Scheduling Efficiently for Irregular Load Distributions in a Large-scale Cluster*

Bao-Yin Zhang[1], Ze-Yao Mo[1], Guang-Wen Yang[2], and Wei-Min Zheng[2]

[1] Institute of Applied Physics and Computational Mathematics, Beijing, 100088, P.R. China
[2] Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, P.R. China
zby@tsinghua.edu.cn

Abstract. Random stealing is a well-known dynamic scheduling algorithm. However, in a large-scale cluster, an idle node must randomly steal many times to obtain a task from another node, especially, this problem severely affects performance in systems where only a few nodes generate most of the system workload. In this paper, we present an efficient dynamic scheduling algorithm, Transitive Random Stealing (TRS) based on random stealing, which makes any idle node rapidly obtain a task from another node for irregular load distributions in a large-scale cluster. Then by the random baseline technique, we experimentally compare TRS with Shis, one of load balance policies in the EARTH system, and random stealing for different load distributions in the Tsinghua EastSun cluster and show that TRS is a highly efficient scheduling algorithm for irregular load distributions in a large-scale cluster. Finally, TRS is implemented in the Jcluster environment, a high performance Java parallel environment, and an experiment result is given in the HKU Gideon 300 cluster.

Keywords: Scheduling, irregular load distribution, large-scale cluster, transitive random stealing.

1 Introduction

The availability of high speed networks and increasingly powerful commodity microprocessors is making the usage of clusters of computers an appealing vehicle for cost-effective parallel computing. The scale of the clusters is becoming more and more large, which is up to hundreds of and thousands of nodes. In order to achieve scalable performance, it is important to evenly schedule the workload among the processing nodes. Two basic approaches [6] to dynamically scheduling task loads can be found in current literature - *random stealing* and *work sharing*.

Random stealing attempts to steal a task from a randomly selected node when a node finds its own task queue empty, repeating steal attempts until it succeeds. Random stealing is provably efficient in terms of time, space, and communication

* This work is supported by Chinese NSF for DYS granted by No. 60425205 and National Postdoctor Science Foundation of China.

Y. Pan et al. (Eds.): ISPA 2005, LNCS 3758, pp. 39–48, 2005.

for the class of fully strict computations [4, 13]; and the natural work stealing algorithm is stable [2]. Communication is only initiated when nodes are idle. When the system load is high, no communication is needed, causing the system to behave well under high loads. Some systems that implement random stealing include Cilk [3], Jaws [8], and Satin [9]. Cilk [3] provides an efficient C-based runtime system for multithreaded parallel programming with a random stealing scheduler. JAWS [8] efficiently schedule load over a dynamically varying computing infrastructure with random stealing algorithm, Satin [9] presents a system for running divide-and-conquer programs on distributed memory systems with random stealing. The EARTH runtime system [7] supported several dynamic load balancer policies, which goal is to design simple balancers that deliver good load distribution with minimum overheads. But a virtual ring network topology is adopted in all the balancers with nodes numbered clock-wise. The authors of the paper [5] evaluate these load-balancing schedulers for a fine-grain multithreading environment.

In this paper, we study the dynamic scheduling algorithms for a large-scale cluster. For random stealing in a large-scale cluster, an idle node must randomly steal many times to obtain a task from another node. Especially, this problem severely affects performance in systems where only a few nodes generate most of the system workload [12]. For overcoming this problem, *Shis*, one of load balance policies in the EARTH system [5], which slightly modifies random stealing was to remember the originating node (history information) from which a task was last received, and to send requests directly to that node. The authors of the paper [11], present two relatively complicated adaptive location policies which record more history information for global scheduling algorithms.

Here we present a scheduling algorithm, Transitive Random Stealing (TRS), which further improves Shis not only remember the originating node from which a task is stolen but also *forward* the information of the node to other remote nodes which want to steal a task from it. With the transitive policy, TRS can make any node obtain a task faster with less times to steal in a large-scale cluster, reduce the idle time for all nodes and improve the overall performance of the system. Then by the random baseline technique, we experimentally compare the performance of TRS with Shis and random stealing for different load distributions in the Tsinghua EastSun cluster, and show that TRS outperforms Shis and random stealing in all test cases. Finally, TRS is implemented in the Jcluster environment [1], a high performance Java parallel environment, and an experiment result is given on HKU Gideon 300 cluster.

