RC 20412 (3/18/96)
Computer Science

IBM Research Report

Clock Synchronization on a Multicomputer

Bulent Abali, Craig B. Stunkel

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Caroline Benveniste

Columbia University
Computer Science Dept., New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

page intentionally left blank for double sided printing.

Clock Synchronization on a Multicomputer

Bülent Abali*, Craig B. Stunkel*, Caroline Benveniste†

* IBM Thomas J. Watson Research Center,
P.O.Box 218, Yorktown Heights, NY 10598
{abali|stunkel}@watson.ibm.com

† Computer Science Dept., Columbia University
cdb@vlsi.columbia.edu

Abstract

We describe hardware and software schemes for achieving precise clock synchronization on SP2 parallel system nodes. The SP2 multistage interconnection network has an unusual hardware feature, a set of distributed counters that the processor nodes may utilize for synchronizing their time–of–day clocks. We describe an algorithm for synchronizing the counters within less than 200 nanoseconds of each other in a network of up to 512 processor nodes. This is 4–5 orders of magnitude better than what can be achieved by existing software schemes. We also describe an experimental system software, called `sptimed`*, for synchronizing the node clocks to the Internet time–of–day, utilizing the synchronous counters in the SP2 network.* `Sptimed` *synchronizes the node clocks typically within 5 microseconds of each other, which is up to 2–3 orders of magnitude better than what can be achieved by previous methods on the SP2 system. Synchronized clocks are useful in parallel and distributed environments, for example for performance measurement, tuning, tracing, debugging, gang scheduling of parallel processes, and timestamping of transactions, to name a few. We also measure the performance of a widely used time synchronization utility, the Network Time Protocol, using the synchronous counters of the SP2 interconnection network.*

A preliminary version of this paper appeared in IPPS '95. This paper is much revised and extended, most notibly adding a section on further improvements to the clock synchronization algorithms (Section 3.3), analyzing more results in Section 4, formalizing the switch synchronization procedure in Section 2.2, adding Sections 1.2 and 3.1, etc.

1 Introduction

Synchronized clocks are useful in parallel and distributed systems, because they allow events occurring on different nodes to be logically ordered in time. In this paper, we describe hardware and software schemes for achieving precise clock synchronization on the SP2 multicomputer [1, 2, 3, 4]. The SP2 multicomputer consists of a collection of RISC System/6000 workstations interconnected by a high speed multistage interconnection network. The SP2 network has an unusual feature, a set of counters that are synchronized within 25 to 200 nanoseconds (ns) throughout a network of 16 to 512 workstation nodes, which is 4 to 5 orders of magnitude better than what can be achieved by existing schemes. The synchronized feature of the network simplifies the well-known clock synchronization problem encountered in distributed systems [5]. We implemented an experimental system software (a daemon) called `sptimed` for synchronizing the processor node clocks in the SP2 system. The daemon continuously tracks the Internet time, and synchronizes the independent clocks of the processor nodes typically within 5 microseconds (μs) of each other. This result is 2 to 3 orders of magnitude better than that of previous methods used in UNIX workstations. The `sptimed` daemon makes use of the SP2 interconnection network to achieve this performance. In this paper, we describe the algorithm used for synchronizing the network counters, the `sptimed` daemon for synchronizing the node clocks, and we present the experimental results. We also characterize the performance of a widely used time synchronization utility called the Network Time Protocol (NTP) [6, 7, 8, 9] on the SP2 parallel system. In the experimental results section, we present a series of figures showing time differences between multiple node clocks, measured with a sub–microsecond accuracy. These measurements were made feasible by the synchronized feature of the SP2 network, and we are not aware of similar published data.

Synchronized clocks may be beneficial for parallel system performance measurement and tuning, gang scheduling of parallel processes, parallel program tracing, debugging, and timestamping transactions in database applications, to name a few.

