An adaptive and hierarchical task scheduling scheme for multi-core clusters

Yizhuo Wang, Yang Zhang, Yan Su, Xiaojun Wang, Xu Chen, Weixing Ji, Feng Shi

Artificial Information

Article info

Article history:
Received 23 April 2014
Received in revised form 15 August 2014
Accepted 4 September 2014
Available online 7 October 2014

Keywords:
Work-stealing, Multi-core cluster
Task scheduling
Work-sharing

ABSTRACT

Work-stealing and work-sharing are two basic paradigms for dynamic task scheduling. This paper introduces an adaptive and hierarchical task scheduling scheme (AHS) for multi-core clusters, in which work-stealing and work-sharing are adaptively used to achieve load balancing.

Work-stealing has been widely used in task-based parallel programming languages and models, especially on shared memory systems. However, high inter-node communication costs hinder work-stealing from being directly performed on distributed memory systems. AHS addresses this issue with the following techniques: (1) Initial partitioning, which reduces the inter-node task migrations; (2) Hierarchical scheduling scheme, which performs work-stealing inside a node before going across the node boundary and adopts work-sharing to overlap computation and communication at the inter-node level; and (3) Hierarchical and centralized control for inter-node task migration, which improves the efficiency of victim selection and termination detection.

We evaluated AHS and existing work-stealing schemes on a 16-nodes multi-core cluster. Experimental results show that AHS outperforms existing schemes by 11–21.4%, for the benchmarks studied in this paper.

1. Introduction

Today, most existing and new cluster systems are multi-core clusters, which present two levels of parallelism. One is shared memory parallelism within the cluster node. Another is distributed memory parallelism among the cluster nodes. How to exploit both shared and distributed memory parallelism is a critical issue to run a large application efficiently on such systems.

To exploit parallelism of the underlying architecture, the application needs to be parallelized with a particular form. Task parallelism is a popular form of parallelization of program. Most parallel programming languages and models are constructed based on task parallelism, such as Intel TBB [1], IBM X10 [2], Microsoft TPL [3], OpenMP 3.0 [4] and Cilk [5]. In these languages and models, work-stealing [6] has been widely used to achieve dynamic load balancing. Under work-stealing, each processor maintains its own work queue. Tasks are enqueued to or dequeued from the work queue at runtime. When a
processor's task queue is empty, it attempts to steal a task from a victim processor's work queue. Tasks in all queues are independent of each other and thus can be executed in parallel.

Work-stealing has been proven to be an effective method for task scheduling on shared memory systems, in which all the worker threads have the same priority and victim is selected randomly. However, work-stealing is inefficient when extended to distributed memory directly. First, the cost of task transfer between cluster nodes is much higher than between the multiple cores within a node. Traditional work-stealing is not optimal for decreasing the number of task migrations. Second, the random victim selection results in useless probing, especially when work is sparse. On distributed memory systems, the overhead of such probing is not negligible. Third, the thief is idle during work-stealing because of passive stealing. On distributed memory system, high latency of task migration would make the thief node inefficient.

To address above issues, we propose AHS, an adaptive and hierarchical task scheduling scheme for multi-core clusters. AHS perceives two levels of hierarchy: cluster nodes and multiple cores on each node. First, initial partitioning is applied to achieve static load balancing among the cluster nodes. Our experimental results show that the initial partitioning reduces the number of inter-node task migrations by about 6.3%. Second, work-stealing is used for dynamic task scheduling among the multiple cores on each node. Then, work-sharing [7] and work-stealing are adaptively used to achieve dynamic load balancing among the cluster nodes. Work-sharing overlaps the latency of task migration with the computation time of local tasks, which contributes to 8.4% performance gain in our experiments.

The rest of the paper is organized as follows. Section 2 reviews work-sharing and work-stealing techniques. Section 3 describes our system architecture. Sections 4 and 5 present the implementation of our techniques in detail. Section 6 provides the theoretical and simulation studies of our technique. Sections 7–9 contain our experimental results, related work discussion and conclusions respectively.

2. Work-sharing and work-stealing

Task scheduling can be classified as static and dynamic. Static scheduling is a proper scheduling strategy when the tasks are determined before running the application, the number of the tasks does not change during the execution, and the system workloads are almost constant. In realistic cases, however, the above conditions normally cannot be satisfied. Therefore dynamic scheduling is widely studied and used.

Work-sharing and work-stealing are two basic dynamic scheduling approaches. In work-sharing, busy processor elements (PEs) push tasks to idle PEs initatively. Task-pushing happens when a new task is generated. In popular implementations of work-sharing [8,9], each PE maintains a local task queue and fetches a task from this work queue to execute each time. During the execution, if a new task is spawned and the number of tasks in the local queue reaches a certain threshold, the current PE probably has too much work and one or several tasks should be re-scheduled to other lightly loaded PEs. Probing to find an idle PE is time consuming. To avoid such probes, a centralized task queue is normally used in shared memory implementations of work-sharing. When the threshold is reached, some tasks of the busy PE are moved to the centralized task queue. Whenever a PE becomes idle, it will attempt to get a task from the centralized queue. Thus, work-sharing between the busy PE and the idle PE is realized.

In work-stealing, idle PEs pull tasks from busy PEs directly. Each PE maintains a task queue and gets a task from it to execute each time. New tasks are inserted into the task queue at runtime. When the task queue is empty, the PE attempts to steal a task from another PE's task queue. In most existing work-stealing schedulers, an idle PE does not probe all the other PEs to find the busiest one, but selects a victim at random. Random victim selection has been proven to be efficient for shared memory systems [6].

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

2.1. Task scheduling policy

The task scheduling policies determine the order of task spawning, fetching and stealing (in work-stealing) or redistributing (in work-sharing). Work-first and help-first are two commonly used task scheduling policies [7]. Under work-first policy, the worker executes a spawned task and leaves the continuation to be transferred between the workers. Contrary to work-first policy, the help-first policy dictates that the worker executes the continuation and leaves the spawned task to be transferred. Work-first tries to follow the serial execution order, thus preserving data locality in original program. It is widely used by Cilk and many Cilk-like systems. Guo et al. in [10] show that help-first policy works better when task migration is frequent, because help-first can quickly produce enough tasks for other workers. Our technique tries to minimize task migrations. Therefore, work-first policy is used in our system.

