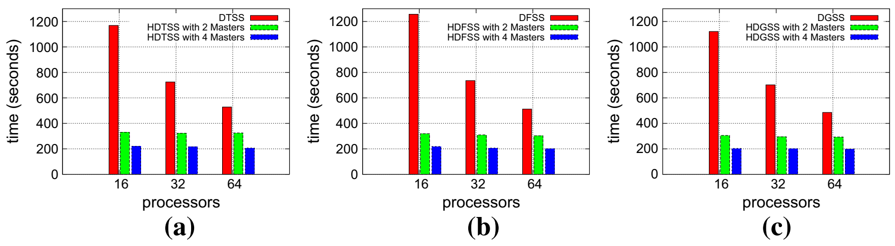

Fig. 9 The maximum difference in masters execution time for matrix multiplication using non-hierarchical distributed and hierarchical distributed schemes. **a** DTSS and HDTSS, **b** DFSS and HDFSS, **c** DGSS and HDGSS

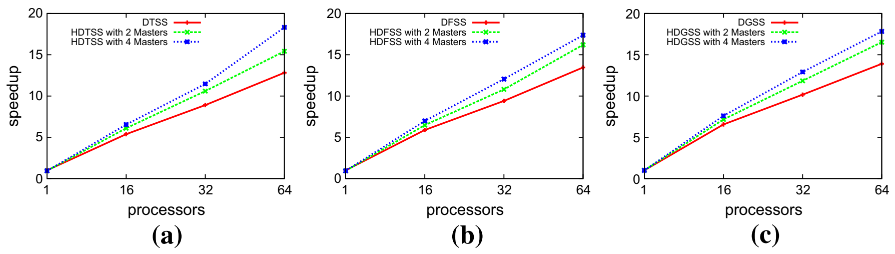

Fig. 10 The speedup for quick sort using non-hierarchical distributed and hierarchical distributed schemes. **a** DTSS and HDTSS, **b** DFSS and HDFSS, **c** DGSS and HDGSS

lization is better. Figures 8 and 9 present the T_{diff} for non-hierarchical distributed schemes (DTSS, DFSS, DGSS) and hierarchical distributed schemes(HDTSS, HDFSS, HDGSS). For non-hierarchical distributed schemes, T_{diff} is the same as T_{master}. It can be observed that the differences in the case of non-hierarchical distributed schemes are quite substantial. The work is centralized using the single master and the communication and synchronization overhead is high. On the other hand, in the case of hierarchical distributed schemes, the results collection is distributed among several masters. Thus T_{diff} is small and the work load is more balanced.

Figures 10 and 11 present the speedup comparison between non-hierarchical distributed schemes schemes and hierarchical distributed schemes with 1, 16, 32 and 64 workers. The speedup is computed by $S_p = \frac{\hat{T_1}}{T_p}$, $\hat{T_1}$ is the execution time for non-hierarchical distributed DTSS with 1 worker. It can be observed that all the speedups of

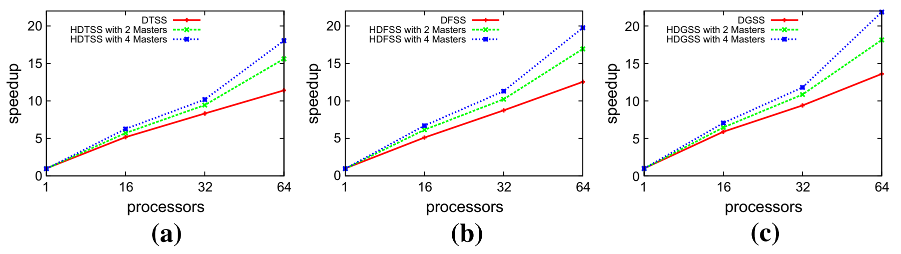

Fig. 11 The speedup for matrix multiplication using non-hierarchical distributed and hierarchical distributed schemes. **a** DTSS and HDTSS, **b** DFSS and HDFSS, **c** DGSS and HDGSS

