An Adaptive Globally-Synchronizing Clock Algorithm and its Implementation on a Myrinet-based PC Cluster

Cheng Liao **Margaret Martonosi** **Douglas W. Clark**
Depts. of Computer Science and Electrical Engineering
Princeton University

Abstract

Fast commodity network-connected PC or workstation clusters are becoming more and more popular. This popularity can be attributed to their ability to provide high-performance parallel computing on a relatively inexpensive platform. An accurate global clock is invaluable for these systems, both for measuring network performance and coordinating distributed applications. Typically, however, these systems do not include dedicated clock synchronization support. Previous clock synchronization methods are not fully suitable to them, either due to their non-commodity hardware requirements or due to insufficient synchronized clock accuracy.

In this paper we present and evaluate an adaptive clock synchronization algorithm. We have implemented and tested the algorithm on a Myrinet-based PC cluster. The algorithm has several important features. First, it does not require any extra hardware support. Second, we show that this algorithm places very low intrusion on the system and has a microsecond-level accuracy. Finally, our results indicate that adding the ability to adaptively adjust the clock's re-synchronization period causes almost no extra overhead while achieving a much better global clock accuracy.

1 Introduction

Performance monitoring is a crucial aspect of parallel programming. Monitoring tools are often essential to extract the best performance from the system. As part of this, accurate latency measurements are key to understanding program performance, and globally-synchronized clocks are the main requirement for getting them. Furthermore, synchronized clocks also allow fine-grained real-time coordination of tasks distributed across several nodes.

Recently, fast, commodity, network-connected clusters of PCs or workstations have become a widespread parallel computing platform. Unfortunately, globally-synchronized clocks are rarely available in these loosely-coupled distributed systems. Although clocks are available in each node, they are not synchronized, or even aware of clocks on other nodes. Without an accurate common time base, it is impossible to measure inter-node latencies.

The problem of clock synchronization has been studied frequently in distributed systems. However, they either require extra hardware support, or they lack the fine clock accuracy necessary in system-area network monitoring. In this paper, we describe a clock synchronization algorithm based on Cristian's algorithm [1]. Our algorithm is implemented in the firmware running on the programmable Myrinet network interfaces. Our approach does not require any extra hardware, and imposes an extremely low overhead on the system. If synchronized clocks get out of synchrony due to fluctuating clock drift or to small inter-node clock frequency differences, our algorithm will dynamically and robustly adjust the resynchronization period based on the current system situation. As a result, our globally-synchronized clock can maintain microsecond-level accuracy.

SIGMETRICS '99 5/99 Atlanta, Georgia, USA

2 Design and Implementation Issues

Our clock synchronization algorithm is most closely related to Cristian's algorithm [1]. However, we do not need extra hardware support. Periodically, Node 0 re-synchronizes the clock on other nodes to its own clock by exchanging *MyriTime* and *MyriTimeDiff* packets. To prevent the inaccuracy caused by a burst of incoming synchronization requests, the time server starts the synchronization instead of having each node initiate contact with the time server.

Since we need to achieve microsecond-level accuracy in the globally-synchronized clock, the resynchronization period must be fairly short, less than $100ms$ according to our experiments. This high re-synchronization frequency will impose a heavy load on the system. We note from the experiments that the drift rates of different node clocks are not equal. However, each individual drift rate is stable, at least within a reasonably long period of say 5 to 10 minutes. Taking advantage of this observation, we modified the algorithm so that the global procedure that exchanges MyriTime packets only happens infrequently. We call this kind of resynchronization *global-resynchronization*. Meanwhile, each node will locally adjust the clock itself by extrapolating the current time difference value from the drift rate it saw before. We call this *self-resynchronization*.