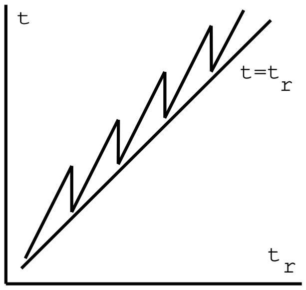

Figure 1: Periodic synchronization of a clock relative to the reference clock (t_r)

1.1 Background

Clocks on multiple processor nodes, each driven by an independent oscillator, will drift apart over a period of time if not periodically synchronized. Figure 1 illustrates a clock running faster than a reference clock and being synchronized periodically. In this paper, *synchronized* means that a set of clocks have the same time value at a given time (± the error allowed.) The *drift* of a clock is the frequency error relative to a reference clock. Oscillator manufacturers quote frequency errors typically on the order of 1 part per million (1μs/s). This means that a clock will drift away from a reference clock at a rate of 1 microsecond per second, or 3.6 milliseconds per hour. We have observed drifts of as much as 10μs/s in practice.

Clock synchronization is performed virtually on all UNIX workstations today by clock synchronization daemons such as the Network Time Protocol (NTP) or the `timed` daemon of 4.3BSD [6, 10]. Clock synchronization daemons are specialized software for distributing and receiving time over local area networks (LAN) and Internet. A time server or a hierarchy of time servers periodically provide the time–of–day to client nodes which can then adjust their clocks accordingly. We made measurements on a cluster of SP2 nodes running NTP, and found that they were synchronized to within 1–3 milliseconds of each other.

Synchronization with a standard clock daemon is not sufficiently accurate for some parallel applications that use the high speed SP2 network. This is particularly true when doing performance analysis of parallel systems using traces. Often traces are generated on individual processors, and the merging of traces occurs postmortem. If the traces from different processors are merged by local timestamp values alone, strange results appear in the merged traces, such as messages being received before sent. To avoid this, the node clocks must be synchronized to within less than the interprocessor communication latency. Software communication latency over the SP2 network is about 30 μs. Therefore, it is desirable to have the node clocks synchronized to within less than 30 μs, so that events occurring on different nodes can be logically ordered in time. In the absence of synchronized clocks, even the simplest measurement in all, the one–way communication latency is impossible to measure, and one is forced to measure the roundtrip latency and then divide the result by 2. This technique assumes that the latency is symmetric which may not be true, especially in the presence of high network traffic.

1.2 Related Work

Mills describes the architecture and algorithms of the Network Time Protocol (NTP) [6]. NTP, installed over hundreds of thousands of workstations is de facto standard for synchronizing clocks and distributing time on the Internet. The NTP system consists of a hierarchy of primary and secondary time servers, and clients dynamically organized according to their accuracy and reliability. Mills describes design improvements in the hardware, NTP software, and Unix system software which improve the accuracy and stability of workstation clocks [7, 8, 9].

In related works, Wu *et al* describes a parallel program trace generation facility that uses the globally synchronous SP2 network clocks for logical ordering of events occurring on multiple nodes [11]. Maillet and Tron describe statistical methods for estimating global time used in event tracing in parallel systems [12]. Mraz reports experiments that show variance in message delays and execution times of a parallel program due to lack of coordinated process scheduling [13]. His results show that if operating system timer interrupts on different nodes are synchronized, then the operating system services occur exactly at the same moment on all nodes, removing one source of the variance, and speedup of the parallel program increases. Feitelson and Rudolph also emphasize the importance of precise scheduling and fine grain synchronization of a "gang" of threads that are scheduled to execute in parallel on a set of processors [14].