2.2. Task queue

Task queue organization impacts all the other aspects of task scheduling. There are some interesting designs of task queue proposed in recent years. For example, some approaches split a task queue into two parts and only allow tasks in one part to

1 Although these issues are commonly discussed in the literature of work-stealing, the same issues go for work-sharing.
be stolen [11]; some techniques expand the size of a task queue with automatic garbage collection [12]; hardware task queue is used to improve the performance in [13]; hierarchical task queues are studied on different parallel systems [14,15], which enable locality-aware victim selection. Considering the two levels of hierarchy in our system, cluster nodes and multiple cores within a node, our task queues are hierarchically organized. Victim is selected inside a node before crossing the node boundary during work-stealing.

2.3. Task granularity

The granularity of tasks is a critical factor in task migration and task splitting. In traditional work-stealing algorithms, one task is stolen at a time. In [16], 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 [17], an approach is proposed to find the best granularity automatically. Above techniques can be incorporated in our system for future work.

On shared memory systems, a task queue of a worker thread can be directly accessed by another worker thread. Thus, the cost of task migration is borne only by the initiator of the migration. In work-sharing, the initiator is a busy processor. In work-stealing, it is an idle processor. Therefore, work-stealing logically is more efficient than work-sharing on shared memory systems. Moreover, the centralized task queue in work-sharing can become a bottleneck when the number of workers or tasks increases. In some work-sharing implementations, there is not local task queue. Whenever a task is spawned, it is inserted into a centralized task queue. In this case, the work-sharing algorithm is actually same with the self-scheduling [18,19] algorithm, in which the synchronization cost for access to the centralized queue is the major scheduling overhead. Work-sharing also can be implemented without a centralized queue. In this case, a busy worker directly pushes a task to a random selected worker, which eliminates the bottleneck of the centralized queue but increases the frequency of the task migrations. As stated above, work-stealing is normally used to achieve dynamic load balancing in most parallel programming models, not work-sharing.

On distributed memory systems, task migration cannot be completed by the initiator alone, like in shared memory systems. The migration costs need to be borne by both the sender and the receiver of a task. Therefore, the advantage of work-stealing in shared memory systems, that is casting the migration cost on an idle processor, no longer exists. We believe both work-sharing and work-stealing should be taken into account when designing a task scheduler for distributed memory system. A good scheduler, using whether work-stealing or work-sharing, should minimize the number of tasks migrations while struggling to balance the workload.

Traditional work-stealing scheme with random victim selection should not be directly used for distributed memory systems due to the following two problems. First, random victim selection would result in many times of useless probing when work is sparse. It would degrade the performance because the cost of probing is not low in distributed memory system. Second, a thief node only steals work when it becomes idle. During stealing, there is not useful work running on it. This makes the thief node inefficient especially when the task migration takes a long time.

Work-sharing is capable of overlapping task migration with useful computations, hence addresses the second problem of work-stealing mentioned above. However, traditional implementations of work-sharing are not suitable for distributed memory systems. First, the bottleneck effect of the centralized queue on distributed memory system is much greater than it on shared memory system. If no centralized queue is used, fine-grained dynamic load balancing cannot be achieved because the load will no longer be balanced when the amount of the work on each worker node reaches a threshold. The threshold normally is not low because lower threshold would cause many more task migrations. Second, whether tasks on a worker node should be re-distributed only depends on the number of tasks on the worker node, which may incur more load imbalance.

To sum up, using work-stealing or work-sharing alone is not appropriate on distributed memory systems. Therefore, we combine these two techniques in our system. Work-stealing is for fine-grained dynamic load balancing and work-sharing is for hiding the task migration latency.
3. System architecture

Fig. 1 depicts our cluster model, in which a fixed number of multi-core workstations are connected by a high speed network. We assume the communication costs between any two nodes are the same. Thus, the multi-core cluster can be viewed as a two-level hierarchical system. One is the distributed memory level which consists of the cluster nodes. Another is the shared memory level which consists of multiple cores within a node. To exploit both inter-node parallelism and intra-node parallelism on these two levels, we propose a hierarchical task scheduling scheme in which tasks are scheduled with different approaches on these two levels in order to achieve dynamic load balancing. Briefly, work-stealing is used for load balancing inside a node and an adaptive approach that supports both work-sharing and work-stealing is used for load balancing among the cluster nodes.

Assume all the program and data files have been deployed on each node in Fig. 1. The user logs on to a node to start the program. 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 manager node. A global scheduler (GS) works on the master node, which is responsible for inter-node task scheduling, including the initial partitioning and the redistribution of tasks between worker nodes. The novel techniques used in the GS distinguish our system from the existing work-stealing systems [14,15,20].

There is no initial partitioning phase in traditional work-stealing schemes. Under traditional work-stealing, one initial task runs on a processing element (PE), new tasks are spawned continually and stolen by idle PEs during the execution. In shared memory systems, a spawned task can migrate to an idle thread very quickly. The absence of initial partitioning just increases a few task migrations and the cost of these migrations is negligible. In distributed memory systems, however, task migration has much higher overhead which is no longer negligible. 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 migrations never happen. Therefore, we adopt an initial partitioning phase in our system. The global scheduler partitions the parallel portions of an application adaptively in this phase, according to the pattern of task parallelism. The detail of our initial partitioning is described in the next section.

After initial partitioning, the tasks which are ready to run will be scheduled onto the cluster nodes and be executed in parallel on each multi-core node. In Fig. 1, every node, including the master node, has a local scheduler (LS) which is responsible for intra-node task scheduling and cooperates with the GS. When LS receives a task from GS, a classical work-stealing scheme is applied on the shared memory multi-core node to split and schedule subtasks.

The LS keeps track of the amount of tasks on the node. According to this value, the LS determines whether an inter-node task migration is necessary.

1. If the amount of tasks is greater than a threshold, the LS sends a work-sharing request message to the GS and hopes to transfer a task in the local task queues to another cluster node which is lightly loaded.
2. If all the local task queues become empty, the LS sends a work-stealing request message to the GS and hopes to help other worker nodes by stealing a task from a heavily loaded node.