In the rest of this paper, we first give the transitive random stealing algorithm in next section. Section 3 evaluates the performance of TRS, Shis and random stealing by the random baseline technique. We show an experiment result on HKU Gideon 300 cluster in Jcluster environment in Section 4. Finally, Section 5 concludes our works.

2 Design the Transitive Random Stealing Algorithm

Our design philosophy for scheduling algorithms is to reduce the idle time for all nodes, rather than balancing work loads equally on all nodes. A node is said to

be in the idle state when it has no tasks to execute. Distributing the workload during application execution is achieved by sending the *tokens* to the schedulers on remote nodes. A token contains all the necessary information to create a new *task*. A Task is a piece of code that is to executed, possibly in parallel with other tasks. Tokens are stored in the task queue on each node.

In the following, we give the transitive random stealing algorithm in detail. First, we show you a figure to illustrate an architecture of a task scheduler based on TRS.

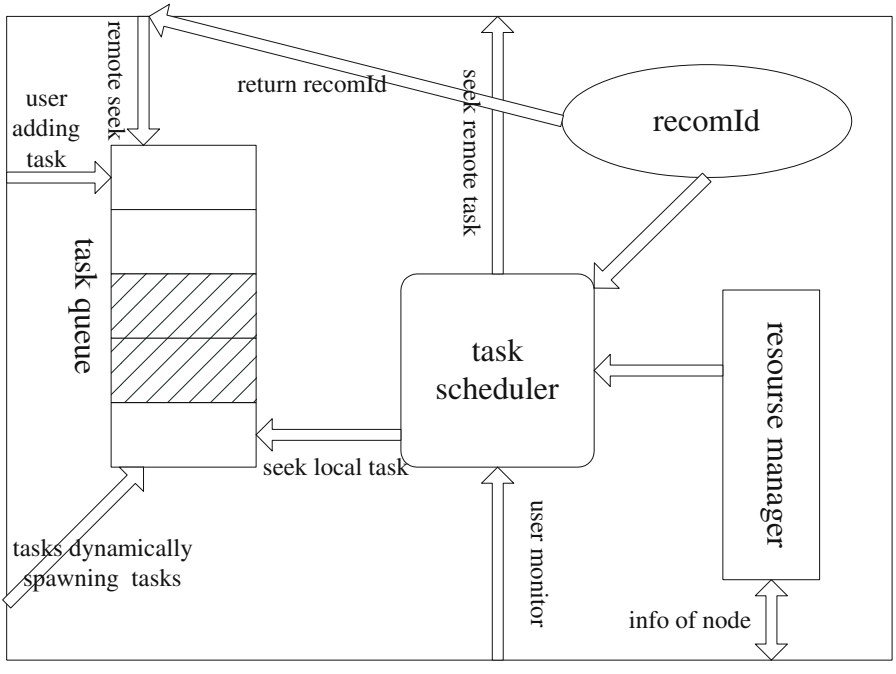

Fig. 1. An architecture of a task scheduler based on TRS

Here resource manager is responsible for adding or deleting nodes and maintains an active list of nodes in the cluster. Task queue is a double-ended queue to store tokens that have been spawned dynamically by tasks or have been added by user, but not yet executed. New tokens spawned dynamically by tasks are pushed into the queue from one end and tokens are also popped from the same end for execution on the local node. On the other hand, new tokens added by user are pushed into the queue from the other end, and a token is also popped from the other end of the task queue when remote nodes asks for tasks. The recomId is a variable which remembers the nodeId of other remote node.

