Scalable Loop Self-Scheduling Schemes Implemented on Large-Scale Clusters

Yiming Han and Anthony T. Chronopoulos

Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, USA
Email: {yhan, atc}@cs.utsa.edu

***Abstract*—Loops are the largest source of parallelism in many scientific applications. Parallelization of irregular loop applications is a challenging problem to achieve scalable performance on large-scale multi-core clusters. Previous research proposed an effective Master-Worker model on clusters for distributed self-scheduling schemes that apply to parallel loops with independent iterations. However, this model has not been applied to large-scale clusters. In this paper, we present an extension of the distributed self-scheduling schemes implemented in a hierarchical Master-Worker model. Our experiments with different self-scheduling schemes demonstrate good scalability when scaling upto 8,192 processors.**

***Index Terms*—Scalable, Master-Worker, Self-Scheduling, Hierarchical.**

I. Introduction

Loops are the largest source of parallelism in many scientific applications. There are several loop scheduling schemes for loops with and without data dependencies on clusters. If the iterations of a loop have no dependencies, each iteration can be considered as a task and can be scheduled independently. Loops can be scheduled statically at compile-time. This type of scheduling has the advantage of minimizing the scheduling time overhead, but it may cause load imbalancing when the loop style is not uniformly distributed. Dynamic scheduling adapts the assigned number of iterations whenever it is unknown in advance how large the loop tasks are. A self-scheduling algorithm is a dynamic algorithm for scheduling loop iterations. An important class of dynamic scheduling are the self-scheduling schemes [1], [2], [3], [4] and references therein. In UMA (Uniform Memory Access) parallel system, these schemes can be implemented using a critical section for the loop iterations and no need exists for dedicating a (master) processor to do the scheduling. This is why these schemes are called self-scheduling schemes. An affinity scheduling algorithm is proposed and studied to reduce communications overhead (nonlocal memory accesses) on shared-memory multiprocessors in [5], [6] and [7]. A feedback guided dynamic loop scheduling is introduced and studied in [8] and [9]. An adaptive weighted factoring which performs well for scheduling loops in parallel unstructured grid applications and N-Body simulations is proposed in [10]. Different self-scheduling algorithms have been proposed which dynamically assign chunks of variable sizes to processors. They differ from each other in the way they calculate the size of the chunk assigned to each processor. The self-scheduling algorithms were initially proposed for loops without dependencies for shared memory parallel systems and later extended to distributed computing systems [2]. There are also results focusing on loops with dependencies [11]. Recent research results have been reported on loop self-scheduling methods for multi-core, graphics processing units, grids, and cloud systems [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26] and [27].

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. The developers of high performance computing application programs may over-schedule resources which can cause load imbalance and low speedup. This is especially true for some nested loops when executed on large-scale clusters. Previous research, [17], [18] 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 [13]. In this paper, we design a scalable hierarchical distributed Master-Worker model for self-scheduling schemes on large-scale clusters. We implement these schemes on a large-scale cluster of Texas Advanced Computing Center, University of Texas at Austin. Our experiments demonstrate the good scalability of the proposed schemes.

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

II. Loop Scheduling Schemes

Self-scheduling is an automatic loop scheduling method in which idle processors request new loop iterations to be assigned to them. We study these methods from the perspective

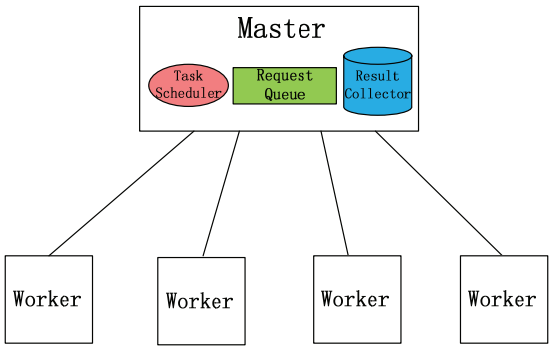


Fig. 1. Self-Scheduling schemes: the Master-Worker model

of distributed systems. For this, we use the Master-Worker architecture model (Figure 1). Idle processors submit a request to the master for new loop iterations. The master node has three components: 1) Task Scheduler. It uses scheduling schemes to divide the whole work into small scheduled chunks; 2) Request Queue. If one or some workers are idle, they request more work from the master node. If the master node is busy serving another worker, the requesting workers are added into Request Queue and wait to be served; 3) Result Collector. When the workers finish their work they send a new request and send the computed results to the Result Collector.