We also include some adaptiveness into our algorithm so that the global-resynchronization period and self-resynchronization period can be adjusted on-the-fly based on the current system situation. Figure 1 shows the pseudo code running on Node 0 and all the other nodes in the system. T_{net} is the time a MyriTime packet spends on the network, and T_{diff} is the clock difference between the local node and Node 0. When a node other than Node 0 receives a MyriTimeDiff packet, it will compare the T_{diff} in the packet with the T_{diff} extrapolated locally. This node will then send back to Node 0 a MyriTimeDiff packet if the difference is over a threshold. Node 0 will decide whether to shorten the global-resynchronization period based on this information. When Node 0 does not get any MyriTimeDiff packets for a long time, it will lengthen the global-resynchronization period. The self-resynchronization period is adjusted on each node to make sure the self-resynchronization is done frequently enough to keep the synchronized clock accurate.

3 Accuracy of Clock Synchronization Algorithm

Table 1 presents the error and standard deviation of the global clock using our synchronization algorithm with and without adaptive adjustment of resynchronization periods, in either idle or heavily-loaded situations. When the system is idle, the adaptive algorithm performs only slightly better than the non-adaptive algorithm. When the system is heavily loaded, however, the adaptive algorithm works nearly twice as well as the non-adaptive algorithm.

We also studied the robustness of our algorithm to less predictable clock drifts. We accomplish this by artificially introducing particular drift rates into the clock on some nodes. We then measure the response of the synchronized clock to these different drift rates. The experiments show that both adaptive and non-adaptive synchronization algorithms will adjust to the changed drift rate. However, the adaptive algorithm is able to follow the system changes

On Node 0

```
if (nodeX need to be resynchronized)
   for (i = 0; i < 3; i ++)
       send(nodeX, MyriTime)
       recv(nodeX, MyriTime)
       update shortest_T_net_this_round and
              corresponding T_diff if necessary
   if (shortest_T_net_this_round - shortest_T_net_ever >
       NETWORK_LATENCY_ERROR_THRESH)
      for (i = 0; i < 3; i ++)
          send(nodeX, MyriTime)
          recv(nodeX, MyriTime)
          update shortest_T_net_this_round and
                 corresponding T_diff if necessary
   send(nodeX, MyriTimeDiff, T_diff)
   update shortest_T_net_ever
   if (recv(nodeX, MyriTimeDiff, &err))
      error[nodeX] += err
      if (error[nodeX] > ERROR_THRESH)
         if (global_resync_period[nodeX] != MIN_GLOBAL_PERIOD)
            global_resync_period[nodeX] >>= 1
         error[nodeX] = 0
   else if (++ counter_of_success[nodeX] == SUCCESS_THRESH)
          if (global_resync_period[nodeX] != MAX_GLOBAL_PERIOD)
             global_resync_period[nodeX] <<= 1
          counter_of_success[nodeX] = 0
```

On other Nodes

```
recv(0, MyriTime)
send(0, MyriTime)
recv(0, MyriTimeDiff)
err = abs(T_diff_from_Node_0 - T_diff_projected)
if (err > DIFFERENCE_THRESH)
   send(0, MyriTimeDiff, err / DIFFERENCE_THRESH)
calculate drift rate
if (drift rate is too large &&
    self_resync_period > MIN_SELF_SYNC_PERIOD)
   reduce self_resync_period
apply drift rate to T_diff every self_resync_period
```

Figure 1: Pseudo code of the synchronization algorithm running on Node 0 and all other nodes.

	System Idle Err (μs)	System Idle Std (μs)	System Loaded Err (μs)	System Loaded Std (μs)
Non-Adaptive	0.80	1.00	2.54	2.83
Adaptive	0.65	0.95	1.26	1.45

Table 1: Clock error and standard deviation of non-adaptive and adaptive synchronization algorithms: idle and heavily-loaded cases.

more quickly, as it can reduce the global-resynchronization period accordingly. This situation is much more clear in heavily-loaded systems.

Figure 2 shows the actual network latencies for different packet sizes collected on Node 1 for the shared virtual memory (SVM) application Radix. Figure 3 is similar except this time we introduce an artificial drift rate of $1/2^{18}$ to all nodes except Node 1. It is clear that the two curves using the adaptive algorithm retain their good accuracy from Figure 2 for the most part, while the two curves for the non-adaptive algorithm have deviated sharply, especially when the system is heavily loaded.

