2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies

A Hierarchical Work-Stealing Framework for Multi-core Clusters

Yizhuo Wang, Weixing Ji, Qi Zuo, Feng Shi
School of Computer Science and Technology
Beijing Institute of Technology
Beijing, China
E-mail: {frankwyz, jwx, zqll27, sfbit }@bit.edu.cn

***Abstract*—Work-stealing has been widely used in task-based parallel programing for dynamic load balancing. The overhead of work-stealing on distributed memory systems is much higher than that on shared memory systems. To minimize the overhead of work-stealing on a multi-core cluster, we propose a hierarchical work-stealing framework, in which work-stealing is performed inside a node before across the node boundary. Two key techniques used in our framework to reduce the inter-node steals are: a) adaptive initial partitioning for different task parallel patterns; b) centralized control for inter-node work-stealing, which improves the efficiency of victim selection and termination detection. We compare our technique to the classical work-stealing scheme and a state-of-the-art work-stealing scheme [1] for multi-core clusters. Our technique outperforms them by 19% and 8% respectively.**

Keywords-work-stealing; multi-core cluster; task scheduling

I. INTRODUCTION

The prevalence of multi-core architecture has brought clusters into a multi-core era. To run an application efficiently on a multi-core cluster, the key issue is how to exploit parallelism in an application and schedule parallel portions of the application on multiple processing elements (PEs) to achieve optimal load balancing.

Task parallelism is a popular form of parallelization of programs. Most parallel programing languages and models are constructed based on task parallelism, such as Intel TBB[2], IBM X10[3], Microsoft TPL[4], OpenMP 3.0[5] and Cilk[6]. In these task-based parallel programming models, work-stealing has been widely used for dynamic load balancing. Under work stealing, each processor maintains its own work queue of tasks which are double-ended. Tasks are enqueued to or dequeued from a task queue at the queue's bottom end at runtime. Tasks in all queues are independent of each other and thus can be executed in parallel. When a processor's task queue is empty, it attempts to steal work from the top end of a victim processor's task queue.

Work-stealing is primarily used on shared memory systems, in which all the worker threads have the same priority and victim is randomly selected. Random victim selection has been proven to be optimal for shared memory systems [6]. However it is inefficient when applied to distributed memory systems because the communication cost between PEs is not negligible and normally is not uniform in distributed system. For this reason hierarchical work-stealing approaches were devised. In [7], [8] and [1], the underlying architecture is viewed as a hierarchical structure. A thief tries to steal tasks from PEs on the same level before sending the steal request to the upper level. However, in these approaches, work-stealing is still a peer to peer process between a thief and a victim. When performing a steal operation, the thief must probe the PEs for work. The random probes would result in performance degradation when work is sparse.

In this paper, we propose a hierarchical work-stealing framework which perceives two levels of hierarchy: cluster nodes and multiple cores inside a node. It is similar to the technique in [1], but we implement a global scheduler for inter-node task scheduling. Victim is directly determined by the global scheduler, thus the overhead of probing for work is eliminated. In distributed memory systems, the cost of task migration is not low. In order to reduce task migrations across cluster nodes, we adopt an adaptive initial partitioning phase in our system which balances the workload among cluster nodes before the parallel execution. To the best of our knowledge, our framework is the first system to incorporate initial partitioning into a work-stealing framework.

The rest of the paper is organized as follows. In section 2, we describe our framework in detail. Experimental results are presented in section 3. Related work is discussed in Section 4. We conclude the paper with Section 5.

II. ARCHITECTURE

A. System Overview

This paper addresses the efficient scheduling of a task parallel application on a multi-core cluster. A multi-core cluster has a two-level architecture. One is the distributed memory level which consists of the cluster nodes. Another is the shared memory level which consists of multi-cores inside a node. To exploit both inter-node parallelism and intra-node parallelism on the two levels, we propose a hierarchical work-stealing framework in which tasks are scheduled with different work-stealing schemes on these two levels to achieve dynamic load balancing.

The system model is shown in Fig. 1. Assuming all the program and data files have been deployed on each node. The user logs on to a node to start the application. Then this node is viewed as a master node. The master node could be any node in the cluster or a specific node, such as a resource

This work was supported in part by the National Natural Science Foundation of China under grant NSFC-60973010.