The number of iterations a processor should be assigned is an important issue. Due to processors' possible heterogeneity and communication overhead, assigning the wrong processor a large number of iterations at the wrong time, may cause load imbalancing. Also, assigning a small number of iterations may cause too much communication and scheduling overhead.

A. Notations:

The following are common notations used throughout the whole paper:

- 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: $i = 1, 2, ...$);
- N is the number of scheduling steps;
- t_j, $j = 1, .., p$, is the execution time of P_j to complete all its tasks assigned to it by the scheduling scheme;
- $T_p = \max_{j=1,..,p}(t_j)$, 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:

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

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. Imbalance depends on the execution time gap between t_j, for $j = 1, \ldots, p$. This gap may be large if the first chunk is too large or (more often) if the last chunk (called the *critical chunk*) is too small.

The different ways to compute C_i has given rise to different scheduling schemes. Some widely used schemes are the following. These schemes are studied or extended in [1], [2], [3], [4], [5], [10] and references therein.

Trapezoid Self-Scheduling (TSS) $C_i = C_{i-1} - D$, with (chunk) decrement : $D = \left\lfloor \frac{(F-L)}{(N-1)} \right\rfloor$, where: the first and last chunk-sizes (F,L) are user/compiler-input or $F = \left\lfloor \frac{I}{2p} \right\rfloor$, $L = 1$. The number of scheduling steps assigned: $N = \left\lceil \frac{2*I}{(F+L)} \right\rceil$. Note that $C_N = F - (N-1)D$ and $C_N \geq 1$ due to integer divisions.

Factoring Self-Scheduling (FSS) $C_i = \lceil R_{i-1}/(\alpha p) \rceil$, where the parameter α is computed (by a probability distribution) or is suboptimally chosen $\alpha = 2$. The chunk-size is kept the same in each *stage* or *round* (in which all processors are assigned a chunk of the same size) before moving to the next stage. Thus $R_i = R_{i-1} - pC_i$ (where $R_0 = I$) after each stage.

Guided Self-Scheduling (GSS) $C_i = \lceil R_{i-1}/p \rceil$. In the last steps too many small chunks are assigned. It assigns large chunks initially, which implies reduced communication/scheduling overheads only in the beginning but small chunks later. A modified version $GSS(k)$ with minimum assigned chunk-size k (chosen by the user) attempts to improve on the weaknesses of GSS.

III. Distributed Loop 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 next present the distributed loop scheduling schemes based on the Master-Worker architecture.

A. Terminology:

- $V_j = Speed(P_j)/min_{1 \leq i \leq p}\{Speed(P_i)\}$, $j = 1, ..., p$, 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.

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: $F = \lfloor \frac{I}{2V} \rfloor$, $L = 1, N = \left\lceil \frac{2*I}{(F+L)} \right\rceil$,
$D = \left\lfloor \frac{(F-L)}{(N-1)} \right\rfloor, Current = F$

Algorithm 1 Calculate DC

```
DC = 0;
for k = 1 → 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 = ⌈R/(αV)⌉ * 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 = ⌈R/(A)⌉ * 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;

IV. Hierarchical distributed schemes

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

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 overhead 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 such as a cluster, where long communication latency can be encountered. Another kind of overhead occurs when 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 2 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) 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

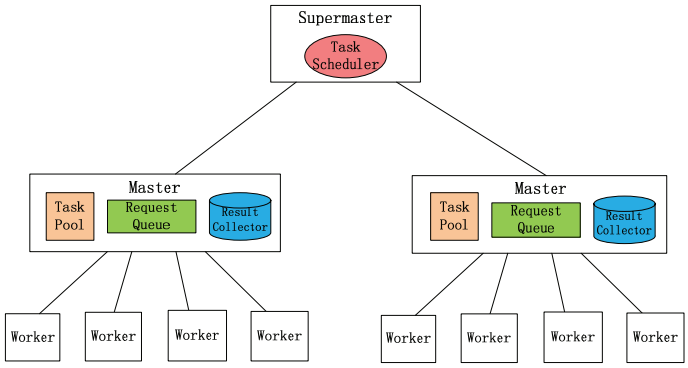


Fig. 2. Hierarchical Architecture

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:

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) to compute small scheduled chunks for each Worker under the requesting Master. Then Master may store the chunks into its Task Pool.

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)** Request work to Super Master to fill Task Pool;
- **(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;

V. Implementation and Experiments

A. Applications

- Mandelbrot Set [28]

  The Mandelbrot Set 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.

  MSetLSM(MSet,nx,ny,xmin,xmax,ymin,ymax,maxiter)

