**ORNL/TM-11744**

Engineering Physics and Mathematics Division

Mathematical Sciences Section

# HYPERCUBE CLOCK SYNCHRONIZATION

Thomas H. Dunigan

Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367
**thd@ornl.gov**

Date Published: September 1994

Research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC-05-84OR21400

## Contents

# HYPERCUBE CLOCK SYNCHRONIZATION

Thomas H. Dunigan

**Abstract**

Algorithms for synchronizing the times and frequencies of the clocks of Intel and Ncube hypercube multiprocessors are presented. Bounds for the error in estimating clock offsets and frequencies are formulated in terms of the clock read error and message transmission time. Clock and communication performance of the Ncube and Intel hypercubes are analyzed, and performance of the synchronization algorithms is presented.

Keywords: **clock synchronization, hypercube communication**.

## 1. Introduction

In distributed computing, there is a need for accurate clocks that give the same time on every computing element. A distributed computation may be implemented over a local area network of computing elements or on a parallel processor. Distributed computing systems that depend on such a common time include transaction-processing systems, real-time systems, and simulation systems. A common time is also needed for event traces and synchronization such as time-outs or checkpoints.

This report addresses synchronizing the clocks in hypercube parallel processors, specifically to eliminate timing anomalies in event trace files. If the clocks are not synchronized, timing anomalies can occur, such as the apparent receipt of a message before it was sent. In the following section, a model for synchronizing clocks in a distributed computing environment is developed. In section 3, the characteristics of the clocks and message-passing systems of the Ncube and Intel hypercubes are analyzed, and clock synchronization algorithms are developed. Section 4 discusses the performance of the hypercube clock synchronization algorithms and the need for hardware clock synchronization.

## 2. Clock Synchronization

We wish to synchronize the clocks in a distributed computing system so we can order events. In particular, if $C_i[a]$ is the time at which processor $i$ sends a message to processor $j$, and $C_j[b]$ is the time processor $j$ receives the message, we want $C_i[a] < C_j[b]$. This is Lamport's condition C2 [7]. To avoid timing anomalies of events on a single processor, we further require that time be monotonically increasing, that is, $C_i(t_2) > C_i(t_2)$, if $t_2 > t_1$ where $C_i(t)$ is time from processor $i$'s clock at physical time $t$.

To develop a light-weight clock synchronization algorithm that a hypercube application program can implement, we assume that there are no faulty clocks in our ensemble of processors. Synchronization techniques in the presence of faulty clocks are discussed in [13] and [11]. A robust and successful implementation of clock synchronization over a network is described in [8] and [9]. This report develops software synchronization algorithms, though we later argue for a single clock or hardware synchronization of the ensemble of clocks. Hardware synchronization, however, is non-trivial [12], [14], [10].

The *offset*, $\theta_{ij}(t)$, is the difference in the time reported by processor $j$'s clock to that of processor $i$'s clock at some physical time, $t$, that is,

$$\theta_{ij}(t) \equiv C_j(t) - C_i(t).$$

The *skew*, $\phi_{ij}$, between two clocks is their difference in frequency (first derivative of offset with time), measured as the change in the offsets of the clocks over an interval $S$, where the skew of clock $i$ with respect to clock $j$ is

$$\phi_{ij} \equiv \frac{\theta_{ij}(t+S) - \theta_{ij}(t)}{C_i(t+S) - C_i(t)}.$$

Computer clocks drift only a few seconds per day, so typically, $\phi < 10^{-5}$. Computer clocks exhibit some variation in skew (second derivative of offset with time) due to aging or temperature variation, but the effect is small (less than a few parts per million (ppm)) and is ignored in this study. Table 2.1 summarizes the notation used in this report.

| **Notation** | |
|---|---|
| $C_i(t)$ | clock time at processor $i$ at real time $t$ |
| $\theta$ | clock offset |
| $\delta$ | round-trip message time |
| $\phi$ | frequency difference (skew) |
| $\rho$ | clock precision |
| $R$ | resynchronization interval |
| $S$ | skew estimation interval |
| $U$ | minimum message transmission time |

**Table 2.1:** *Notation conventions.*

To avoid timing anomalies in sending and receiving messages between two processors, the sum of the offset and skew of the transmitter's clock must be less than $U_0$ (the minimum message transmission time), that is $\theta + U_0\phi < U_0$, or

$$\frac{\theta}{1-\phi} < U_0.$$

$U_0$ is usually the time to send a zero-length message to an adjacent processor. We will have timing anomalies if the two clocks are not within $U_0$ of each other, and, even if they are initially that close, after some period of time the two clocks will have drifted greater than $U_0$ apart because of the difference in the clock frequencies. Thus, to synchronize clocks, we must synchronize both time (eliminate or reduce the offset) and frequency (make both clocks run at the same rate).