Whether the LS decides to push a task to or steal a task from another node, the target node (victim) needs to be determined before the task migration. As mentioned in the above section, random victim selection is optimal for shared memory systems, but it is inefficient when applied to distributed memory systems. In our system, the victim is not randomly selected by the LS, but determined by the GS. As shown in Fig. 1, when the GS receives a work-sharing or work-stealing request, it selects the most lightly loaded node as the victim for work-sharing request or the busiest node as the victim for work-stealing request. Then the GS notifies the victim to migrate a task between it and the requester. Thus, the task migration among the cluster nodes is centralized controlled by the GS. To support such centralized control, the GS needs the real-time information of tasks on all the nodes. The information includes task sizes and task migration costs. However, such information is neither cheap nor easy to get. A simplified approach is using the number of tasks to measure the workload on a node. In our implementation, the GS maintains a task counter for each worker node. Each node periodically updates the task counter with the number of tasks existing in the local task queues of the node, by sending a message to the GS. The task counters are used for (1) determining whether a work-sharing or work-stealing operation should be conducted or not, (2) selecting the victim, and (3) detecting the global termination. The details of the implementation can be found in Section 5.

The use of GS does not mean that the system scalability is limited. On the contrary, our GS/LS design is suitable for building a scalable system. The reasons are as follows. First, tasks are not transferred through the GS, but transferred between two LSs directly. Only a few short messages are exchanged between GS and LS (see the implementation section). Second, as the number of the cluster nodes increases, we can deploy multiple GSs on the system to construct a hierarchical architecture to achieve scalability. Each GS controls a limited number of LSs and the GSs communicate with each other or with an upper level scheduler. Moreover, the multi-level schedulers can be adapted to the topology of the system architecture to improve data locality.

\footnote{In this case, our framework can be improved by utilizing the information of available resources and real-time loads, which is obtained from the resource manager.}
To overlap task migration and task execution, another method which can be applied in our system is to initiate work-stealing requests when the number of tasks in the local queues gets below a threshold (>1). Compared with work-sharing, this method also gives rise to many problems, such as how to determine the threshold, how to do termination detection (traditional “barrier” methods need to be modified) and so on. The two methods have similar realization complexity and just make the overlap at different time. Work-sharing makes it at task spawn-time and the above work-stealing method makes it at task complete-time. In this paper, we focus on evaluating the performance benefit of work-sharing. The comparison between the two methods would be made as future work.

4. Initial partitioning

In this section, we first give the general model of our initial partitioning approach, and then we present three practical approaches which are selected adaptively at runtime, according to the pattern of parallelism. This initial approach can be considered to be a simple form of work-sharing, because work-sharing is generally described as a strategy where work is divided between members of a team [21].

4.1. General model

There are \( N \) ready tasks \( t_1, t_2, \ldots, t_N \). Our goal is to partition these tasks equally among \( p \) worker nodes \((P_1, P_2, \ldots, P_p)\). Let \( S_i \) denote computational weight of task \( t_i \). We assume the communication costs between any two nodes are identical for a unit of transmission. Thus, the costs of distributing a task to different nodes are the same. Let \( C_i \) denote the distributing cost of \( t_i \). Let \( l_j \) and \( u_j \) denote the lower and upper bounds of the index of the tasks distributed to \( P_j \). When all the nodes have the same processing capability, the partitioning is to make the bounds satisfy that for all \( P_j \) \((j = 1, 2, \ldots, p)\)

\[
\frac{\sum_{i=l_j}^{u_j} (S_i + C_i)}{p} = \sum_{i=1}^{N} (S_i + C_i) / p.
\]

Note that

\[
l_1 = 1; \quad l_{j+1} = u_j + 1, \quad j = 1, 2, \ldots, p - 1; \quad u_p = N.
\]

The bounds \( (l_j, u_j) \) can be calculated with \( O(N) \) complexity in this case.

Determining the bounds will be complicated when the capabilities of the nodes are not the same. Let \( a_i \) denote the capability of the node \( P_i \) \((i = 1, 2, \ldots, p)\). 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 \( P_2 \). Then the bounds should satisfy

\[
\frac{\sum_{i=l_j}^{u_j} \left( \frac{1}{a_j} S_i + C_i \right)}{p} = \sum_{i=1}^{N} \left( \frac{1}{a_1} S_i + C_i \right) / p.
\]

However, the value of the summation in the left side of the equation is uncertain before the bounds for each node are determined. We propose a heuristic method to calculate the bounds \( (l_j, u_j) \) in this case [22]. First, the initial values of the bounds are calculated by assuming the nodes have the same processing capability. Then the cost of distributing and running tasks on \( P_j \), denoted as \( T_j \), is calculated as follows:

\[
T_j = \sum_{i=l_j}^{u_j} \left( \frac{1}{a_j} S_i + C_i \right), \quad j = 1, 2, \ldots, p.
\]

Let \( \mu_j \) be the mean of \( T_j \) and \( \sigma_j \) be the variance of \( T_j \) \((j = 1, 2, \ldots, p)\). Our goal is to make \( T_1, T_2, \ldots, T_p \) as equal as possible, i.e., all the nodes finish the execution at approximately the same time. For a node \( P_j \), if \( T_j \) is greater than \( \mu_1 \), which means this node has much workload, we need to decrease the number of tasks assigned to \( P_j \) by adjusting the bounds, and vice versa. The change is defined as \( (\mu_j - T_j)/\bar{t} \), where \( \bar{t} \) is the mean cost of tasks, which is defined as \( \bar{t} = \sum_{j=1}^{p} T_j / N \).

The above procedure is repeated to adjust the bounds until one of the following conditions is met:

1. The maximum number of steps, which is user inputted, is reached.
2. The coefficient of variation (c.o.v.) of \( T_j (\sigma_j/\mu_j) \) becomes less than a given threshold.
3. The variance \( (\sigma_j) \) in the current step is greater than that in the last step.

4.2. Adaptive initial partitioning

In general, there are three patterns in task parallelism\(^3\): 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

\(^3\) Pipeline parallelism is a special issue, which is not considered in this paper.
parallel algorithms. We use different initial partitioning approaches for above patterns. These approaches are adaptively selected by the GS.