Many papers on clock synchronization exist in the literature. See [7, 8, 9] for surveys. All of the synchronization methods that we know of are statistical techniques for measuring and estimating roundtrip communication delays between processors, subject to the uncertainties in the network load and delays in the communication software protocol. Our method of synchronization differs from the previous methods in that we do not measure the roundtrip communication delays between processors, but we first synchronize deterministically (with a known upper bound of 25 to 200 nanoseconds depending on the network size) a set of distributed counters driven by a single common oscillator (described in Section 2). We then formulate the Internet time as a function of the synchronized counter values, utilizing a linear time equation (described in Section 3). With the linear time equation in hand, processors can use their respective counter values to remain tightly synchronized to each other, and also remain synchronized to the Internet time. The advantage of our hardware/software combined method is that (a) synchronization is completely deterministic and eliminates statistical uncertainties of measuring and estimating round–trip delays in the network (b) synchronization is drift free and immune to local oscillator instabilities of the processor nodes. The methods presented here may be utilized in shared memory or distributed memory multiprocessors.

The rest of the paper comprises three main parts. In Section 2, we give an overview of the SP2 network and synchronization of the network counters. In Section 3 we describe the `sptimed` daemon for synchronizing OS clocks. We also give a parallel program tracing application as an example, where synchronized clocks would be useful. Results obtained for `sptimed` and comparison with NTP are presented in Section 4.

2 Synchronizing the Network Counters

The SP2 multicomputers connect hundreds of RISC System/6000 processors via a communication network called the High-Performance Switch, an example of which is shown in Fig. 2. In each network element is a counter called the Absolute Time Counter (ATC). The ATC in each element is synchronized within one clock cycle (25 ns) of one or more of its immediate neighbors' ATCs. The ATCs facilitate a closely synchronized, non-drifting *global*

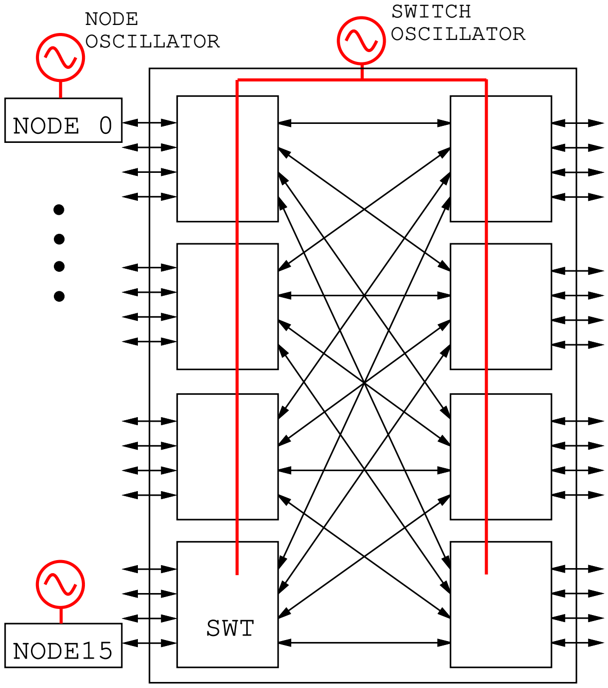

Figure 2: A 16 processor node SP2 network with 8 switch chips

time reference available to all the processor nodes, and thus they simplify the well-known clock synchronization problem encountered in distributed systems [5]. The ATCs primary function is to enable the network to synchronously cycle between its two primary modes of operation called the *run* mode for normal data transfer and the *service* mode for servicing the network. In this section, we give an overview of the network architecture and we describe the algorithm for synchronizing the ATCs.

2.1 SP2 Network Architecture

The SP2 network [1, 3] is a bidirectional multistage interconnection network (MIN) as shown in Fig. 2. In a bidirectional MIN [15, 16] each communication link comprises two channels which carry data in opposite directions. The network consists of two types of elements—8 port switch chips and single port network interface chips [3]. The network interface chip on each processor node is an interface between a switch chip and the RISC System/6000 processor. Figure 2 illustrates a 16–node SP2 network, with nodes connected by links to a switch board that consists of 8 switch chips. Unused ports on the right side of the switch board in Fig. 2 may be interconnected to other switch boards to construct larger networks. The largest network constructed to date has 512 processor nodes. Each port consists of an 8-bit wide input and an 8-bit wide output. Ports are interconnected by potentially long communication links that can have electrical lengths of up to 15 clock cycles to support large number of nodes. Thus, the cable interconnecting two network elements is a transmission line which may contain up to 15 bytes in each direction at any given time.