The transitive policy is simple and TRS can be easily implemented. But with this simple transitive policy, TRS can make any idle node obtain a task from

```
The main-loop function for stealing tasks from another nodes:
void run(){
  While(true){
    if (idle of node){
      if (local task queue has tokens){
        get a token to execute;
      }else{
        if (recomId is blank){
          randomly select a remote node from the list of nodes,
          and ask for a token from it;
        }else{
          ask for a token from the remote node whose nodeId is recomId;
        }
        wait to receive an answer message;
        update its recomId with the recomId in the answer message;
        if (the answer message includes a token){
          execute the token;
        }
      }
    }else{
      wait for some task running over;;
    }
  }
}
The function for answering the request of another nodes:
Message answer(){
  if (local task queue has tokens){
    return a message with its own nodeId as recomId and a token
    from local task queue;
  }else{
    return a message with its recomId and no tokens;
  }
}
```

Pseudo code of the transitive random stealing algorithm

another node with less times to steal in a large-scale cluster. As a result, this will greatly reduce the idle time for all nodes and improve the scalability of the system. At the same time, TRS inherits the advantages of simple random stealing policy: communication is only initiated when nodes are idle. When the system load is high, no communication is needed, causing the system behave well under high loads.

As we can see, a few more bytes (recomId) is sent in the replying message for TRS than Shis and RS. But the time and bandwidth of the communication are very similar for those messages with little different sizes. In a sense, the key factor which influences the network communication overhead is the times of sending messages.

Note. In some very special conditions, there may be a loop transition of the recomId. In order to avoid this case, the implementation of the algorithm can limit the times of transition of the recomId. In fact, in the later experiments, we empirically limit the times of transition of recomId by $\max\{[\log_2 n - 3], 1\}$, where n is the number of the nodes in the cluster.

3 Performance Evaluation Based on Random Baseline Technique

In this section, by the random baseline technique, we experimentally compare TRS with Shis, one of load balance policies in the EARTH system, and random stealing for different load distributions on the Tsinghua EastSun cluster which has 32 nodes (4×Xeon III 700s, Fast Ethernet, Redhat 8.1). Here we implement each of the three algorithm as an MPI application in which a process simulates a node. The processes implement two threads except the process with rank 0, one thread for dealing the main loop, the other for handling the request. The process with rank 0, by the random baseline technique, implements a task generator which distributes the same load distributions to the other processes for the three algorithms respectively.

In order to stress to test the performance of algorithms on the different load distributions, we make use of the task generator generating different load distributions instead of scheduling some real parallel programs. The task generator generates three types of load distributions uniformly distributed on all