4.2.1. 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 could be uniform or non-uniform. For the uniform workload, we just partition the loop iterations equally among the cluster nodes if the capabilities of the nodes are practically identical. Otherwise, the loop bounds of each partition can be calculated using the following equations:

$$l_1 = 1; \quad l_{j+1} = u_j + 1; \quad u_j = \frac{\sum_{i=1}^{j} a_i N}{\sum_{i=1}^{j} a_i}, \quad j = 1, 2, \ldots, p - 1; \quad u_p = N.$$ 

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 $P_j$, and $a_i$ is the capability of $P_i$ (see the previous subsection). The complexity of the above computations is $O(p)$. Therefore, the overhead of initial partitioning for the uniform workload is very low.

If the loop is non-uniform, each iteration or a chunk of iterations can be viewed as a task and approaches in the general model could be applied.

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 spent on dispatching 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 $p\sqrt{NT}$. We set this value as the maximum number of worker nodes used for the execution of the loop.

4.2.2. 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, the 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.

Like for flat parallel programs, it is necessary to determine how many nodes to be used for recursive parallel programs if the number of available nodes is large. This problem is more complicated for recursive parallel programs than 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 stop splitting when the sizes of the tasks in RTQ are smaller than a predefined threshold.

4.2.3. 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 [17] can be classified as static or dynamic. Static DAG scheduling allocates all the tasks of an application on the worker nodes to minimize the

![Fig. 2. The status of the computation tree and the task queues.](image-url)
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, the 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 (P0 and P1) are available, we partition the tasks in RTQ into two batches with approaches in the Section 4.1 (General Model), 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 ready tasks in RTQ are scheduled and these tasks are independent of each other.

5. Hierarchical scheduling

5.1. Overview

Fig. 4 shows a multistage overview of our task scheduling scheme. After initial partitioning, tasks in RTQ are scheduled onto the worker nodes. The tasks are further split at runtime and execute in parallel by multiple cores on each node. To achieve load balancing among the multiple cores within a node, a classical work-stealing algorithm with work-first scheduling policy is implemented in our system. Random victim selection is applied for the intra-node work-stealing, and a barrier is used for termination detection [5]. Certainly, prior techniques [11,23] for improving the efficiency of shared memory work-stealing can be adopted in our system. To achieve load balancing among the cluster nodes, tasks are dynamically migrated with work-stealing or work-sharing policy.

In our system, the 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 actually the overhead of updating the task counters, is scattered throughout the execution of the application, which reduces the latency to termination. Each LS periodically sends the total number of tasks in the local task queues of the node to the GS. In addition, the task counters are also updated during work-stealing and work-sharing.

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

Fig. 4. Overview of AHS scheduling algorithm.
5.2. Implementation

Task scheduling among cluster nodes is implemented using explicit message passing. We defined messages for work-stealing requests, work-sharing requests, task dispatching, etc. The schedulers in our system communicate with each other through these messages. Pseudo code of the global scheduler (GS) and the local scheduler (LS) is shown in Fig. 5. We implemented GS and LS in a runtime library.

The 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 the LS. When a work-stealing request is received, the GS selects the most heavily loaded node as a victim. Let \( c_i \) denote the task count of the node \( P_i \) \((i = 1, 2, \ldots, p)\). The victim \( P_v \) is the node with the largest task count \( c_v \). Now the question is, whether it’s worthwhile to transfer a task from the victim to the requester. Here, we set a threshold \( T_{wh} \). If \( c_v \) is greater than \( T_{wh} \), the GS sends a message to the victim to start the task migration. The \text{VICTIM} message contains the address of the requester node. In addition, if all the task counters are zero, a termination message will be sent to all the worker nodes.

When a work-sharing request is received, the GS select the most lightly loaded node as a victim, which is the node with the smallest task count \( c_i \). Like work-stealing, a threshold \( T_{wh} \) is set for work-sharing. If \( c_i \) is greater than \( T_{wh} \) which means the victim also has a lot of work to do, no task should be transferred to it. Therefore, only when \( c_i \) is less than \( T_{wh} \), the GS sends a message to the requester to start the task migration. This message contains the address of the victim node.

When a \text{UPDATE_TC} message is received, the GS updates the task counter of the sender. The updating process is designed 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.

Next, let us see the implementation of the LS. Assume there are \( n \) worker threads \( \{w_1, w_2, \ldots, w_n\} \) on a node, the LS maintains task queues \( Q_i \) for \( w_i \) \((i = 1, 2, \ldots, n)\), balances tasks among them and sends the total number of tasks on the node to the GS periodically. When a worker thread is idle, it will call a function of the LS to ask for a task to run. Pseudo code of this function is above the dashed line in Fig. 5. A variable \( \gamma \) holds the total number of tasks on the node and \( \Delta \) holds the changes of \( \gamma \). \( \gamma \) increases by one when a task is spawned and decreases by one when a task is completed. \( \Delta \) increases both in task spawning and finishing. We use atomic operations to access \( \gamma \) and \( \Delta \). A threshold \( T_f \) is set by user to adjust the frequency of the task counter updating. When \( \Delta \) is greater than \( T_f \), \( \gamma \) is sent to GS and \( \Delta \) is reset. \( T_f \) should be an even number because the total number of tasks may not change while \( \Delta \) equals an even number. When \( \gamma \) 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 at a barrier. Note that the message that will be received could be a task message from the victim or a termination message from the master node.

A work-sharing request is sent from the LS to the GS when a new task is spawned and \( \gamma \) is greater than a threshold \( T_s \). For applications with fixed number of parallel tasks, the work-sharing request can be sent periodically after a number of tasks are finished.

The message loop of LS is shown under the dashed line in Fig. 5, which runs as a separate thread. When a \text{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 another node. The address of the target node is attached in the message. Note that this message could be received in both work-sharing and work-stealing. In work-stealing, it is received by the victim node. If a few tasks are remained on the victim, the stealing may take more overhead than it’s worth. Therefore, task is sent to the thief only when \( \gamma \) is greater than a threshold \( T_s \). 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 the GS after the victim finishes the remained tasks. In work-sharing, the \text{VICTIM} message is received by the work-sharing requester node. From the perspective of task migration, work-sharing and work-stealing handle this message in the same way.