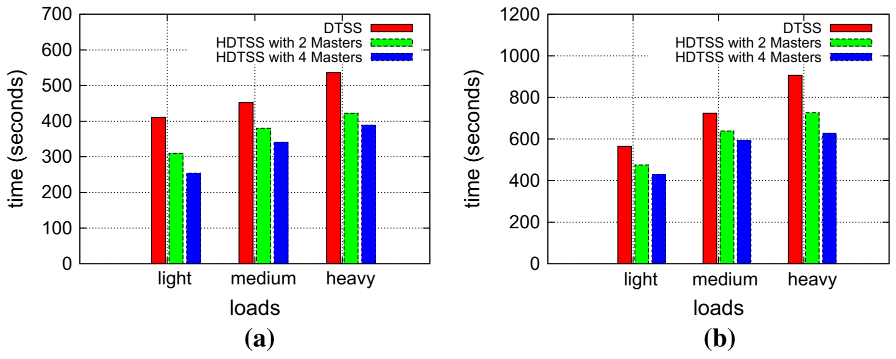

Fig. 12 The performance of hierarchical versus distributed schemes under varying loads. **a** QuickSort, **b** MatrixMultiply

hierarchical distributed schemes are better than non-hierarchical distributed schemes' with 1, 16, 32 and 64 workers for both applications. Also, as the number of workers increases, the speedup of hierarchical distributed schemes improves which shows that the schemes are scalable.

Figure 12 shows the results of testing the distributed versus the hiearchical schemes under varying loads of the VMs on the cloud platform. We chose the DTSS and HDTSS scheduling scheme for this test. We ran the Matrix multiply and the quick sort problems on 32 VMs. We tried three load factors representing (light, medium and heavy) loads. This was implemented using the Stress software [32]. We observe that the time measurements are significantly lower for HDTSS with 4 masters compared to DTSS in both problems. We expect this difference to increase in favor of all the hierachical schemes (vs the non-hierarchical) for larger number of VMs for all problems.

7 Conclusion and Future Work

In this paper, we proposed a hierarchical distributed model for self-scheduling schemes. We implemented the new schemes on a homogeneous large-scale cluster and on a heterogeneous cloud environment. Our experiments validate the scalability and the better overall performance of the hierarchical schemes. MapReduce is a programming model which offers an alternative to MPI implementation of many data parallel applications. In the future, we plan to implement our schemes in MapReduce and compare to MPI for scientific loops. In the future, we plan to test our methodologies using large scale benchmarks and also using loops with dependencies.

Acknowledgements The authors acknowledge: (1) support by NSF grant (HRD-0932339) to the University of Texas at San Antonio; and (2) time grants to access the facilities of Institute for Cyber Security(ICS) of University of Texas at San Antonio and FutureGrid at Indiana University, Bloomington; (3) the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC, visualization, database, or grid resources that have contributed to the research results reported within this paper. URL:http://www.tacc.utexas.edu.

References

1. Keahey, K: Cloud computing for science. In: Proceeding of 21st Scientific and Statistical Database Management Conference. vol. 5566, pp. 478–478 (2009)
2. Wang, L., Tao, J., Kunze, M., Castellanos, A., Kramer, D., Karl, W.: Scientific cloud computing: early definition and experience. In: 10th IEEE International Conference on High Performance Computing and Communications, HPCC'08, pp. 825–830 (2008)
3. Yeo, S., Lee, H.-H.: Using mathematical modeling in provisioning a heterogeneous cloud computing environment. Computer **44**, 55–62 (2011)
4. Han, Y., Chronopoulos, A.: Distributed loop scheduling schemes for cloud systems. In: IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), pp. 955–962 (2013)
5. Sonnek, J., Greensky, J., Reutiman, R., Chandra, A.: Starling: minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration. In: 39th International Conference on Parallel Processing, ICPP'10, pp. 228–237 (2010)
6. Han, Y., Chronopoulos, A.: A hierarchical distributed loop self-scheduling scheme for cloud systems. In: 12th IEEE International Symposium on Network Computing and Applications (NCA), pp. 7–10 (2013)
7. Han, Y., Chronopoulos, A.: Scalable loop self-scheduling schemes implemented on large-scale clusters. In: IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), pp. 1735–1742 (2013)
8. Joyent corporation. http://joyent.com/
9. Kejariwal, A., Nicolau, A., Polychronopoulos, C.: History-aware self-scheduling. In: International Conference on Parallel Processing Parallel Processing, ICPP'06, pp. 185–192 (2006)
10. Chronopoulos, A.T., Penmatsa, S., Xu, J., Ali, S.: Distributed loop-scheduling schemes for heterogeneous computer systems. Concurr. Comput. Pract. Exp. **18**, 771–785 (2006)
11. Banicescu, I., Velusamy, V., Devaprasad, J.: On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. Clust. Comput. **6**, 215–226 (2003)
12. Chronopoulos, A.T., Penmatsa, S., Yu, N., Yu, D.: Scalable loop self-scheduling schemes for heterogeneous clusters. Int. J. Comput. Sci. Eng. **1**, 110–117 (2005)
13. Li, P., Ji, Q., Zhang, Y., Zhu, Q.: An adaptive chunk self-scheduling scheme on service grid. In: IEEE Asia-Pacific Services Computing Conference, APSCC'08, pp. 39–44 (2008)
14. Yang, K.-W.C., Chao-Tung, Li, K.-C.: An efficient parallel loop self-scheduling on grid environments. In: Network and Parallel Computing, pp. 92–100 (2004)
15. Yang, C.-T., Chang, S.-C.: A parallel loop self-scheduling on extremely heterogeneous pc clusters. In: Proceedings of International Conference on Computational, Science, pp. 1079–1088 (2003)
16. Cheng, K.-W., Yang, C.-T., Lai, C.-L., Chang, S.-C.: A parallel loop self-scheduling on grid computing environments. In: Proceedings of the 2004 International Symposium on Parallel Architectures, Algorithms and IEEE Networks, pp. 409–414 (2004)
17. Andronikos, T., Ciorba, F.M., Riakiotakis, I., Papakonstantinou, G., Chronopoulos, A.T.: Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems. Perform. Eval. **67**(12), 1324–1339 (2010)