```
BEGIN
 FOR iy = 0 TO ny-1 DO
  cy = ymin+iy*(ymax - ymin)/(ny - 1)
  FOR ix = 0 TO nx-1 DO
   cx = xmin+ix*(xmax - xmin)/(nx - 1)
   MSet[ix][iy]=MSetLevel(cx,cy,maxiter)
  END FOR
 END FOR
END
```

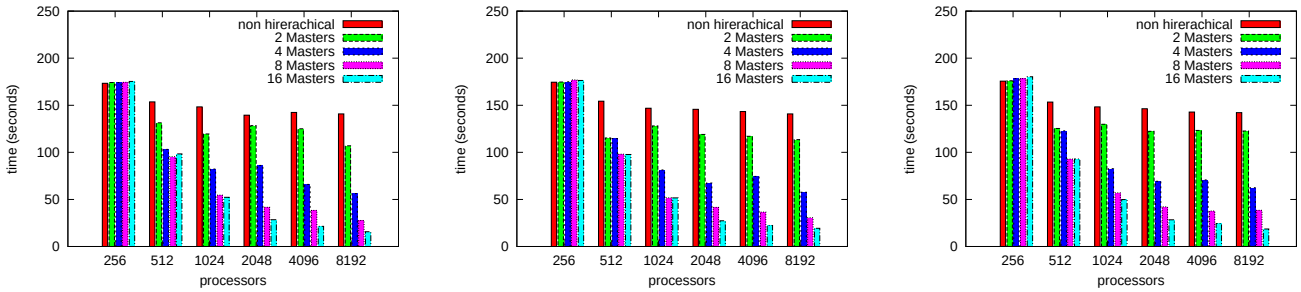
  MSetLevel(cx,cy,maxiter)

```
BEGIN
 x = y = x2 = y2 = 0.0, iter = 0
 WHILE(iter<maxiter)AND(x2+y2<2.0)DO
   temp = x2 - y2 + cx
   y = 2*x*y + cy
   x = temp
   x2 = x*x
   y2 = y*y
   iter = iter + 1
 END WHILE
 RETURN(iter)
END
```

- Adjoint Convolution

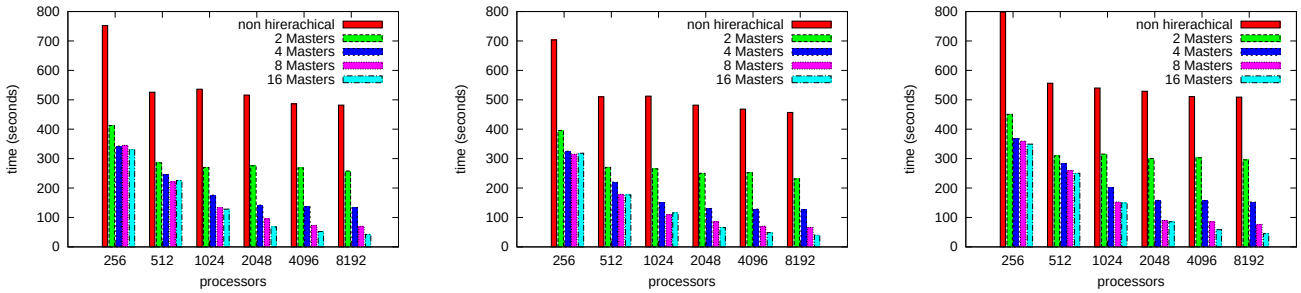
  This application involves computation of decreasing task sizes. Thus, it can cause load imbalance in the parallel computation. The i_{th} iteration's time is $O(N^2 - i)$.

```
BEGIN
 FOR I = 1 TO N * N DO
  FOR J = I TO N * N DO
   A(I) = A(I) + X * B(J) * C(J - I)
  END FOR
```



a. $HDTSS$ b. $HDFSS$ c. $HDGSS$

Fig. 3. The performance of Mandelbrot Set using hierarchical distributed schemes



a. $HDTSS$ b. $HDFSS$ c. $HDGSS$

Fig. 4. The performance of Adjoint Convolution using hierarchical distributed schemes