978-0-7695-4879-1/12 $26.00 © 2012 IEEE
DOI 10.1109/PDCAT.2012.17

manager node[1]. A global scheduler (**GS**) works on the master node, which is responsible for inter-node task scheduling, including the initial partitioning and task migration (work-stealing) across the nodes. The novel techniques of initial partitioning and inter-node work-stealing distinguish our system from the existing work-stealing systems [1][7][8].

Traditional work-stealing schemes are completely dynamic scheduling schemes, in which there is no initial partitioning phase. One initial task runs on a worker thread, and new tasks are spawned continually and stolen by idle threads during the execution. In shared memory systems, a worker thread can access the task queues of other worker threads directly. Therefore, a spawned task can migrate to an idle thread very quickly. The absence of initial partitioning just arouse a few times of additional task migrations. The cost of these migrations is negligible in shared memory systems. In distributed memory systems, however, task migration has much higher overhead which consists of the communication costs and the cost of task serialization that is packaging a task into a message. This overhead is no longer negligible in distributed memory systems. Initial partitioning could balance the load statically before the parallel execution of the tasks and thus reduce the frequency of dynamic task stealing across the nodes. An ideal partitioning could even make the inter-node task migration never happen. In our framework, the global scheduler partitions the parallel portions of the application adaptively in an initial partitioning phase, according to the type of task parallelism. The detail of our initial partitioning is described in the next subsection.

In Fig. 1, every node, including the master node, has a local scheduler (**LS**). When LS receives a task from GS, the classical work-stealing scheme is applied on the shared memory multi-core node to split and schedule tasks. In the shared memory implementation of work-stealing, each worker thread maintains a task queue. When a thread's task queue is empty, it attempts to steal a task from another thread's task queue. If all the task queues on the node are empty, the LS sends a steal request to the GS and try to get a task from other nodes. Intra-node work-stealing only happens between a thief and a random selected victim. As mentioned in [9], such work-stealing scheme is inappropriate on a distributed system because the random probes introduce significant overhead. Therefore, we let the inter-node work-stealing be centrally controlled by the GS. When GS receives a steal request, it determines which node is the busiest node and selects this node as the victim. In practice, the busiest node is determined in terms of the number of tasks in task queues of a node. In our system, inter-node work-stealing involves not only the thief and the victim, but also the master node.

Because of GS, the termination detection in our system is much more efficient than it in the existing work-stealing algorithms. In the shared memory implementations of work-stealing, termination can be detected with a barrier algorithm [10]. In the distributed memory case, token-based termination detection algorithms have been shown to be more suitable than barriers for termination detection [11][12]. However, these algorithms still need to traverse all of the nodes. It leads to a high communication cost and latency to termination. In our system, GS maintains a task counter for each worker node. When all the counters become zero, the global termination is detected and GS will send a termination message to all the LSs. The overhead of our termination detection method, that is the overhead of updating the task counters, is scattered throughout the execution of an application, which reduces the latency to termination. In our system, each LS periodically sends the total number of tasks in task queues on its node to GS. In addition, the task counters are also updated immediately in the following cases: 1) there is no task in the node and then the LS of the node will send a steal request to the GS, along with updating its task counter; 2) when the stolen task arrives, the task counter of the thief node needs to be updated. Task counters are used not only for termination detection, but also for victim selection. We explain the details of our implementation in the third subsection.

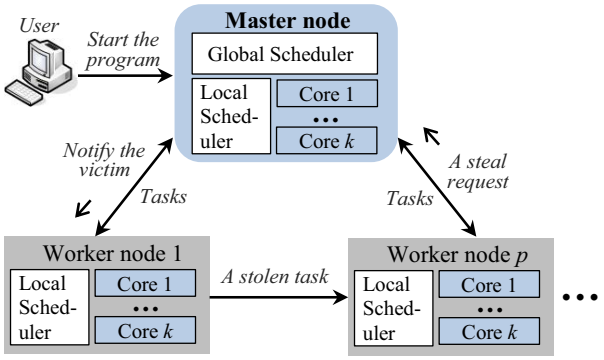

Figure 1. System model for task scheduling in a multicore cluster.

B. Adaptive Initial Partitioning

In general, there are three patterns in task parallelism[2]: ***flat parallelism*** provided by *do-all loops*, ***recursive parallelism*** provided by *divide-and-conquer* algorithms and ***irregular parallelism*** using *task DAG* (Directed Acyclic Graph) for describing parallel algorithms. We use different initial partitioning approaches for above task parallel patterns. These approaches are adaptively selected by the GS.