**Time synchronization.**

Processor $i$ can synchronize its time with processor $j$ by sending a message to processor $j$ at time $C_i(t_i)$ requesting that processor $j$ send its current time back in a message (Figure 2.1). If processor $i$ receives processor $j$'s reply at time $C_i(t_\theta)$ then processor $i$ can measure the round-trip time, $\delta$, that it took to send the message and receive the reply where

$$\delta = C_i(t_\theta) - C_i(t_i).$$

With the reply and the round-trip time, processor $i$ can estimate the offset of processor $j$'s clock with

$$\hat{\theta}_{ij}(t) = C_j(t_j) + \delta/2 - C_i(t_\theta).$$

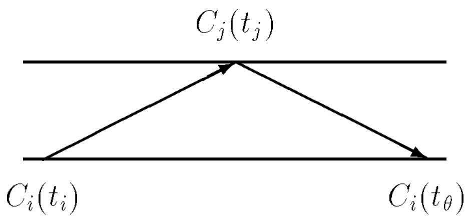


**Figure 2.1:** *Processor $i$ requests processor $j$'s time.*

In most message-passing systems, the round-trip time shows some variance and results in some uncertainty or error in our offset estimate. We can usually measure or calculate what the minimum transmission time is for a time request or reply message between two processors. The minimum transmission time, $U_x$, is usually a function of message size and distance (or number of hops in a hypercube or mesh). Thus processor $i$ only knows that $C_j(t_j)$ lies in the interval $\beta - \alpha$ (Figure 2.2). Choosing the midpoint of the interval, $\delta/2$, gives an unbiased estimate of when processor $i$ thinks processor $j$ sent the time reply. If the two clocks run at the same frequency, then the error in our offset estimate is at most $\delta/2 - U_x$. If the skew between the two clocks is $\phi_{ij}$, the slope of the line in Figure 2.2, then our offset estimate could be off by an additional $(\delta/2 - U_x)\phi_{ij}$. Finally, our offset estimate can be no more accurate the the precision of the clocks, $\rho$. Thus we can estimate the offset only to within some error, $\epsilon_\theta$, where

$$\epsilon_\theta < (\delta/2 - U_x)(1 + \phi_{ij}) + \rho. \tag{2.1}$$

In section 3, we will quantify the error of our offset estimator and develop filters to reduce those bounds.