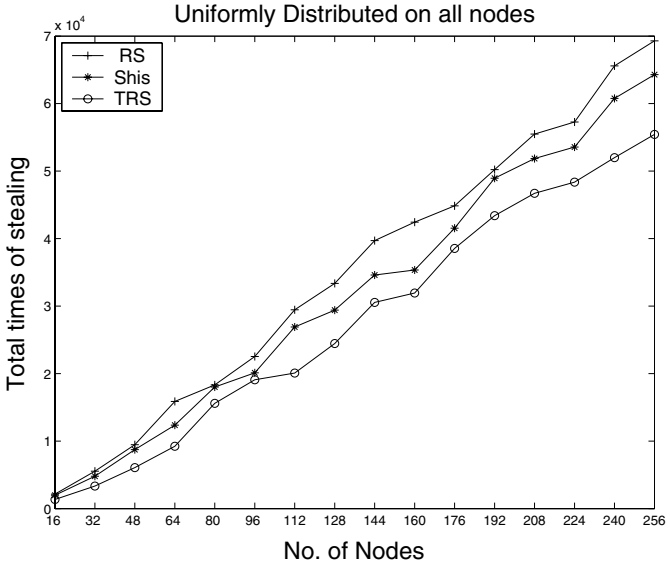

Fig. 2. Task load uniformly distributed on all nodes

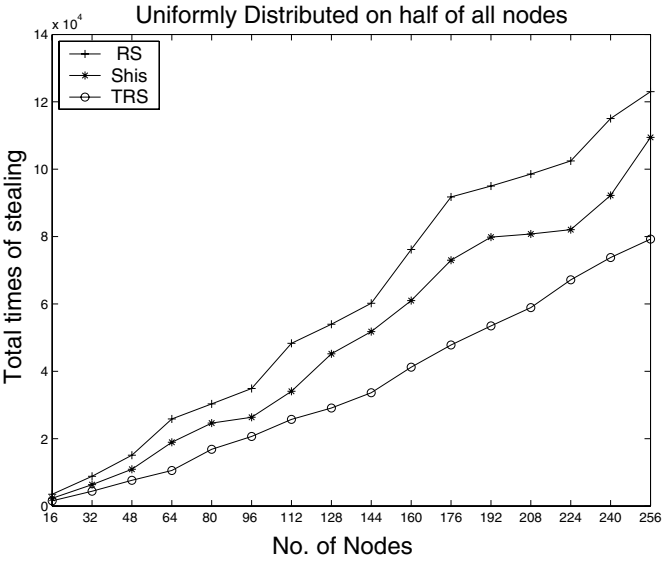

Fig. 3. Task load uniformly distributed on half of all nodes

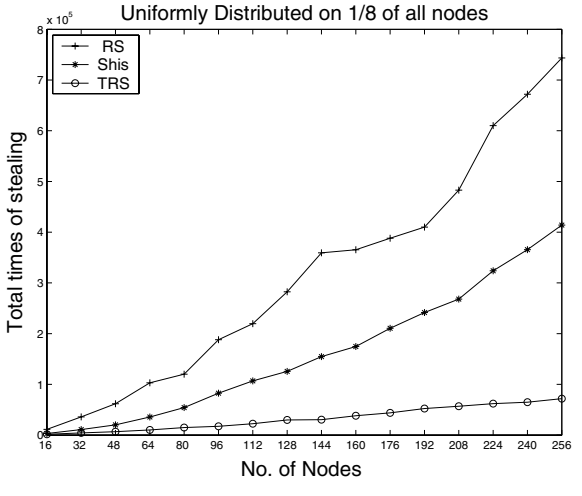

Fig. 4. Task load uniformly distributed on 1/8 of all nodes

nodes, half of all nodes and 1/8 of all nodes, two types of binomial distributions, Bi$(n, 1/3)$ and Bi$(n, 1/8)$, where n is the number of the nodes. From the knowledge of Statistics, the binomial distribution Bi(n, p) approaches the Poisson distribution, when the number n is large, and the probability p is small. The five types of load distributions all distribute $5n$ tasks to the nodes for 10 times,