```
  END FOR
END
```

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.

B. Platform

We use the Ranger cluster system as our platform. The Ranger cluster system is located at TACC (Texas Advanced Computing Center) in University of Texas at Austin. The 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.

VI. Results

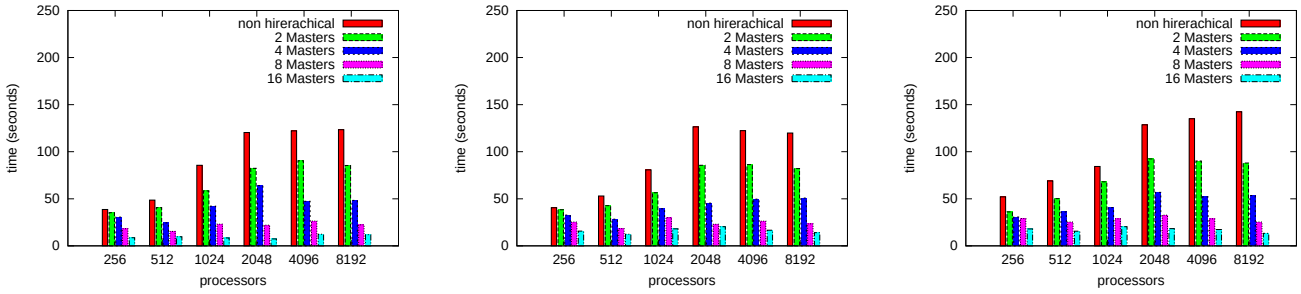
In this section, we compare the performance of the various schemes, non-hierarchical (single master) and hierarchical (2 masters, 4 masters, 8 masters, 16 masters) and with a number of workers (processors) from 256 to 8,192. The Mandelbrot Set computation domain is [-2.0, 2.0] $\times$ [-2.0, 2.0] and its size is 200K $\times$ 200K. The Adjoint Convolution has a size of 800 $\times$ 800 and the arrays are generated randomly.

In order to avoid too many small chunks at the end of scheduling which may introduce unnecessary synchronization overhead, we add a threshold to terminate if the chunk size drops below it. In our experiment, the threshold equals 5, which means the master can not assign a chunk with size less than 5, except possibly the last chunk.

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

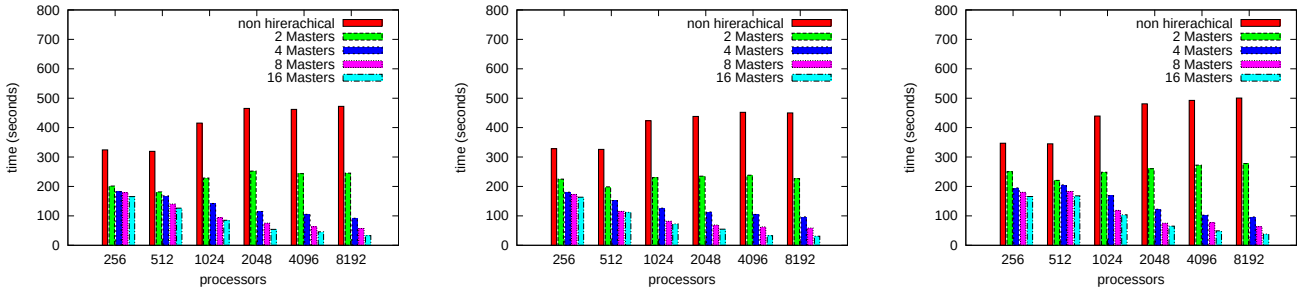
The performance presented in Figure 3 and Figure 4 is organized from left to right in doubling numbers of workers using HDTSS, HDFSS, HDGSS schemes for Mandelbrot Set and Adjoint Convolution. It can be observed that the hierarchical distributed scheme with more master nodes can achieve better performance 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. 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 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 Figure 5 and Figure 6, we show the non-overlapped



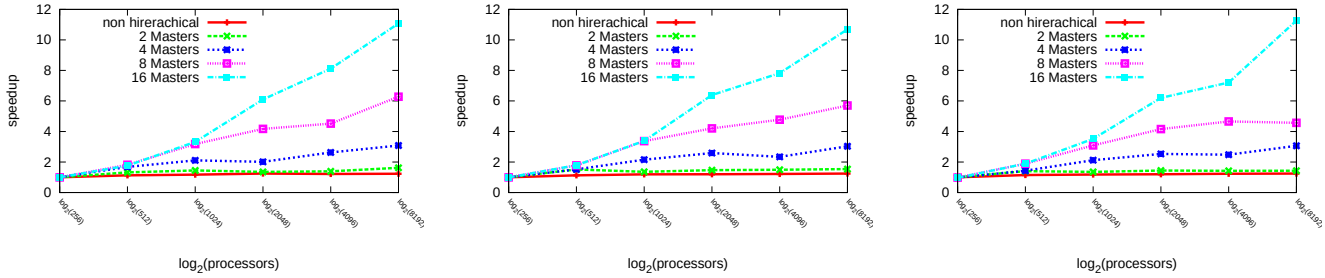
a. $HDTSS$ b. $HDFSS$ c. $HDGSS$

Fig. 5. The non-overlapped communication and synchronization overhead $T'_{overhead}$ of Mandelbrot Set



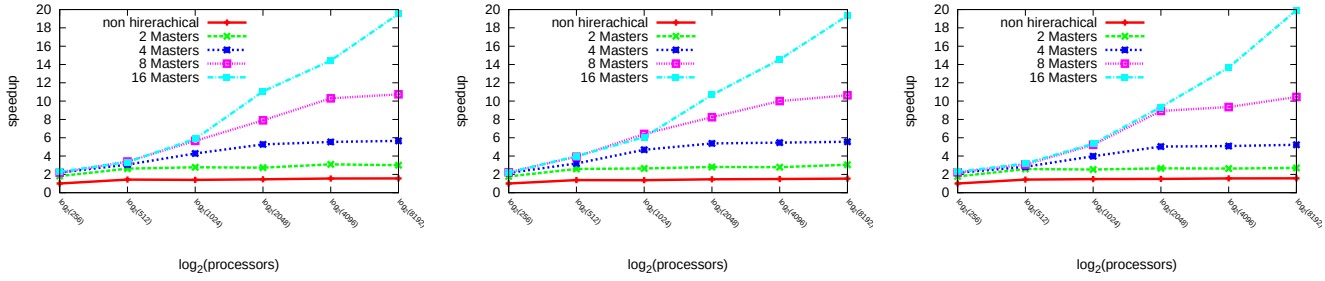
a. $HDTSS$ b. $HDFSS$ c. $HDGSS$

Fig. 6. The non-overlapped communication and synchronization overhead $T'_{overhead}$ of Adjoint Convolution



a. $HDTSS$ b. $HDFSS$ c. $HDGSS$

Fig. 7. The speedup of Mandelbrot Set using hierarchical distributed schemes



a. $HDTSS$ b. $HDFSS$ c. $HDGSS$

Fig. 8. The speedup of Adjoint Convolution using hierarchical distributed schemes

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 most slowest part for large problems in distributed memory systems.

Figure 7 and Figure 8 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.

VII. Conclusion and future work

In this paper, we studied and implemented (in MPI) hierarchical distributed loop scheduling schemes. We showed that non hierarchical loop scheduling algorithms with Master-Worker model does not scale well when there are hundreds of workers in the system. We proposed and implemented a hierarchical distributed model for self-scheduling schemes on large-scale clusters that maintains the load balancing properties of some well known loop scheduling schemes, and also shows better scalability on large-scale clusters. There are past results on thread based and work-stealing based implementation of loop parallelization [29] [30]. In the future, we plan to study these approaches in our method.