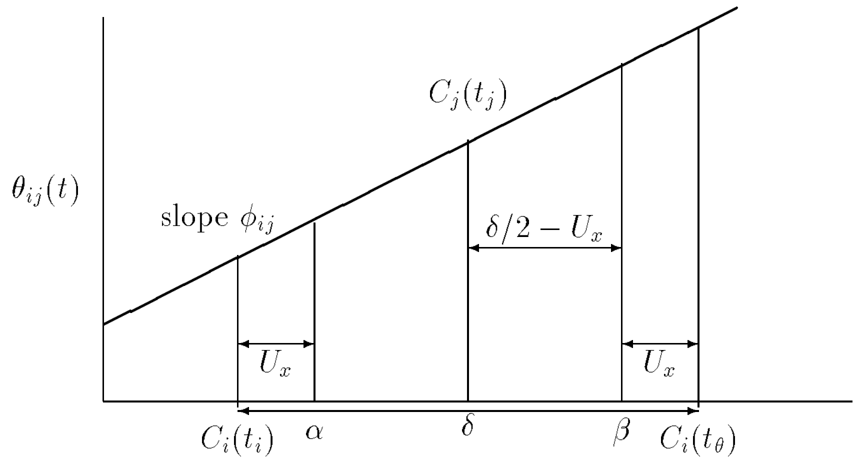


**Figure 2.2:** *Offset error due to random delay and skew.*

**Frequency synchronization.**

Even if we can make the offsets suitably small, after $U_0/\phi_{ij}$ seconds, the two clocks would again be greater than $U_0$ seconds apart, resulting in potential timing anomalies. Thus the clocks would have to be resynchronized every $U_0/\phi_{ij}$ seconds, or, more conservatively, every $(U_0 - \epsilon_\theta)/\phi_{ij}$ seconds. We can reduce this resynchronization interval by measuring the skew between the two clocks and using this estimate to "adjust" the frequency of one of the clocks. Processor $i$ can estimate the difference in frequency by taking the ratio of the differences of the two clocks over a period $S$, where

$$\hat{\phi}_{ij} = \frac{\hat{\theta}_{ij}(t+S) - \hat{\theta}_{ij}(t)}{S}. \tag{2.2}$$

The error in the estimate is

$$\epsilon_\phi < \frac{2\epsilon_\theta}{S}$$

so $\phi_{ij}$ can be estimated to the desired accuracy by increasing the interval $S$. Since we assume that the frequency difference between two clocks is constant, we need calculate $\phi_{ij}$ only once.

Having calculated both the time and frequency estimates, then the adjusted clock on processor $i$ that is synchronized to the clock on processor $j$ is

$$T_i(t) = C_i(t) + \hat{\theta}_{ij} + \hat{\phi}_{ij}(t - t_\theta), \tag{2.3}$$

and the resynchronization interval is then

$$R = \frac{U_0 - \epsilon_\theta}{\epsilon_\phi}. \tag{2.4}$$

In the following section, we will apply this clock synchronization model to hypercube parallel processors.

## 3. Hypercube clocks and communication.

A hypercube parallel processor is an ensemble of computers interconnected by a communication network with the topology of an $n$-dimensional hypercube. Each processor has its own local memory, clock, and communication channels to $n$ other processors. The processors work concurrently on an application and coordinate their computation by passing messages. First-generation hypercubes used a store-and-forward scheme to route messages between non-adjacent processors. Present generation hypercubes use hardware-assisted direct routing, and message-passing times are on the order of hundreds of microseconds [3]. By contrast, message-passing times on local area networks are on the order of tens of milliseconds.

| **Hypercube Synchronization Parameters** | | | | |
|---|---|---|---|---|
| | iPSC/860 | iPSC/2 | N6400 | N3200 |
| Crystal frequency (MHz) | 40 | 20 | 80 | 32 |
| Measured frequency difference ($\phi$) (ppm) | 80 | 25 | 0 | 0 |
| Clock precision ($\mu$s) | 0.1 | 1.0 | 0.05 | 128 |
| Effective clock precision ($\rho$) ($\mu$s) | 1 | 15 | 1 | 128 |
| Clock read error ($\mu$s) | 60 @ 10ms | 260 @ 50ms | 30 @ 20ms | 128 |
| Round-trip error (d6) ($\mu$s) | 120 | 527 | 60 | 128 |
| Filtered round-trip error (d6) ($\mu$s) | 25 | 114 | 18 | 128 |
| Minimum message time ($U_0$) ($\mu$s) | 64 | 302 | 151 | 511 |
| Barrier exit error (d6) ($\mu$s) | 250 | 400 | 100 | 128 |
| Monotonic clock | yes | yes | yes | no |

**Table 3.1:** *Intel and Ncube clock and communication parameters.*

Hypercube processor clocks may be driven by a single time source, or each processor may have its own time source. The Ncube hypercubes use a single crystal oscillator to provide timing for all of the processors, and thus clock synchronization is fairly simple. The Intel hypercubes use a separate crystal on each processor, and the communications hardware on each processor has its own crystal. Clock synchronization for the Intel hypercubes requires both time and frequency synchronization. Table 3.1 summarizes the clock and communication characteristics of both families, and details of the Intel and Ncube characteristics are discussed in the next two sections.