During task migration, the task sender needs to be blocked waiting for a reply from the receiver after the \text{TASK} message is sent out. When the receiver receives the task, it increases its task counter by sending a \text{UPDATE_TC} message to the GS, then replies to the sender that the task is received. Such design is to avoid error determining of termination. For example, in work-stealing, when a thief receives a stolen task but has not updated its task counter, the victim finishes all its tasks and would clear its task counter if it is not blocked waiting for a reply. In this case, all the task counters become zero, the GS will send a termination message to all the nodes and the thief would exit before the stolen task is completed.

For divide-and-conquer applications, a lot of tasks may be quickly spawned at the beginning of the execution. To avoid too many work-sharing requests and task counter updates at the beginning of the divide procedure, we set two enable flags \( E_h \) and \( E_o \) for each node. Work-sharing is not allowed when \( E_h \) is 0, and the task counter will not be updated when \( E_o \) is 0. The LS initializes \( E_h \) and \( E_o \) to 0. When a stop splitting threshold is first reached, the LS set \( E_o \) to 1. After that, the LS send \text{UPDATE_TC} messages to the GS periodically. The GS determines whether the minimal task counter is less than a given threshold after it receives \( 2p \) \text{UPDATE_TC} messages. \( E_h \) is enabled when the determination returns true.

To support inter-node task migration, two functions, serialize and un serialize, are defined in our system. When a task is going to be transferred to another node, the function serialize is called to package the task information into a message. Then this message is sent to another node and the task is unserialized from the message to run. The task information is related to the application. For example, in a standard matrix multiplication program, only some row and column numbers are needed to represent a task.
6. Analysis

The novelty of this research includes two major aspects: (1) a hierarchical task scheduling framework, and (2) using work-sharing and work-stealing together to achieve load balancing. Hierarchical work-stealing (HWS) has been adopted in Kaapi [24], X10 [2], etc. A brief analysis of HWS is given in [14]. Our hierarchical framework has two levels, inter-node and intra-node. At the intra-node level, traditional work-stealing scheduling is applied. Such work-stealing in shared

---

**GS:**
InitPartition();
InitTaskCount();
while (true) {
    ReceiveMsg(msg);
    switch (msg) {
    STEAL_REQ:
        if(all the task counters are zero)
            SendMsg(TERMINATION); ...
        else{
            Select a victim \( P_v, c_v = \max\{c_1, c_2, ..., c_p\} \).
            if \( c_v > T_w \)
                SendMsg(VICTIM); ...} // This message is sent to the work-stealing victim, containing the ID of the requester.

    SHARE_REQ:
        Select a victim \( P_v, c_v = \min\{c_1, c_2, ..., c_p\} \).
        if \( c_v < T_w \)
            SendMsg(VICTIM); // This message is sent to the work-sharing requester, containing the ID of the victim.

    UPDATE_TC:
        Update task counter and
        SendMsg(TC_UPDATED);
    ...
    }

**LS:**
if (\( \Delta > T_r \)) Update(\( \gamma \)); \( \Delta = 0 \); }
if (\( \gamma = 0 \)) {
    SendMsg(STEAL_REQ);
    Suspend the current thread \( w_r \). ...
} if (\( Q_l != NULL \)) Pop a task \( t \) from \( Q \).
else \( t = \text{Random}\_\text{steal}(\) ); // steal a task \( t \).
if (\( t != NULL \)) Run \( t \);
if (\( \gamma > T_h \)) SendMsg(SHARE_REQ); // Determine at task spawn-time.

while (true) {
    ReceiveMsg(msg);
    switch (msg) {
    VICTIM:
        if(\( \gamma > T_r \)){
            SendMsg(TASK);
            ReceiveMsg(TASK_RECEIVED); ... }

    TASK:
        Update(\( \gamma \));
        SendMsg(TASK_RECEIVED);
        Resume a worker thread. ...

    TERMINATION:
        Exit worker threads \( w_1, w_2, ..., w_n \).
        Break;
    ...
    }
}

---

**Fig. 5.** Pseudocode for the schedulers (GS and LS).
memory multi-cores has been widely studied both in theory and practice [6,25–28]. Therefore, we focus on the combination of work-sharing and work-stealing at the inter-node level in this section, in order to see whether the work-sharing can provide performance benefit when it is used in conjunction with work-stealing.

There are three ways to evaluate a novel technique: theoretical analysis, simulation, and direct experimentation. Our experimental results in a real system are presented in the next section. Theoretical analysis of a scheduling method is normally based on probabilistic model, in which the dynamic scheduling is often represented by a Markov process [29], and the bound on the execution time is analyzed based on potential functions [30,31,42] or differential equations [32,33].

An application is represented as a DAG for the analysis. For general work-stealing scheduling, Blumofe and Leiserson proved $\mathbb{E}[T_p] \leq T_1/p + O(T_\infty)$ [6], where $T_p$ is the execution time with $p$ processors, $T_1$ is the number of tasks, $T_\infty$ is the critical path of the DAG. Arora, Blumofe and Plaxton later proved that [30]

$$\mathbb{E}[T_p] \leq \frac{T_1}{p} + 32 \cdot T_\infty$$
$$\mathbb{P}\left\{ T_p \geq \frac{T_1}{p} + 64 \cdot T_\infty + 16 \cdot \log_2 \frac{1}{\varepsilon} \right\} \leq \varepsilon$$

for a binary tree DAG. It is applicable to early thread parallel programs, but not quit suitable for the popular task parallel programs which are represented as unrestricted DAG. Tchiboukdjian, Gast and Trystram studied unrestricted DAG model and proved [42]

$$\mathbb{E}[T_p] \leq \frac{T_1}{p} + 3.24 \cdot (T_\infty + \frac{1}{2\log_2 1}) + 1$$
$$\mathbb{P}\left\{ T_p \geq \frac{T_1}{p} + 3.65 \cdot (T_\infty + \log_2 \frac{1}{\varepsilon}) + 1 \right\} \leq \varepsilon$$

which is the best bound on the makespan for work-stealing to date. The work-sharing in our technique is incorporated into the work-stealing scheme. If we think of the initiator in work-sharing as the victim in work-stealing and the receiver in work-sharing as the thief in work-stealing, the work-sharing process is identical to the work-stealing process. Therefore, the existing analysis results apply to our technique. That is, the above equations also give theoretical bounds of our scheduling scheme.