Acknowledgement

We gratefully acknowledge the following: (i) the reviewers for their helpful and constructive suggestions, which considerably improved the quality of the manuscript; (ii) support by NSF grant (HRD-0932339) to the University of Texas at San Antonio; and (iii) time grants to access the facilities of Texas Advanced Computing Center (TACC) at University of Texas at Austin and FutureGrid at Indiana University, Bloomington.

References

[1] Y. W. Fann, C. T. Yang, S. S. Tseng, and C. J. Tsai, "An intelligent parallel loop scheduling for parallelizing compilers," *Journal of Information Science and Engineering*, pp. 169–200, 2000.

[2] A. T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali, "Distributed loop-scheduling schemes for heterogeneous computer systems," *Concurrency and Computation:Practice and Experience*, vol. 18, pp. 771–785, 2006.

[3] A. R. Hurson, J. T. Lim, K. M. Kavi, and B. Lee, "Parallelization of doall and doacross loops - a survey." *Advances in Computers*, pp. 53–103, 1997.

[4] A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos, "An efficient approach for self-scheduling parallel loops on multiprogrammed parallel computers," *Proceedings of the 18th international conference on Languages and Compilers for Parallel Computing*, pp. 441–449, 2006.

[5] E. P. Markatos and T. J. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 4, pp. 379–400, 1994.

[6] 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.

[7] W. Shi, W. Hu, and Z. Tang, "Affinity-based self scheduling: A more practical load balancing scheme for home-based software dsms," 1999.

[8] J. M. Bull, "Feedback guided dynamic loop scheduling: Algorithms and experiments," *Proc. of 4th Intl Euro - Par Conference*, pp. 377–382, 1998.

[9] T. Tabirca, S. Tabirca, and L. Tianruo Yang, "An O(logp) algorithm for the discrete feedback guided dynamic loop," *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pp. 321–326, 2006.

[10] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring," *Cluster Computing*, pp. 215–226, 2003.

[11] F. M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou, and A. T. Chronopoulos, "Enhancing self-scheduling algorithms via synchronization and weighting," *Journal of Parallel and Distributed Computing*, vol. 68, pp. 246–264, 2008.

[12] J. Herrera, E. Huedo, R. Montero, and I. Llorente, "Loosely-coupled loop scheduling in computational grids," *20th International Parallel and Distributed Processing Symposium(IPDPS)*, pp. 338–338, 2006.

[13] 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(2007)*, pp. 1–8, 2007.

[14] S. Fujita, "Semi-dynamic multiprocessor scheduling with an asymptotically optimal performance ratio," *Institute of Electronics, Information and Communication Engineers(IEICE)*, pp. 1764–1770.