### 3.1. Intel hypercubes.

#### Clocks

The Intel iPSC/860 hypercube uses 40 MHz oscillators on each processor. The crystal oscillator, manufactured by MF Electronics, has a frequency stability of 0.01% (200 ppm) and a variation of frequency less than $\pm 5$ ppm with respect to temperature. We measured the frequency differences between the processors on three different iPSC/860 hypercubes and found the maximum frequency difference to be less than 100 ppm (ORNL 128 processors $<$ 80 ppm; NASA Langley 128 processors $<$ 60 ppm; ICASE 32 processors $<$ 40 ppm). The actual differences can change as processor boards are replaced or as temperature varies.

The iPSC/860 provides a 100 nanosecond timer that is readable with the *dclock()* function available under NX 3.2. The resolution between two successive calls to *dclock()* is about 1.5 microseconds (averaged over 2,000 samples) with little variance. However, a 40 microsecond spike occurs every 10 milliseconds from an operating system timer interrupt. When the iPSC/860 is "booted," all of the processor timers are reset simultaneously.

The Intel iPSC/2 hypercube uses a 20 MHz oscillator on each 80386 processor board. The oscillator operating characteristics are similar to those of the oscillator used on the iPSC/860. We measured a maximum frequency difference of less than 25 ppm on the 64-processor ORNL iPSC/2. Time is provided through two 16-bit, microsecond countdown timers and an operating system variable that is incremented by 50 milliseconds when the high-order counter interrupts.

On the iPSC/2, only one millisecond resolution is provided by the Intel *dclock()* function, and the time is not monotonic (successive calls will return the same time, since several calls can be made in less than a millisecond). However, the PICL library [5] provides a microsecond resolution time function, *clock0()* for the iPSC/2. Successive calls to *clock0()* take an average of 138 microseconds, with a 260 microsecond spike every 50 milliseconds as a result of the operating system timer interrupt. (The iPSC/2 hypercube operating system (NX 3.2) actually provides multitasking for an application on each processor of the hypercube, so, as with any time-sharing operating system, even larger variations between successive calls to *clock0()* are possible if multiple tasks are active.) Since the software that reads the countdown timers must avoid the roll-over intervals of the counters, the values returned by *clock0()* exhibit more variance than the iPSC/860 *dclock()* calls, reducing the effective resolution to about 15 microseconds. Since time on the iPSC/2 is maintained with an operating system variable, when the hypercube is "booted," this variable is reset only after the operating system is loaded on each processor. On a 64-processor iPSC/2, several seconds can pass before the last processor's operating system is loaded and the time reset. Thus, the iPSC/2 clocks will have offsets of several seconds as soon as they are started.

### Communication

Each iPSC/2 and iPSC/860 processor has a separate communication module that is clocked by its own 28 MHz crystal oscillator. Message transmission time varies linearly with message size [3]. Part of the startup cost for sending a message is the time for the processor and communication modules to synchronize. For messages between non-adjacent processors, additional time is required to synchronize the intervening communication modules. The minimum transmission time, $U_0$, (one hop, zero-length message) is 64 microseconds for the iPSC/860 and is 302 microseconds for the iPSC/2. The clock synchronization model developed in section 2 uses the round-trip delay to estimate offset between two processor's clocks. Figure 3.1 illustrates the variance in the round-trip delays for 2,000 samples using an 8-byte message, for adjacent iPSC/860 processors and processors that are six hops away. The figure shows the spikes when a timer interrupt occurs on one or both of the processors involved. As expected, the multi-hop delays are larger and show more variance than nearest neighbor communication. (iPSC/2 delays show somewhat less variance than the iPSC/860 delays due to the less precise clock and due to the slower communication.)

From equation (2.1), our offset estimate improves as the round-trip delay approaches the minimum round-trip delay. But the delay will be the minimum with only a small probability (Figure 3.2). We can be certain only that the delay will be less than the maximum round-trip delay. (We shall assume for now that the delay does have an upper bound.) If we wish to improve the error bound of our offset estimate, but at the risk of some uncertainty, we can choose a delay, $D$, less than the maximum. If $p$ is the probability that processor $i$ observes a round-trip delay greater than $D$, then the probability of observing a round-trip delay less than $D$ is $1 - p$. However, if we make $k$ successive clock requests, the probability of at least one delay less than $D$ increases to $1 - p^k$ [1]. Thus we can choose a level of certainty and estimate the offset between clocks with a small error bound at the cost of many messages or with a larger error bound but at the cost of a only a few messages. We can also improve the deterministic estimate