1) Flat Parallelism

For a parallel loop, initial partitioning determines the loop bounds of the partitions which will be assigned to each node at the beginning. The workload distribution of the loop iterations and the processing capabilities of the worker nodes are two major factors impacting load balancing in this case. There are different types of workload distribution (uniform, increasing, decreasing and random). For the uniform workload, we just partition the loop iterations equally among

[1] In this case, our framework can be improved by using the information of available resources and current loads, which is obtained from the resource manager, to balance the workload.

[2] Pipeline parallelism is a special issue, which is not considered in this paper.

the cluster nodes if the capabilities of the nodes are practically identical. Otherwise, we use a_i to represent the capability of the node P_i (i=1, 2…, p). We normalize a_i with a_1. For instance, a_1=1 and a_2=2, which means that the execution time of the same workload on P_1 is twice as that on P_2. Subsequently, the loop bounds of each partition can be calculated using the following equations under the assumption that the loop is uniform.

$$l_1 = 1; \quad l_{j+1} = u_j + 1;$$
$$u_j = \left\lfloor \frac{\sum_{i=1}^{j} a_i}{\sum_{i=1}^{p} a_i} N \right\rfloor, j = 1, 2, ..., p-1; \quad u_p = N. \qquad (1)$$

where N is the number of iterations, l_j and u_j are the lower and upper bounds of the partition assigned to the node P_j. The complexity of the above computations is $O(p)$. Therefore, the overhead of initial partitioning for the uniform workload is very low. For the non-uniform workload and different capabilities of the nodes, it is a NP-hard problem to partition the workload to achieve optimal load balancing, that is, to minimize the finishing time difference between the workers. We proposed a heuristic method to solve this problem. Details can be found in [13].

The time cost of initial partitioning and scheduling the partitions to the workers is not low in a cluster system, and it increases as the number of the nodes increases. For a parallel loop, this overhead is negligible only if the number of loop iterations (denoted by N) is much larger than the number of cluster nodes (denoted by p) or the execution time of an iteration (denoted by T) is much longer than the time used to dispatch a task to a node (denoted by D). In other cases, this overhead would result in performance degradation. For example, when N=p and T<D, it is definitely inefficient to send each node an iteration. Assuming the overhead of task dispatching grows linearly as more tasks are dispatched. Ideally, the execution time of the parallel loop using p workers is $NT/p+pD$. When $NT/p = pD$, the execution time is minimized, where $p = \sqrt{NT/D}$. We set this value as the maximum number of worker nodes used for the execution of the loop[3].

2) Recursive Parallelism

We use the Fibonacci program as an example to illustrate the initial partitioning for recursive parallel programs. As shown in Fig. 2, there is only a root task fib(40) at the beginning of the execution. The global scheduler maintains two task queues, *ITQ* (intermediate task queue) and *RTQ* (ready task queue), which are both FIFO. The ready tasks are stored in *RTQ* and the others are stored in *ITQ*. Every task has a pointer to its parent and a counter of its child tasks. When the counter becomes zero, the task will be moved from *ITQ* to *RTQ*. When a task spawns child tasks, it will be pushed into *ITQ*. In Fig. 2, the task fib(40) spawns fib(39) and fib(38). Therefore, fib(40) is pushed into *ITQ* and fib(39) and fib(38) are pushed into *RTQ*. Then fib(39) is popped from *RTQ* to be executed and spawns fib(38) and fib(37) which are pushed into *RTQ*. The fib(39) becomes a parent task and is pushed into *ITQ*. Similarly, the first fib(38) task spawns fib(37) and fib(36). The current state of the queues is shown in Fig. 2. Assuming the number of the cluster nodes is p, the above process is continued until the number of tasks in *RTQ* is greater than or equal to p. Then, GS sends each node a task in *RTQ*.

The above process is a recursive process in breadth-first order. It is different from the sequential recursive process which is in depth-first order. This process could quickly produce enough subtasks which have the balanced load, based on the number of available worker nodes.

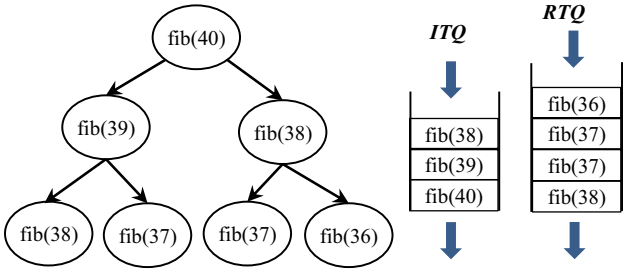

Figure 2. The status of the computation tree and the task queues.