[15] J. Díaz, S. Reyes, A. Niño, and C. Muñoz Caro, "Derivation of self-scheduling algorithms for heterogeneous distributed computer systems: Application to internet-based grids of computers," *Future Generation Computer Systems*, pp. 617–626, 2009.

[16] W.-C. Shih, S.-S. Tseng, and C.-T. Yang, "Performance study of parallel programming on cloud computing environments using MapReduce," *2010 International Conference on Information Science and Applications (ICISA)*, pp. 1–8, 2010.

[17] C.-T. Yang, C.-C. Wu, and J.-H. Chang, "Performance-based parallel loop self-scheduling using hybrid OpenMP and MPI programming on multicore SMP clusters," *Concurrency and Computation: Practice and Experience*, pp. 721–744, 2011.

[18] 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.

[19] P. Li, Q. Zhu, Q. Ji, and X. Zhu, "An approach of chunk-based task runtime prediction for self-scheduling on multi-core desk grid." *Journal of Computers*, pp. 1339–1345, 2011.

[20] Y. C-T, C. K-W, and L. K-C, "An efficient load balancing scheme for grid-based high performance scientific computing," *Proceeding of the 19th International Conference on Advanced Information Networking and Applications(AINA)*, 2005.

[21] K.-W. Cheng, C.-T. Yang, C.-L. Lai, and S.-C. Chang, "A parallel loop self-scheduling on grid computing environments," *7th International Symposium on Parallel Architectures, Algorithms and Networks*, pp. 409–414, 2004.

[22] P. Li, Q. Ji, Y. Zhang, and Q. Zhu, "An adaptive chunk self-scheduling scheme on service grid," *IEEE Asia-Pacific Services Computing Conference(APSCC)*, pp. 39–44, 2008.

[23] M. Kandemir, T. Yemliha, S. W. Son, and O. Ozturk, "Memory bank aware dynamic loop scheduling," *Proceedings of the conference on Design, automation and test in Europe*, pp. 1671–1676, 2007.

[24] M. Athanasaki, E. Koukis, and N. Koziris, "Scheduling of tiled nested loops onto a cluster with a fixed number of SMP nodes." *IEEE Parallel, Distributed and Network-Based Processing*, pp. 424–433, 2004.

[25] C. S. and X. J., "Partitioning and scheduling loops on NOWs," *Computer Communications*, vol. 22, 1999.

[26] A. Kejariwal, A. Nicolau, and C. Polychronopoulos, "History-aware self-scheduling," *International Conference on Parallel Processing(ICPP)*, pp. 185–192, 2006.

[27] H. Huo, C. Sheng, X. Hu, and B. Wu, "An energy efficient task scheduling scheme for heterogeneous GPU-enhanced clusters," *2012 International Conference on Systems and Informatics(ICSAI)*, pp. 623–627, 2012.

[28] B. B. Mandelbrot, *Fractal Geometry of Nature*. W.H. Freeman & Co, 1988.

[29] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," *IEEE International Symposium on Parallel and Distributed Processing(IPDPS)*, pp. 1–8, 2008.

[30] S. Olivier and J. Prins, "Scalable dynamic load balancing using UPC," *37th International Conference on Parallel Processing(ICPP)*, pp. 123–131, 2008.