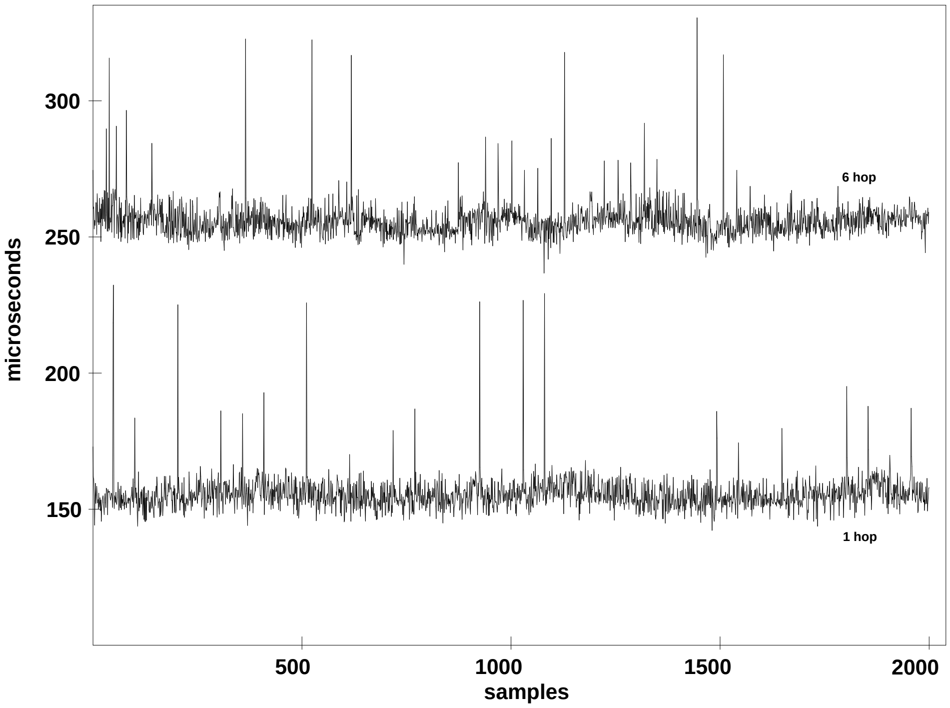


**Figure 3.1:** *8-byte message round-trip delay ($\mu s$), iPSC/860, 2000 samples.*

since some of the variation in round-trip delay is not random, but is caused by fixed-interval, timer-interrupt spikes. The interval between timer-interrupt spikes is much longer than the round-trip delay, thus we can eliminate the effect of these spikes by taking the minimum round-trip delay of at least three successive clock requests. Table 3.2 shows the reduction in round-trip delay variance with increasing $k$ using a minimum filter and 2,000 readings. For our implementation, we use four successive requests to estimate the offset between two clocks and use the resulting $\epsilon_\theta$ (Table 3.2) to select the interval, $S$, for estimating the frequency correction $\phi_{ij}$ (equation 2.2) to the desired precision. For example, to adjust the frequency to within an error of one part per million for our dimension-6 iPSC/860, $S$ (2.2) would need to be at least 26 seconds. Similarly, for our dimension-6 iPSC/2, the estimation interval would need to be at least 144 seconds.

Given $\epsilon_\phi$ and $\epsilon_\theta$ we can calculate the resynchronization time from equation (2.4). If the frequency adjustment is accurate to within one ppm, the iPSC/860 processor would have to be resynchronized in less than 51 seconds, and in less than 230 seconds for the iPSC/2.