Like for flat parallel programs, it is necessary to determine how many nodes to use for recursive parallel programs if the number of available nodes is large. This problem is more complicated than it for flat parallel programs because task splitting is not simply modifying the loop bounds, the time cost of task splitting is application-dependent, and the overhead of dividing and reducing a task depends on "in" and "out" data of the task. We set a threshold to stop splitting when the tasks in *RTQ* are too small in our implementation.

3) Irregular Parallelism

To support irregular parallelism, an application is normally represented in the form of a task DAG. As shown in Fig. 3(a), the numbers in the circles represent the computational weights of the respective tasks. DAG scheduling [16] can be classified as static or dynamic. Static DAG scheduling allocates all the tasks of an application on the worker nodes to minimize the schedule length or makespan without violating the precedence constraints. The mapping of tasks to the workers is determined before the execution and never changes during the execution. Dynamic DAG scheduling makes decision of the mapping of a task to a worker at runtime. In our system, GS is a dynamic DAG scheduler in practice. For the sample in Fig. 3(a), there are six tasks in *RTQ* initially, as shown in Fig. 3(b). Assuming two worker nodes, P_0 and P_1, are available, we partition the tasks in *RTQ* into two batches and send each node a batch in the initial partitioning phase. After that, the work-stealing mechanism takes charge of load balancing. When there are a lot of tasks in *RTQ*, dispatching tasks as batches can reduce the scheduling overhead and achieve better locality than dispatching tasks one by one. Note that our initial partitioning is not a static DAG scheduling because only the

[3] Determining how many workers to use is a critical issue whether in shared memory systems or distributed memory systems [14][15].

ready tasks in *RTQ* are scheduled and these tasks are independent of each other.

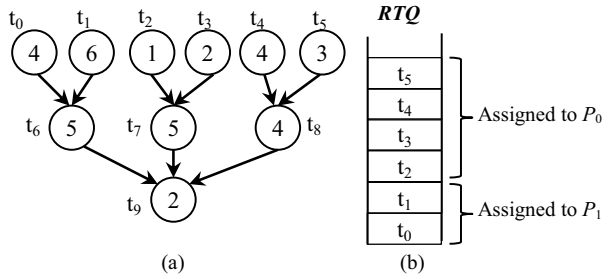

Figure 3. (a) A sample DAG. (b) The initial status of *RTQ*

C. Hierarchical Scheduling Scheme

After initial partitioning, tasks in *RTQ* are scheduled onto the nodes and will be split into subtasks to execute in parallel on multi-cores. A hierarchical scheduling scheme which consists of inter-node work-stealing and intra-node work-stealing will balance the tasks dynamically. As shown in Fig. 1, the local schedulers (LS) deal with intra-node work-stealing and the global scheduler (GS) deals with inter-node work-stealing. In our system, LS implements a classical work-stealing algorithm with work-first scheduling policy, random victim selection and barrier-based termination detection. Certainly, prior techniques [17][18] for improving the efficiency of work-stealing can be used in the implementation of LS.

Task scheduling among cluster nodes is implemented using explicit message passing. We defined message formats for steal requests, task dispatching, etc. The schedulers in our system communicate with each other through these messages. Pseudo code of LS and GS is shown in Fig. 4. GS resets the task counters, makes the initial partitioning, dispatches the ready tasks to the nodes, and then enters a message loop where it waits for a message from LS. When a steal request is received, GS selects the node with the largest task count as a victim and sends a message to it. This message contains the ID of the requester node. If all the task counters are zero, a termination message will be sent to all the worker nodes. When a *UPDATE_TC* message is received, GS updates the task counter of the sender. The updating process needs to be a handshake process because other messages, such as steal request messages, also change the task counters and a race hazard may occur when the message receipt is out of order.