The communication cost between tasks and the cost for task migration and distribution are assumed to zero in the existing theoretical analyses of work-stealing because work-stealing algorithms most often target shared memory systems. But this assumption is not applicable to our method because we adopt work-sharing mainly to overlap communication and computation. However, if the communication cost is considered in the theoretical analysis, it becomes much more complicated and difficult to analyze. We left it as further work.

In this section, we mainly focus on the simulation study. We implement a task scheduling simulator and use the TGFF tool [34] to generate task graphs (DAGs). All the code is available online.4 Fig. 6 shows the basic working procedure of our simulator. The tasks of the DAG in Fig. 6(a) are dynamically scheduled onto the processor $P_0$ and $P_1$. The edge weights denote communication cost between two tasks. There are two timelines for each processor in Fig. 6(b). The black one is the execution timeline and the grey one is the communication timeline. The dashed part of the execution line indicates the processor is idle for execution, where the processor is receiving the necessary data of the next task in its ready task queue or stealing a task from another processor. The communication line is dashed when the processor does not send or receive data. Using the two timelines meets the actual situations in which a dedicated thread is used for network communication. Assume that task $t_0$ and $t_1$ are in the ready task queue of $P_0$ at the initial time, and the migration cost of $t_1$ is 10 time units. Then, $P_1$ steals $t_1$ from $P_0$ to execute at time 10. If the in-degree of the successor of the current node is greater than one, we need to determine which processor will run this successor node. The rule we adopt is that the successor is scheduled to run on the processor which finishes one of the predecessor nodes first. Thus $t_1$ will be executed by $P_1$. When $P_1$ finishes $t_0$, $(1)$ $t_2$ becomes a ready task to be executed by $P_0$ directly, where we ignore the communication cost between tasks running on the same processor, and $(2)$ $P_0$ sends the data required by $t_3$ to $P_1$, which takes 20 time units. Because the communication is performed by both the sender and the receiver, the communication lines of $P_0$ and $P_1$ both extend forward for 20 time units. Then $P_1$ executes $t_3$ at time 120.

---

The following 6 task scheduling algorithms are simulated.

- **WS**: Cilk-style work-stealing algorithm, in which one task is stolen each time.
- **WShalf**: Steal-half work-stealing algorithm, in which half of the tasks of the victim is stolen each time.
- **AHS**: The proposed adaptive and hierarchical scheduling algorithm, in which work-sharing is used in conjunction with work-stealing and one task is migrated each time.
- **AHShalf**: Modified AHS, in which half of the tasks are migrated from heavy loaded processor to light loaded ones each time.
- **WSL**: Cilk-style work-stealing algorithm with accurate victim selection policy, which calculates the total load of each processor, that is the sum of computation costs of the tasks in the ready task queue of each processor, and selects the processor with the maximum load as the victim.
- **AHSL**: AHS with accurate work-stealing victim selection and work-sharing target selection policy.

First, we use TGFF to generate 100 task graphs with 2000 ± 200 nodes (tasks). Each task has two properties, computation cost and migration cost. The migration cost is around 1/10 of the computation cost and the communication cost between two tasks is 3/10–5/10 of the computation cost. In the case, the communication cost and the migration cost are not negligible. The results are reported in Fig. 7. Compared to WS, WShalf achieves 4.6% speedup, AHS achieves 13.1% speedup and AHShalf achieves 11% speedup in average. Thus, AHS outperforms WShalf by about 8.5%. In addition, AHShalf is not better than AHS, especially when the number of processors is large. The performance of WSL is close to WShalf and AHSL is worse than AHS. It implies that the optimal selection at each scheduling step cannot yield an overall optimization. That is, the practical selection methods based on task counters are efficient enough.

Second, in order to see whether the work-sharing can improve performance on shared memory systems, we simulate the algorithms with 100 task graphs (2000 ± 200 nodes) in which task communication cost and migration cost are set to be zero. The results are in Fig. 8. AHS outperforms WS by 7% and outperforms WShalf by 3.5% in average. It suggests that work-sharing yields little performance improvement. We collected the number of work-sharing and work-stealing for each algorithm. AHS shows less times of work-stealing than WS and WShalf, but more times of task migration (including work-sharing and work-stealing) than them. That is, AHS balances workload more frequently.

Third, we generate 1080 task graphs with 800–3200 nodes which have various in/out-degree, communication cost, task migration cost and graph types. By studying the large number and great variety of task graphs, we want to investigate the
expected performance gain achieved by AHS. Each normalized execution time in Fig. 9 is an average of all the task graphs. As the figure shows, the relative performance of each algorithm changes little with the increase of the number of processors. AHS outperforms WS by about 7% and outperforms WShalf by about 5%. AHShalf is a little worse than AHS, AHS-L is a little better than AHS and WSL is a little better than WShalf.

6.1. Parameters analysis

There are three parameters $T_{ws}$, $T_h$ and $T_{wh}$ (see Section 5.2), influencing the performance of AHS. $T_{ws}$ is the threshold for work-stealing, $T_h$ is the threshold for the task sender in work-sharing, and $T_{wh}$ is the threshold for the task receiver in work-sharing.

First, we evaluate WS, WShalf, AHS and AHShalf with different setting of $T_{ws}$. WSL and AHSL are not evaluated as their victim selection policies do not use $T_{ws}$. We set 4 processors, $T_h = T_{wh} = 3$, and take the above 1080 task graphs as input in the simulation. The results are in Fig. 10. As the figure shows, the execution time of WS increases most significantly as we increase the value of $T_{ws}$. The execution time increases of WShalf and AHShalf are also remarkable. But the execution time of AHS changes very little. Increasing the value of $T_{ws}$ causes the reduction of the number of real work-stealing and the increase of the total number of victim probing, which degrades performance of WS and WShalf. For AHS and AHShalf, work-sharing is another way to balance workload at runtime besides work-stealing, which benefits the performance. Collecting the number of work-sharing, work-stealing and probing, we observe that the number of work-sharing increases by 3 times for AHShalf and 1.2 times for AHS, the changes of the number of work-stealing for AHS and AHShalf are similar, and the number of probing for AHShalf increases much faster than AHS, while the value of $T_{ws}$ increases from 2 to 17. The large number of probing is the reason of AHShalf’s performance degradation.