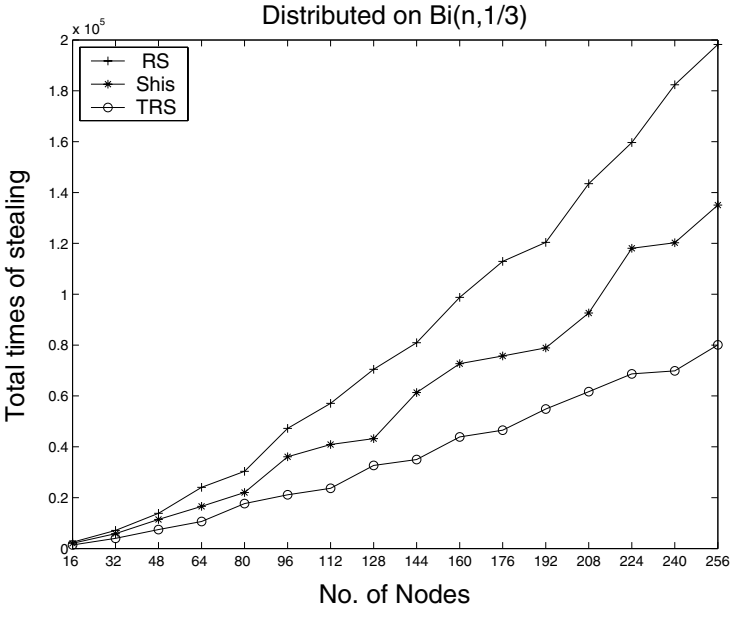

Fig. 5. Task load distributed on Bi$(n, 1/3)$

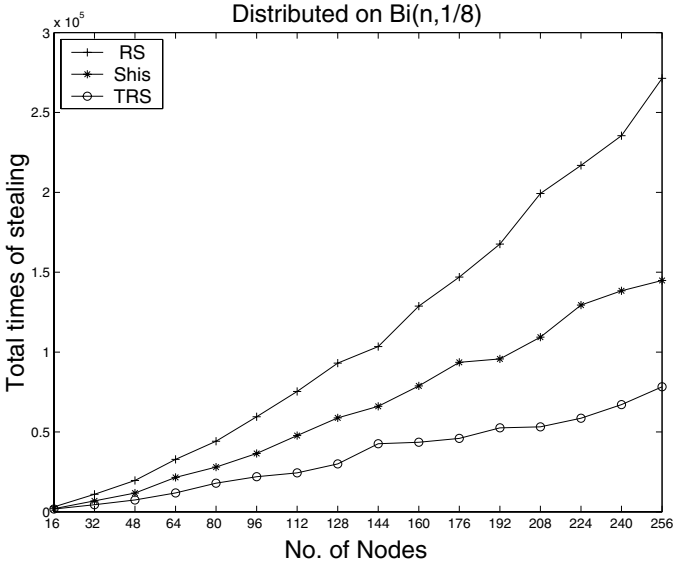

Fig. 6. Task load distributed on Bi$(n, 1/8)$

where n is the number of the nodes. In addition, we assume that every task has the same executing time and every node has the same power of computing.

For obtaining a good performance, the algorithm must make any idle node obtain a task faster with less times to steal. Therefore, we compare the performance of the three algorithms by counting the total number of stealing tasks from remote nodes for each algorithm (the total number includes the times of stealing nothing from remote nodes). The experiments are implemented in the Jcluster environment, a high performance Java parallel environment which provides MPI-like message passing interface on the Tsinghua EastSun cluster. Figure 2,3,4,5,6 illustrate the results for the five type of load distribution.

For the task load distribution uniformly distributed on all nodes, the difference of the performance for the three algorithms is small on the small-scale clusters, however, with the increase of the size of the nodes, TRS behaves with the good performance. For the task load distributions uniformly distributed on half of all nodes and on 1/8 of all nodes, binomial distributions, Bi$(n, 1/3)$ and Bi$(n, 1/8)$, TRS exhibits a much better performance than Shis and random stealing, especially, for the large-scale clusters. Therefore, we can conclude that TRS is a high performance scheduling algorithm for irregular load distributions in a large-scale cluster.

4 An Experiment Result in the Jcluster Environment

Jcluster environment [1] that provides a high performance PVM-like and MPI-like message passing interface implements the TRS algorithm to schedule the

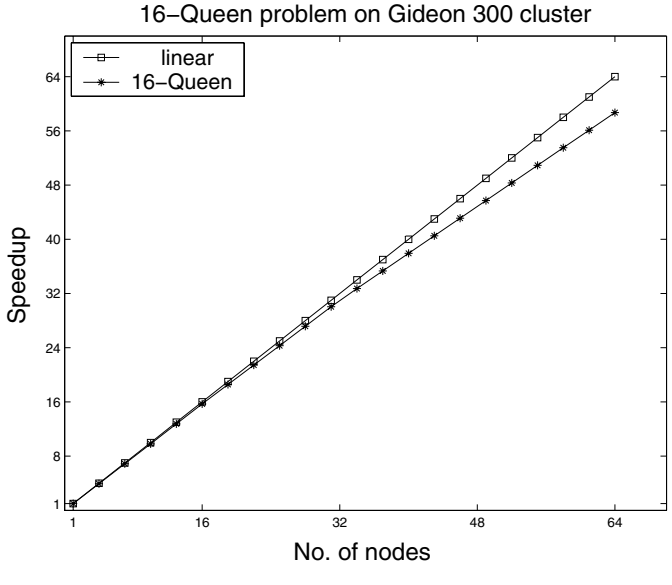

Fig. 7. 16-Queen problem on HKU Gideon 300 cluster

tasks dynamically in a large-scale cluster. Here a divide-and-conquer program, 16-Queen problem, is used to stress to test the task scheduler based on TRS. There are more than 2,200 subtasks which will be dynamically spawned on some nodes to be scheduled. With the help of Prof. Francis C.M. Lau, Prof. C.L. Wang and Weijian Fang, the test for 16-Queen problem has been held on HKU Gideon 300 cluster (Pentium IV 2.0 GHz, Fast Ethernet, redhat 8.0, Jdk 1.4.0) at the University of Hong Kong. Figure 7 illustrates the results.