Assume there are n worker threads (w_1, w_2…, w_n) on a node, the LS maintains task queues q_i for w_i (i=1, 2…, n), balances tasks among them and sends the total number of tasks on the node to GS periodically. When a worker thread is idle, it will call a function of LS to ask for a task to run. Pseudo code of this function is above the dashed line in Fig. 4. A variable γ holds the total number of tasks on the node and Δ holds the changes of γ. γ increases by one when a task is spawned and decreases by one when a task is completed. Δ increases both in task spawning and finishing. We use atomic operations to access γ and Δ. A threshold λ is set by user to adjust the frequency of the task counter updating. λ should be an odd value. When Δ is greater than λ, γ is sent to GS and Δ is reset. When γ is zero, which means no task running on the node, a steal request message is sent to GS and the worker threads will be waiting for a message from other node. Note that the message that will be received could be a task message from victim node or a termination message from master node.

```
GS:
InitPartition( );
InitTaskCount( );
while ( true ) {
  ReceiveMsg ( msg );
  switch ( msg ) {
    STEAL_REQ:
      if( all the task counters are zero)
        SendMsg(TERMINATION); …
      else
        Select a victim and SendMsg( VICTIM ); …
    UPDATE_TC:
      Update the task counter and SendMsg( TC_UPDATED );
  …}
}

LS:
if ( Δ > λ ) {  Update( γ );  Δ = 0;  }
if (γ = 0 ) {
  SendMsg (STEAL_REQ);
  Suspend the current thread w_i.  … }
if (q_i != NULL )  Pop a task from q_i to run.
else  t = Random_steal ( );  // steal a task t.
if ( t != NULL )    Run t;  …
--------------------------------------------------------------------------------
while ( true ) {
  ReceiveMsg ( msg );
  switch ( msg ) {
    VICTIM:
      if(γ > φ ){
        SendMsg (TASK);
        ReceiveMsg (TASK_RECEIVED);  …  }
    TASK:
      Update( γ );
      SendMsg (TASK_RECEIVED);
      Resume a worker thread.  ...
    TERMINATION:
      Exit worker threads w_1, w_2…, w_n.
      Break;
  … }
}
```

Figure 4. Pseudocode for the schedulers (GS and LS).

The message loop of LS is shown under the dashed line in Fig. 4, which runs in a separate thread. When a *VICTIM* message is received, the task queues on the node are checked to find a non-empty queue. Then a task is popped from the bottom of this queue. It will be packaged and sent to the thief node. If a few tasks are remained on this node, the stealing would take more overhead than it's worth. Therefore, task is sent to the thief only when γ is greater than a threshold φ. Otherwise, the stealing is canceled. The thief node does not need to be notified of the cancellation because a termination message will soon be sent to all the nodes by GS after the victim node finishes the remaining tasks.

In a steal phase, the victim needs to be blocked waiting for a reply from the thief after it sends a task to the thief. When the thief receives the stolen task, it increases its task counter by sending a *UPDATE_TC* message to GS, then replies to the victim that the task is received. Such design is

to avoid error determining of termination. For example, when a thief receives a stolen task but has not updated its task counter, the victim finishes all its tasks and clears its task counter if it is not blocked waiting for a reply. In this case, all the task counters become zero, GS will send a termination message to all the nodes and the thief would exit before the stolen task is completed.

III. EVALUATION

We compare the implementation of our work-stealing framework (denoted by HWS) with the classical work-stealing implementation (denoted by WS) and the state-of-the-art work-stealing implementation for multi-core HPC clusters (denoted by SWS) [1]. All these implementations use MPI for message passing across nodes and perform work-stealing inside a node before crossing the node boundary. In contrast to HWS, WS and SWS have not an initial partitioning phase and select victim randomly. SWS is different from WS in that SWS uses split queues which alleviate locking overhead and detects termination with a token-based method.

We evaluate the performance of our framework using the following programs:

- **Fib**: Recursive Fibonacci (n=46).
- **Nqueens**: The n-queens problem (n=18).
- **MSort**: Parallel merge sort on an integer array of four Gigabytes.
- **MM**: Standard matrix multiplication using a loop nest where the outermost loop is parallelized (10000*10000 double matrix).
- **Strassen**: Dense matrix multiplication using Strassen's algorithm (10000*10000 double matrix).

Experiments were carried out on a cluster which consists of 16 multi-core nodes. 12 nodes are equipped with 2.13 GHz quad-core Intel Xeon E5606 processors, 12GB of memory for each node. The other 4 nodes are equipped with 2.4GHz quad-core Intel Xeon E5620 processors which support 8 hardware threads, 24GB of memory for each node. Each computation node is connected to a switch on Gigabit Ethernet.