Second, we investigate the effect of $T_h$ and $T_{wh}$ on the performance of AHS and AHShalf. Simulation shows it is probably best to set $T_h$ equal to $T_{wh}$. Then, the last thing to know is how much they should be set to. As the performance is impacted by the three parameters in combination, we do the simulation with setting $T_{ws}$ to 2 and 16 respectively. Fig. 11 shows the execution times of AHS and AHShalf with different setting of $T_h$ ($T_{wh} = T_h$). When $T_{ws}$ is set to 2, the execution times change slightly, which implies that load balancing is mainly achieved by work-stealing in this case. In addition, we observe that the number of work-stealing increases, the number of work-sharing decreases and the total number of task migrations decreases a little with the increase of $T_h$. When $T_{ws}$ is set to 16, AHShalf achieves the best performance at $T_h = 5$, and the performance of AHS decreases with the increase of $T_h$. A small value of $T_h$ leads to frequent work-sharing, which increases the
total scheduling cost. That is the reason for the performance degradation when \( T_h \) is less than 5. In general, a small value of \( T_h \) is better for AHS.

7. Experiments

7.1. Experimental setup

We compare the implementation of our scheduling scheme (denoted by AHS) 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 HWS) [20]. All these implementations use MPI for message passing across nodes. In contrast to WS and HWS, AHS has an initial partitioning phase, support work-sharing, and select victim not randomly. HWS is different from WS in that HWS uses split queues which alleviate locking overhead and detects termination with a token-based method.

We evaluate the performance of the scheduling schemes using the following programs:

- **Fib**: Recursive Fibonacci \((n = 46)\).
- **Nqueens**: The \(n\)-queens problem \((n = 16)\).
- **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 \((10,000 \times 10,000 \text{ double matrix})\).
- **Strassen** [35]: Dense matrix multiplication using Strassen’s algorithm \((10,000 \times 10,000 \text{ double matrix})\).

All the programs are coded in their natural way, and each has its own task definition. The tasks and associated data are transferred between cluster nodes during inter-node task migrations. For Fib and Nqueens, small amount of data is transferred as they are computation-intensive problems, each task contains all necessary information and no extra data is required for its execution. For MSort, a large amount of data needs to be transferred during merge. We split the data into packages of 32 KB and overlap data communication and computation through double buffering in our implementation. For MM, each task is responsible for the computation of a block of the result matrix. The result submatrices need to be transferred to a specific node. For Strassen, there are two input matrices for each task. If one of them does not present in the node executing the task, it must be fetched from another node. Thus, a substantial amount of data is transferred.

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

To evaluate the benefits of initial partitioning in AHS, we remove the initial partitioning phase from our implementation of AHS. This modified implementation is denoted by AHS-I. To evaluate the benefits of work-sharing in AHS, we remove \textit{SHARE_REQ} message and related processing. This implementation is denoted by AHS-II. In all, we implemented following five scheduling schemes:

- **WS**: Classic work-stealing [6].
- **HWS**: Hierarchical work-stealing [20].
- **AHS**: Adaptive and hierarchical scheduling.
- **AHS-I**: AHS without initial partitioning.
- **AHS-II**: AHS without work-sharing.
7.2. Results

Fig. 12 presents the performance comparison of the above scheduling schemes. Each performance result is the average of 50 executions. The execution times have been normalized w.r.t. WS. We observed that for all five programs, both AHS (including the two varieties, AHS-I and AHS-II) and HWS show improvements over WS. AHS achieves the best performance, which performs 21.4% better than WS and 11% better than HWS in average. Compared with AHS, AHS-I and AHS-II degrade performance by 5.8% and 8.4% in average respectively. The results indicate that the initial partitioning contributes to about 5.8% performance gain of AHS and work-sharing contributes to about 8.4% performance gain of AHS. In addition, we observe AHS outperforms WS and HWS significantly for MSort. The reasons for this are as follows. (1) The cost of task migration in MSort is high because large amounts of data need to be transferred in the merge step. Work-sharing in AHS overlaps task migration and task execution, which results in the significant performance improvement. (2) The initial partitioning of HWS is high because large amounts of data need to be transferred in the merge step. Work-sharing in AHS overlaps task migration and task execution, which results in the significant performance improvement. (2) The initial partitioning of HWS reduces the number of inter-node steals, and therefore reduces the data transferred between the nodes.

We list the number of inter-node task migrations for different scheduling schemes in Table 1. For WS and HWS, the “Requested” column is the total number of steal requests and the “Real” column is the number of steals which have actually happened. Because WS and HWS use random victim selection, an idle node could be selected as a victim, especially while there are a few tasks remained. Therefore, the number of real steals is much less than the number of steal requests. For AHS, whenever the GS received a work-sharing or work-stealing request, the “Requested” number is increased by one. Because the GS determines whether the re-distribution is worth doing and some requests are aborted, the number of real task migrations is less than the number of requests. From the table, we see that AHS decreases the numbers of requested and real task migrations by 9.0–8.4%, compared with WS and HWS. The major reason is that the victim is directly determined by the global scheduler, not randomly probed by the requester in AHS. Compared with AHS, AHS-I increases the number of inter-node task migrations by 6.3% in average because of the absence of initial partitioning, which results in the performance degradation by about 5.8%. For AHS-II, the absence of work-sharing makes the number of task migration requests decreased by 8.6%. But the real migrations increases by 7.7%.

The overhead of initial partitioning in AHS is shown in Table 2, which is the time between program start and when every target node receives a task. Compared with the total execution time, the overhead is so small as to be negligible. Fib is a divide-and-conquer program, in which the first few steps of the recursive calculation are done at the initial partitioning phase. Therefore the overhead for Fib is relatively high. Similarly, the first ready tasks in Strassen belong to different types and they are generated with a recursive calculation at the initial partitioning phase, which results in the relatively high overhead.

The work-sharing threshold in Fig. 4, that is \(T_h\) in Section 5.2, is a critical factor of AHS, which affects the frequency of work-sharing. Fig. 13 shows the execution times while the threshold is set to different values. We see that the best threshold is not too large or too small, and the value of it is relative to the application. Generally, small threshold causes more performance degradation than large threshold, compared to the best case. Therefore, we suggest using a larger threshold in practice. In future work, we will find a method to derive the optimal threshold automatically.