The efficiency of the speedup reaches up to 91.73% on 64 nodes, which exhibits an efficient scheduling of TRS on the real platform.

5 Conclusion and Further Works

In this paper, we present the Transitive Random Stealing algorithm (TRS) which provides an efficient scheduling policy making any idle node rapidly obtain a task from other remote node for irregular load distributions in a large-scale cluster. Then by the random baseline technique, we experimentally compare TRS with Shis, one of load balance policies in the EARTH system, and random stealing for different load distributions on the Tsinghua EastSun cluster and conclude that TRS is a highly efficient scheduling algorithm for irregular load distributions in a large-scale cluster. Finally, Jcluster environment implements a task scheduler based on TRS to obtain a good experiment result for 16-Queen problem on HKU Gideon 300 cluster. In the future, more real parallel applications will be developed to evaluate the algorithm on some real platforms.

Acknowledgements

We are very grateful to Prof. Francis C.M. Lau, Prof. C.L. Wang and Weijian Fang in the University of Hong Kong for their warmhearted help.

References

1. http://vip.6to23.com/jcluster/
2. Berenbrink, P., Friedetzky, T., Goldberg, L.A., "The Natural Work-Stealing Algorithm is Stable", SIAM Journal on Computing, Vol. 32(5), pp. 1260-1279, 2003.
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., and Zhou, Y., "Cilk: An efficient multithreaded runtime system", Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95, Santa Barbara, California, pp. 207-216, July 1995.
4. Blumofe, R.D., and Leiserson, C.E., "Scheduling Multithreaded Computations by Work Stealing", Proceedings of the 35th Annual IEEE conference on Foundations of Computer Science (FOCS'94), Santa Fe, New Mexico, November 20-22, 1994.
5. Cai, H., Olivier Maquelin, Prasad Kakulavarapu, and Gao, G.R., "Design and Evaluation of Dynamic Load Balancing Schemes under a Fine-grain Multithreaded Execution Model", Proc. of the Multithreaded Execution Architecture and Compilation Workshop, Orlando, Florida, January 1999. Delaware, May 1999.

6. Eager, D.L., Lazowska, E.D., and Zahorjan, J., "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing", Performance Evaluation, Vol. 6, pp. 53-68, 1986.
7. Herbert H.J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Lau-rie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. "A design study of the EARTH multiprocessor", Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95 (Lubomir Bic, Wim Bohm, Paraskevas Evripidou, and Jean-Luc Gaudiot, eds.), Limassol, Cyprus, ACM Press, pp. 59-68, June 27-29, 1995.
8. Mao, Z.M., So, H.S.W., Woo, A., "JAWS: A Java Work Stealing Scheduler Over a Network of Workstations", Technical report, The University of California at Berkeley, June 1998.
9. Rob V. van Nieuwpoort, Kielmann, T., and Bal, H., "Satin: Efficient Parallel Divide and Conquer in Java", Proc. Euro-Par 2000, Munich, Germany, pp. 690-699, Aug. 29-Sep. 1, 2000.
10. Sanders, P., "Randomized receiver initiated load balancing algorithms for tree shaped computations", The Computer Journal, Vol. 45(5), pp. 561-573, 2002.
11. Shivaratri, N.G., and Krueger, P., "Two Adaptive Location Policies for Global Scheduling Algorithms", IEEE International Conference on Distributed Computing Systems, 1990.
12. Shivaratri, N.G., Krueger, P., and Ginghal, M., "Load Distributing for Locally Distributed Systems", IEEE Computer, Vol 25(12), pp. 33-44, Dec. 1992.
13. Wu, I.C., and Kung, H., "Communication Complexity for Parallel Divide and Conquer", 32nd Annual Symposium on Foundations of Computer Science (FOCS'91), San Juan, Puerto Rico, pp. 151-162, Oct. 1991.