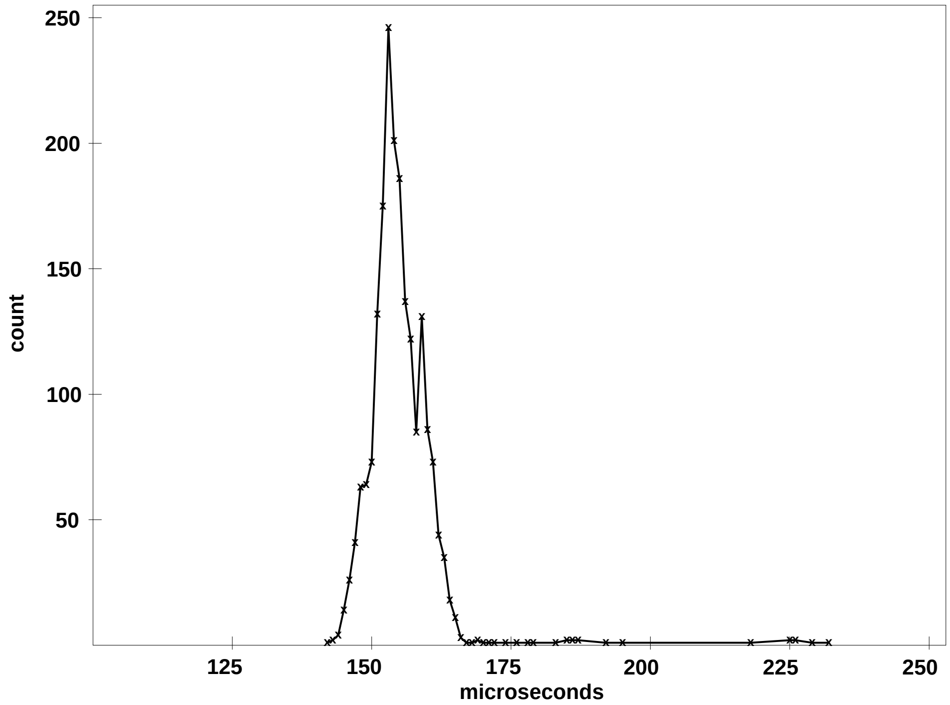


**Figure 3.2:** *One-hop round-trip delay distribution, iPSC/860, 2000 samples.*

### Implementation

Given the clock and communication characteristics, we can synchronize the time and frequency between two processors' clocks. For an application program to synchronize all of the processors' clocks in a dimension-$n$ hypercube, we use a master-slave model. We designate processor 0 as the master clock, and all of the remaining processors synchronize their clocks with processor 0. (For the present, we require only that the clocks be consistent within the hypercube; we are not trying to synchronize the clocks with an external source such as UTC

| | iPSC/860 | | | | | | iPSC/2 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 hop | | | 6 hops | | | 1 hop | | | 6 hops | | |
| $k$ | min | max | $\epsilon_\theta$ | min | max | $\epsilon_\theta$ | min | max | $\epsilon_\theta$ | min | max | $\epsilon_\theta$ |
| 1 | 147 | 235 | 45 | 246 | 339 | 47 | 704 | 985 | 155 | 797 | 1325 | 278 |
| 2 | 147 | 195 | 25 | 246 | 294 | 14 | 704 | 838 | 82 | 797 | 931 | 82 |
| 3 | 147 | 171 | 13 | 246 | 272 | 13 | 704 | 835 | 80 | 797 | 916 | 74 |
| 4 | 147 | 166 | 9 | 246 | 271 | 13 | 704 | 835 | 80 | 797 | 911 | 72 |
| 5 | 147 | 164 | 9 | 246 | 271 | 13 | 704 | 834 | 80 | 797 | 911 | 72 |
| 10 | 147 | 160 | 7 | 246 | 267 | 11 | 704 | 832 | 79 | 797 | 907 | 70 |

**Table 3.2:** *Round-trip delay ($\mu s$) with minimum of $k$ tries. (2,000 samples)*

[9].) Our clock synchronization algorithm consists of two subroutines, *clk_sync()* and *sclock()*. At the start of the application, each processor calls *clk_sync(S)* to synchronize the time and estimate the frequency correction over $S$ seconds. The application then uses *sclock()* to read the processor's synchronized clock, where the synchronized time is calculated from equation (2.3). On the Intel hypercubes, time resynchronization can be scheduled and managed in the "background" using *alarm(R)* and *signal(SIGALRM)* on the slave processors in conjunction with *hrecv()* on the master processor. Though the error in the offset and skew between a slave and the master is $\epsilon_\theta$ and $\epsilon_\phi$, the error in the offset between two slaves may be as much as $2\epsilon_\theta$ and $2\epsilon_\phi$. Thus the re-syncrhonization interval becomes

$$R = \frac{U_0 - 2\epsilon_\theta}{2\epsilon_\phi}.$$

These routines (less resynchronization) have been added by Worley to PICL [5] and have eliminated the timing anomalies previously experienced without clock synchronization. PICL had previously used a barrier-synchronization (implemented with a dimensional exchange) to synchronize time when event tracing was enabled. However, timing anomalies occurred frequently, since the variation in barrier exit (Table 3.2) was often larger than the minimum message transit time, and there was no compensation for clock skew.