Figure 5 presents the performance comparison of HWS, WS and SWS. The execution times have been normalized w.r.t. WS. We observed that for all five programs, both HWS and SWS show improvements over WS. HWS performs 19% better than WS and 8% better than SWS on average. For MSort, HWS outperforms WS and SWS significantly. MSort transfers large amount of data during the parallel execution. The initial partitioning of HWS reduces the number of inter-node steals, and therefore reduces the data transferred between the nodes, which results in the significant performance improvement. The SWS performance gain is mainly attributed to using split queues and token-based termination detection. We could expect better performance if split queues are used in HWS.

Table 1 lists the number of inter-node steals for different approaches. For WS and SWS, the left column is the number of steal requests and the right column is the number of real task migrations. Because an idle node randomly probe other nodes for work in WS and SWS, the value of the left column is much greater than the value of the right column for WS and SWS. For HWS and HWS1 which is a variant of HWS we will introduce later, the number of real steals almost equals the number of steal requests because the victim is directly determined by the global scheduler, not randomly probed by the thief. Therefore, we only report the number of steal requests in Tab. 1 for HWS and HWS1. From the table, we observe that HWS reduces the number of inter-node steals by almost half, compared with WS. It is attributed to the initial partitioning and the centralized inter-node stealing management.

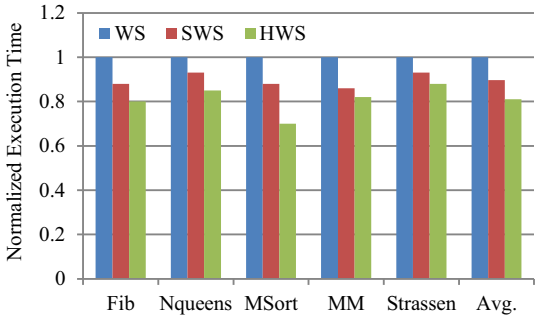

Figure 5. Performance comparison of HWS with WS and SWS.

To evaluate the benefits of initial partitioning in HWS, we remove the initial partitioning phase from our implementation of HWS. The modified implementation is denoted by HWS1. The execution time of HWS1 is reported in Fig. 6. For contrast, results of HWS is also shown in Fig. 6 and all the values are normalized w.r.t. HWS. From the figure, we observe that HWS1 degrades performance by 10.6% on an average. Table 1 shows HWS1 increases the number of inter-node steals by 16% on an average, compared with HWS. In summary, the initial partitioning makes about ten percent performance improvement.

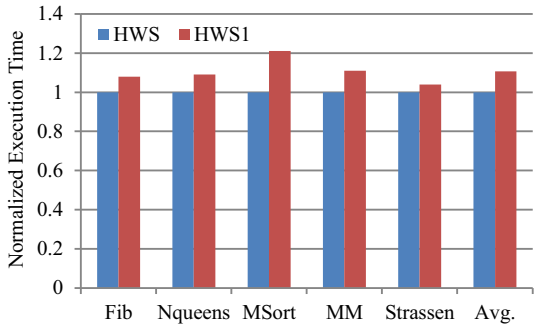

Figure 6. Performance comparison of HWS with HWS1

TABLE I. THE NUMBER OF INTER-NODE STEALS FOR DIFFERENT APPROACHES

	WS		SWS		HWS	HWS1
Fib	2614	1752	2015	1428	1211	1520
Nqueens	5122	3695	4063	3004	3105	3296
MSort	1856	1233	1642	1128	951	1108
MM	432	228	354	221	178	219
Strassen	855	520	792	563	515	562

IV. Related Work

Prior researches of work-stealing mostly focus on the following issues:

Task scheduling policies. The task scheduling policies determine the order of spawning, fetching and stealing. Work-first and help-first are two commonly used task scheduling policies. Work-first is widely used by Cilk and many Cilk-like systems. Under work-first policy, the worker executes a spawned task and leaves the continuation to be stolen. Contrary to work-first policy, the help-first policy dictates that the worker executes the continuation and leaves the spawned task to be stolen. Guo et al. in [19] show that help-first policy works better when stealing is frequent because it can quickly produce enough tasks for free workers. TBB [2] schedules tasks based on a "breadth-first theft and depth-first work" policy, which is similar to Cilk.