18. Ciorba, F., Andronikos, T., Riakiotakis, I., Chronopoulos, A., Papakonstantinou, G.: Dynamic multi phase scheduling for heterogeneous clusters. In: Proceedings of the 20th IEEE International Parallel Distributed Processing Symposium, IPDPS06, p. 10 (2006)
19. Riakiotakis, I., Ciorba, F.M., Andronikos, T., Papakonstantinou, G., Chronopoulos, A.T.: Towards the optimal synchronization granularity for dynamic scheduling of pipelined computations on heterogeneous computing systems. Concurr. Comput. Pract. Exp. **24**, 2302–2327 (2012)
20. Diaz, J., Reyes, S., Nino, A., Munoz-Caro, C.: Derivation of self-scheduling algorithms for heterogeneous distributed computer systems: application to internet-based grids of computers. Future Gener. Comput. Syst. **25**, 617–626 (2009)
21. Wu, C.-C., Yang, C.-T., Lai, K.-C., Chiu, P.-H.: Designing parallel loop self-scheduling schemes using the hybrid MPI and openMP programming model for multi-core grid systems. J. Supercomput. **59**, 42–60 (2012)
22. He, Y., Liu, J., Sun, H.: Scheduling functionally heterogeneous systems with utilization balancing. In: IEEE International Parallel Distributed Processing Symposium, IPDPS'11, pp. 1187–1198 (2011)
23. Penmatsa, S., Chronopoulos, A., Karonis, N., Toonen, B.: Implementation of distributed loop scheduling schemes on the teragrid. IN: IEEE International Parallel and Distributed Processing Symposium, IPDPS'07, pp. 1–8 (2007)
24. Kyriakopoulos, K., Chronopoulos, A., Ni, L.: An optimal scheduling scheme for tiling in distributed systems. In: IEEE International Conference on Cluster Computing, pp. 267–274 (2007)
25. Chronopoulos, A., Ni, L., Penmatsa, S.: Multi-dimensional dynamic loop scheduling algorithms. In: IEEE International Conference on Cluster Computing, pp. 241–248 (2007)
26. Hwang, K., Dongarra, J., Fox, G.C.: Distributed and cloud computing: from parallel processing to the internet of things. Morgan Kaufmann, (2011)
27. Wen, G., Hong, J., Xu, C., Balaji, P., Feng, S., Jiang, P.: Energy-aware hierarchical scheduling of applications in large scale data centers. In: International Conference on Cloud and Service Computing, CSC'11, pp. 158–165 (2011)
28. Iosup, A., Ostermann, S., Yigitbasi, M., Prodan, R., Fahringer, T., Epema, D.: Performance analysis of cloud computing services for many-tasks scientific computing. IEEE Trans. Parallel Distrib. Syst. **22**, 931–945 (2011)
29. Ekanayake, J. Fox, G.: High performance parallel computing with clouds and cloud technologies. In: Proceedings of the First International Conference on Cloud Computing, pp. 20–38 (2010)
30. Evangelinos, C., Hill, C.N.: Cloud computing for parallel scientific HPC applications: feasibility of running coupled atmosphere–ocean climate models on Amazon's EC2. In: 5th International Conference on Computability and Complexity in Analysis, pp. 159–168 (2008)
31. Shih, W.-C. Tseng, S.-S., Yang, C.-T.: Performance study of parallel programming on cloud computing environments using MapReduce. In: International Conference on Information Science and Applications, ICISA'10, pp. 1–8 (2010)
32. Stress. http://www.hecticgeek.com/2012/11/stress-test-your-ubuntu-computer-with-stress/