### 3.2. Ncube hypercubes.

The Ncube 6400 hypercube (also known as the Ncube 2) uses one 80 MHz crystal oscillator to drive all of the processors and their associated clocks, count-down timers, and communication channels. Each processor provides a 50 nanosecond running timer accessible through the *atickcnt()* function under Vertex 2.0. Successive calls to *atickcnt()* require about 8 microseconds. There is a 30 microsecond spike from an operating system timer interrupt every 20 milliseconds. When the hypercube is rebooted all of the running timers are reset simultaneously. Thus all of the processors are synchronized with respect to time, and since there is only one oscillator, there is no frequency skew.

The older Ncube 3200 hypercube (previously called the Ncube/ten) uses one 8 MHz crystal oscillator to drive all of the processors. No running timer is provided, and each processor keeps time using an operating system variable and a count-down timer interrupt. When the Ncube 3200 is "booted," time is reset by the operating system after it is loaded, so the processor clocks will not be synchronized. Time (actually clock rate divided by 1024) is provided by the *ntime()* function, giving a resolution of 128 microseconds. Successive calls to *ntime()* can return the same value. Figure 3.3 illustrates how *ntime()* can be packaged so that it returns monotonically increasing values.

```
#define MICRO 0.000001
#define SPT 0.000128
static double bump = 0;
static int lastt= 0;
double sclock()
{
        /* return seconds, each call will return larger value */
        int t;

        t = ntime();  /* ticks */
        bump += MICRO;
        if (t != lastt){  /* rollover */
                bump = 0;
                lastt=t;
        }
        return(bump + t * SPT);
}
```

**Figure 3.3:** *Monotonic clock for the Ncube 3200.*

Since all Ncube processors and communication channels are driven by the same timing signal, there is much less variance in transmission times. On the Ncube 3200, the low resolution of the clock masks any variance, so transmission times are within plus or minus one tick (128 $\mu$s). Transmission time increases with message size and the number of hops [3], and for the Ncube 6400, the variance in transmission time increases with multiple hops.

No clock synchronization is required for the Ncube 6400 hypercubes, and only initial time synchronization is required for the Ncube 3200 hypercubes. The Ncube 3200 requires no frequency estimation, and since the variance is small, successive time-request messages and minimum filtering are not required. The Ncube 3200 application merely calls *clk_synch()* on each processor when the program starts and uses *sclock()* to read the synchronized and monotonically increasing time. As with the Intel hypercubes, *clk_synch()* uses a master-slave model, each processor synchronizing its time with processor 0.

For both the Intel and Ncube hypercubes, synchronization could be done faster using a spanning tree [4]. Processor 0 would be used to synchronize the clocks of its nearest neighbors. These clocks would then be the "master" clocks for their respective subtrees. There would be $n$ levels of master clocks in a dimension-$n$ hypercube, each level synchronizing its clocks with the level above. Clock synchronization could be completed in $n$ steps rather than the $2^n - 1$ steps required by our master-slave scheme. However, using spanning tree synchronization increases the error in both the offset estimate and skew estimate for a dimension-$n$ hypercube to $n\epsilon_\theta$ and $n\epsilon_\phi$ , where $\epsilon_\theta$ ($\epsilon_\phi$) is the one-hop error. For both the Intel and Ncube hypercubes, the $n$-hop error is much less than $n\epsilon_\theta$ (Table 3.2), so we choose the more accurate master-slave scheme. The time for the $n$-step synchronization is not great, since the time to synchronize a node is less than one millisecond for the iPSC/860.

## 4. Limitations and Extensions.

Our master-slave, minimum filter, synchronization algorithm has eliminated the timing anomalies in our event trace files. The algorithm is implemented as application subroutines, requires no modifications to the operating system, and can be extended to other distributed, parallel processor topologies such as a mesh. Preliminary testing of the iPSC/860 synchronization subroutines on the 512-node, i860-based Intel Touchstone DELTA mesh [6] at Cal Tech was successful. The DELTA clocks are the same as those used on the iPSC/860, but message passing is much faster for multi-hop messages and has less variance. The minimum message-passing time, $U_0$, is only slightly smaller (50 $\mu$s) [2].