Task queues. Task queuing impacts other aspects of the task scheduling. Therefore, besides the traditional distributed task queues, people proposed many improved designs of task queues. Some approaches split a task queue into two parts and only allow tasks in one part to be stolen [18]. Some techniques [20] are proposed to expand the size of a task queue with automatic garbage collection. Hardware task queue is used to improve the performance in [21].

Task granularity. The granularity of tasks is a critical factor in task stealing and task splitting. In traditional work-stealing algorithms, one task is stolen at a time. In [12], a worker steals more than one task from the victim at a time, which reduces the stealing overhead. The performance will degrade if the tasks are too fine-grain or too coarse-grain. The Auto-Partitioner (AP) in TBB decide whether a parallel loop task should be split or not with a manually tuned threshold. In [16], techniques are proposed to find the best granularity automatically.

For clusters and grids, ATLAS[22], Satin[23] and Kaapi[24] use hierarchical work-stealing. [25] extends X10 to balance the workload across X10's places. In [12][18], PGAS programming model is used to implement work-stealing on distributed memory systems. [1] and [11] use MPI to implement work-stealing across cluster nodes like ours. However, none of them adopts initial partitioning and determines victim node directly like ours.

V. Conclusion

In this paper, we proposed a work-stealing task scheduling framework for multi-core clusters. The key characteristics of our framework are an adaptive initial partitioning approach and a novel implementation of the inter-node work-stealing in a hierarchical scheduling scheme. Our experimental results show that our technique outperforms the state-of-the-art techniques. As future work, we would like to incorporate prior techniques such as split task queues [18] and lazy binary splitting [17] into our framework to improve the performance.

References

[1] K. Ravichandran, S. Lee, and S. Pande. Work stealing for multi-core HPC clusters. In *Euro-Par*, pp. 205-217, Berlin, Heidelberg, 2011.

[2] Intel(R) Threading Building Blocks, Intel Corporation.

[3] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pp. 519–538, 2005.

[4] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA*, pp. 227-242, 2009.

[5] OpenMP Architecture Review Board. OpenMP Application Program Interface, v3.0. May 2008.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[7] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP*, pp. 34-43, New York, USA, 2001.

[8] J.-N. Quintin and F. Wagner. Hierarchical work-stealing. In *EuroPar*, pp. 217–229, Berlin, Heidelberg, 2010.

[9] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based global load balancing. In *PPoPP*, pp. 201–212, San Antonio, USA, 2011.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5): 212–223, 1998.

[11] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *PMEO-PDS*, pp. 1-8, 2007.

[12] S. Olivier and J. Prins. Scalable Dynamic Load Balancing Using UPC. In *ICPP*, pp. 123-131, Washington, USA, 2008.

[13] Y. Wang, W. Ji, F. Shi, Q. Zuo and N. Deng. Knowledge-based adaptive self-scheduling. In *NPC*, in press, 2012.

[14] A. Nicolau and A. Kejariwal. How many threads to spawn during program multithreading?. In *LCPC*, pp. 166-183, Berlin, 2010.

[15] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa. Backtracking-based load balancing. In *PPoPP*, pp. 55-64, New York, USA, 2009.

[16] D. Bozdag, F. Ozguner, E. Ekici, and U. Catalyurek. A Task Duplication Based Scheduling Algorithm Using Partial Schedules. In *ICPP*, pp. 630-637, Washington, USA, 2005.

[17] A. Tzannes, G. C. Caragea, R. Barua, U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *PPoPP*, pp. 179-190, 2010.

[18] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *SC*, pp. 1–11, New York, NY, USA, 2009.

[19] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *PPoPP*, pp. 341-342, New York, USA, 2010.

[20] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent Work Stealing. In *PPoPP*, pp. 45–54, 2009.

[21] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA*, 162-173, 2007.

[22] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. Atlas: an infrastructure for global computing. In *Proceedings of the 7th workshop on ACMSIGOPS European workshop*, pp. 165–172, New York, USA, 1996.

[23] R. V. van Nieuwpoort, G. Wrzesińska, C.J.H. Jacobs, and H. E. Bal. Satin: A high-level and efficient grid programming model. *ACM Trans. Program. Lang. Syst.* 32(3), 39 pages, 2010.

[24] MOAIS software: http://kaapi.gforge.inria.fr

[25] O. Tardieu, H. Wang, and H. Lin. A work-stealing scheduler for X10's task parallelism with suspension. In *PPoPP*, pp. 267-276, New York, USA, 2012.