### Table 1

<table>
<thead>
<tr>
<th></th>
<th>WS Requested</th>
<th>Real</th>
<th>HWS Requested</th>
<th>Real</th>
<th>AHS Requested</th>
<th>Real</th>
<th>AHS-I Requested</th>
<th>Real</th>
<th>AHS-II Requested</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib</td>
<td>2614</td>
<td>1752</td>
<td>2015</td>
<td>1428</td>
<td>1689</td>
<td>1302</td>
<td>1771</td>
<td>1382</td>
<td>1590</td>
<td>1383</td>
</tr>
<tr>
<td>Nqueens</td>
<td>512</td>
<td>3695</td>
<td>4063</td>
<td>3004</td>
<td>3610</td>
<td>3024</td>
<td>3906</td>
<td>3211</td>
<td>3364</td>
<td>3212</td>
</tr>
<tr>
<td>MSort</td>
<td>1856</td>
<td>1233</td>
<td>1642</td>
<td>1128</td>
<td>1406</td>
<td>910</td>
<td>1505</td>
<td>988</td>
<td>1328</td>
<td>1007</td>
</tr>
<tr>
<td>MM</td>
<td>432</td>
<td>228</td>
<td>354</td>
<td>221</td>
<td>293</td>
<td>202</td>
<td>321</td>
<td>216</td>
<td>249</td>
<td>221</td>
</tr>
<tr>
<td>Strassen</td>
<td>855</td>
<td>520</td>
<td>792</td>
<td>563</td>
<td>678</td>
<td>512</td>
<td>729</td>
<td>531</td>
<td>611</td>
<td>542</td>
</tr>
<tr>
<td>Avg. (&quot;AHS-I or AHS-II&quot; – AHS)/AHS</td>
<td>+7.4%</td>
<td>+6.3%</td>
<td>-8.6%</td>
<td>+7.7%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
All the 16 nodes are used in the above experiments. We conducted another set of experiments for a variable number of nodes from 2 to 16. All the nodes are the same with the above E5606 nodes. Fig. 14 presents the speedup curves for AHS and WS. Speedup is calculated relative to the one node implementation. We see that AHS outperforms WS in most cases. Generally, greater performance gains are attained for the larger number of nodes, comparing AHS to WS. In addition, MSort, MM and Strassen achieve the best speedups on 8, 12 and 10 nodes respectively for AHS. It indicates that determining the number of nodes to use is important for performance. Nevertheless, the research on this issue is not very extensive, suggesting that more work needs to be done to predict the optimal number of active nodes.

### 8. Related work

Work-stealing has been implemented in many parallel programing models and languages, such as Cilk [6], Cilk++ [36], Intel TBB [1], IBM X10 [2], Microsoft TPL [3], OpenMP 3.0 [4], Java Concurrency Utilities [8], ATLAS [37], Kaapi [24], Wool [38] and Satin [39]. Some of them are designed for shared memory systems and some for distributed memory system. Based on traditional work-stealing scheme, some techniques have been proposed to improve performance in several ways which are introduced in Section 2.

### Table 2

<table>
<thead>
<tr>
<th></th>
<th>Fib</th>
<th>Nqueens</th>
<th>MSort</th>
<th>MM</th>
<th>Strassen</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitP cost</td>
<td>954 µs</td>
<td>412 µs</td>
<td>217 µs</td>
<td>240 µs</td>
<td>1865 µs</td>
</tr>
<tr>
<td>Exe time</td>
<td>22 s</td>
<td>57 s</td>
<td>161 s</td>
<td>423 s</td>
<td>216 s</td>
</tr>
</tbody>
</table>

**Fig. 13.** Impact of the work-sharing threshold on the performance. Horizontal axis is the number of tasks in the local queues. Vertical axis is execution time (s).

**Fig. 14.** Comparison of AHS with WS when varying the number of nodes.
The communication cost between nodes is not negligible and sometimes is not uniform on distributed memory system.
For this reason hierarchical work-stealing approaches were devised. In [14,15,20], 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. ATLAS, Satin and Kaapi implement hierarchical work-stealing in practice.

There are two ways to implement work-stealing on a multi-core cluster, PGAS and MPI. In [11,16], PGAS programming
model is used to implement work-stealing on distributed memory systems. [20] and [40] use MPI to implement work-
stealing across cluster nodes like ours. None of them adopts initial partitioning and determines victim node directly like ours.
Work-sharing is rarely used singly. Habanero-Java and Habanero-C [9] provide both work-sharing and work-stealing
schedulers. However they are selected by programmer, not like ours in which they are used adaptively. In [41], a hierarchical
work-stealing scheme is proposed for multi-core clusters. But it does not adopt work-stealing to overlap the latency of task
migration with the computation time of local tasks.

9. Conclusions

In this paper, we proposed an adaptive and hierarchical task scheduling scheme (AHS) for multi-core clusters, in which
work-stealing and work-sharing are used together to achieve dynamic load balancing. We describe a practical implementa-
tion of AHS, in which a global scheduler makes an initial partitioning of tasks with respect to the pattern of task parallelism,
and cooperates with local schedulers by message passing. Work-stealing is implemented by the local schedulers to balance
load between worker threads on a cluster node, and work-sharing is used in conjunction with work-stealing to achieve load
balancing between the cluster nodes. We present the theoretical, simulation and experimental studies of our technique. The
results show that work-sharing provides performance benefit and AHS outperforms the existing work-stealing schemes with
real programs. As future work, we would like to test AHS in a large scale context with more cluster nodes and with some
other scientific intensive applications. These tests will allow us to better analyze the behavior of AHS.

Acknowledgements

The authors thank Dr. Rosario Cammarota, Professor Alexandru Nicolau, and Professor Alexander V. Veidenbaum for their
inputs during discussion sessions pertaining to the present study, and the anonymous reviewers for their valuable comments
on the manuscript. This work was partially supported by the National Natural Science Foundation of China under Grant NSFC –
61300011.

References

Y. Wang et al. / Parallel Computing 40 (2014) 611–627