The algorithm requires an initial delay while the processors estimate frequency corrections, followed by periodic resynchronizations. (Frequency corrections could be calculated when the hypercube is rebooted and saved in a file, these corrections could then be read by *clk_synch()* when an application starts.) Our implementation requires only four messages per processor to synchronize the hypercube clocks within tens of microseconds of each other. For the Ncube 3200 hypercube, only initial time synchronization is needed, no frequency correction or resynchronization is required. The Ncube 6400 hypercube clocks are synchronized by design.

Our algorithm has assumed a dedicated hypercube and idle communication channels. However, in some cases, other traffic may be competing for the communication channels, for example, if the hypercube is being shared with other applications or if there is traffic from other parts of the application during resynchronization. In this case the variance in the message delay and our error bounds increase. In fact, message delays can be much larger, so to maintain the same error bounds, it might be be necessary to add a minimum-delay threshold and keep sampling until delays fall below that threshold.

If multi-tasking is supported on a hypercube processor, then the synchronization algorithm needs to be implemented in the operating system, so that each process on a processor has a synchronized, monotonic clock. If time synchronization is required with external clocks, then processor 0 could synchronize with the hypercube host processor. However, typical external clock synchronization algorithms over a local area network are usually accurate to only tens of milliseconds [9]. Finally, it is possible for time to appear to run backwards as a result of a resynchronization, possibly resulting in a timing anomaly in an application. More sophisticated local clock software is needed to slew a clock to the proper time, rather than making a step adjustment backwards in time. Mills describes the necessary characteristics of a local clock [9].

Distributed parallel processors need high precision, synchronized clocks. We have demonstrated that hypercube clocks can be synchronized using only application software with very little overhead. However, to support fault tolerance, slewing, and tighter synchronization, much more software will be needed. It is much more desirable to have a single clock like the Ncube hypercubes or hardware synchronized clocks, eliminating the intrusion and software overhead and providing tighter synchronization.

## 5. References

[1] F. Cristian. Probablistic clock synchronization. Technical report, IBM Almaden, 1989. Research Report RJ 6432 (62550).

[2] T. H. Dunigan. Communication performance of the Intel Touchstone DELTA mesh. Technical report, Oak Ridge National Laboratory, 1991. ORNL/TM-11983.

[3] T. H. Dunigan. Performance of the Intel iPSC/860 and Ncube 6400 hypercubes. Technical report, Oak Ridge National Laboratory, 1991. ORNL/TM-11491.

[4] J. C. French. A global time reference for hypercube multicomputer. In John L. Gustafson, editor, *Hypercube Concurrent Computers and Applications 1989*, pages 217–220, New York, 1990.

[5] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A Users' Guide to PICL A Portable Instrumented Communication Library. Technical report, Oak Ridge National Laboratory, October 1990. ORNL/TM-11616.

[6] Intel. *A Touchstone DELTA System Description.* Intel, Portland, Oregon, 1991. Intel Advanced Information.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 7:558 – 565, 1978.

[8] D. L. Mills. Network time protocol (version 2) specification and implementation. Technical report, DARPA Network Working Group, September 1989. RFC-1119.

[9] D. L. Mills. Network time protocol (version 3) specification and implementation. Technical report, DARPA Network Working Group, September 1990. Draft RFC.

[10] P. Ramanathan, D. D. Kandlur, and K. G. Shin. Hardware-assisted software clock synchronization for homogeneous distributed systems. *IEEE Trans. Computers*, 39:514–524, 1990.

[11] P. Ramanathan, K. G. Shin, and R. W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23:33–42, 1990.

[12] K. G. Shin and P. Ramanathan. Transmission delays in hardware clock synchronization. *IEEE Trans. Computers*, 37:1465–14671, 1988.

[13] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *J. ACM*, 34:626–645, 1987.

[14] N. Vasanthavada and P. N. Marinos. Synchronization of fault-tolerant clocks in the presence of malicious failures. *IEEE Trans. Computers*, 37:440–448, 1988.