To summarize, on our normal system, both non-adaptive and adaptive algorithms can achieve microsecond-level accuracy in a globally-synchronized clock. However, the accuracy of the non-adaptive algorithm will drop dramatically as the load on the system or the fluctuation of the clocks increases. Although the accuracy of the adaptive algorithm also drops as system load or clock drift fluctuations increase, it deteriorates much slower than in the non-adaptive case.

4 Perturbation due to Clock Synchronization

To quantify the system perturbation due to our clock synchronization algorithm, we first consider the impact of our clock synchronization code on the overall behavior of real applications. To address this, we have run several SVM applications with and without clock synchronization code running. The results show that synchronizing overhead measured at the program level is almost negligible: less than 0.3% in all applications.

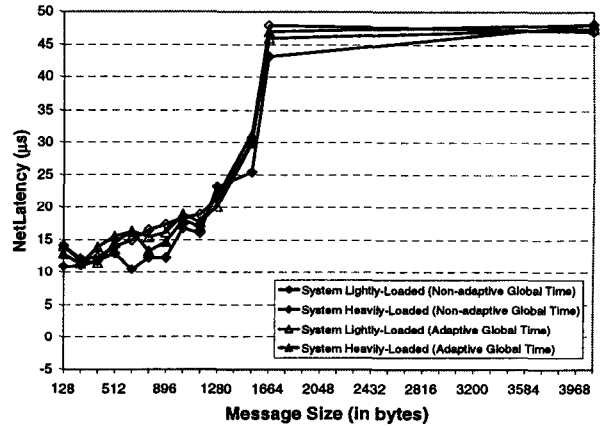

Figure 2: Network latencies collected on Node 1 for Radix.

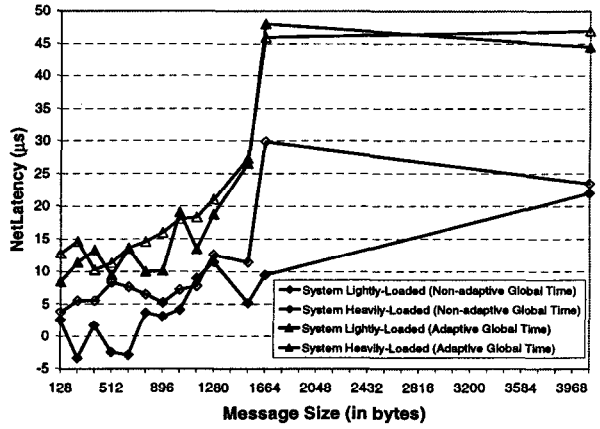

Figure 3: Network latencies collected on Node 1 for Radix with a $1/2^{18}$ introduced artificial drift rate.

We also ran microbenchmark experiments to isolate particular clock perturbation aspects, including latencies and bandwidths. The results show that the overhead due to clock synchronization is also very small: less than 3% in most cases.

To summarize, the global clock synchronization code has very little perturbation on the system, both in real applications and microbenchmark level. In addition, the adaptive portion of the method has a negligible impact on the system.

5 Conclusions

We have presented an algorithm to maintain a globally-synchronized clock without extra hardware support. It works on loosely-coupled parallel systems which do not have a globally-synchronized clock, although each node does have an unsynchronized local clock. Our algorithm is focused on keeping the globally-synchronized clock as accurate as possible while imposing only a low perturbation on the system. This work offers a cheap and portable way to meet the strict accuracy requirements that performance monitoring tools impose for globally-synchronized clocks.

We have implemented our synchronization algorithm on a Myrinet-based system. The results show that our algorithm achieves microsecond-level accuracy while keeping a very low perturbation on the system. Furthermore, adding the ability to adaptively adjust the re-synchronization period has almost no extra overhead, while it facilitates better accuracy in the globally-synchronized clock. This is especially true in situations where the system is heavily loaded or the clock drift rates of some nodes are fluctuating. In general, we believe that our algorithm is suitable for and fairly easy to implement in current network-connected clusters.

References

[1] F. Cristian. Probabilistic Clock Synchronization. *Distributed Computing*, vol 3:146–158, 1989.