The SP2 network operates synchronously. All network elements are driven by a single 40 MHz (25 ns cycle) oscillator whose signal is distributed throughout the network, as illustrated in Fig. 2. It is not necessary to align the phase of the oscillator signal received by each element. Instead, using adjustable delay lines in the data paths, each element adjusts the phase of its *data* by the process of *tuning* [1, 2, 3] such that data transmission is synchronous from an output port to the corresponding input port of a neighboring network element. This allows both the oscillator signal links and the data links to be of arbitrary length up to the limits posed by reliability and protocol considerations. Tuning is performed throughout the network prior to ATC synchronization operations, which are described in the next section.

2.2 Synchronization of ATCs

Every SP2 network element, both the switch and the network interface chip, contains a 48-bit counter known as the Absolute Time Counter (ATC) which increments every clock cycle. Since all network elements are driven by a common oscillator, the ATCs are drift free. ATCs primary function is to control the timing of the mode transitions of the synchronous SP2 network. Without going into further detail, it suffices here to state that ATCs must be accurately synchronized to prevent loss of link data in transit during these mode transitions. Further details on the SP2 network operation may be found in [1, 2, 3]. Synchronization of ATCs along with other network initialization functions are integrated into a software package called *the Worm* which runs on an arbitrarily selected processor node designated as the *service processor* [3].

During initialization of the network, the Worm determines the propagation delay of all the communication links in the form of round-trip latency (delay) measurements, which are then

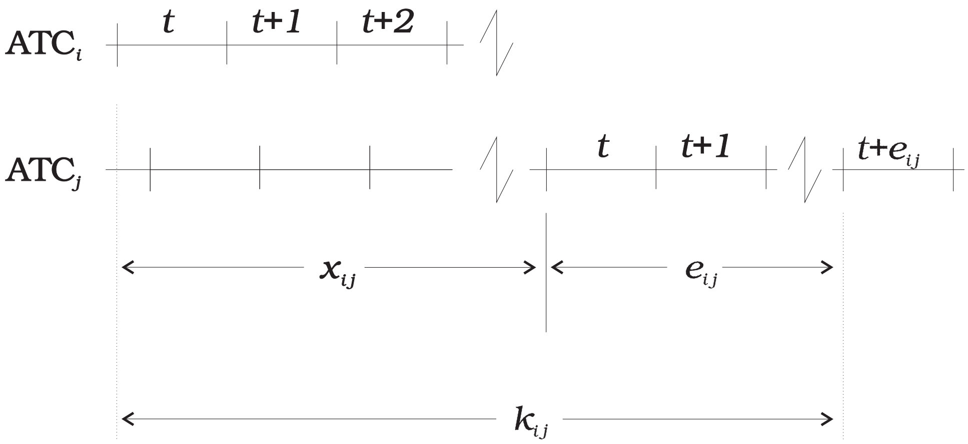

Figure 3: Effective latency, e_{ij}, for message sent from element i to element j at time t

used for synchronizing the ATCs. The Worm can determine the round-trip delay (in number of clock cycles) of a pulse sent out of the service processor's output port, then circulated through a selected set of network elements, and then received back at the service processor's input port.

Consider an arbitrary **bidirectional** network represented by the **directed** graph $G = (V, E)$, where the set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ represents the network elements, and the set of edges E represents the point-to-point bidirectional links between these elements. Edges will be expressed by an ordered pair of vertices (v_i, v_j). For each edge (v_i, v_j) in the graph, there exists an edge (v_j, v_i) which we shall term the *mirror* edge of (v_i, v_j). Two links (edges) are *adjacent* if they share a common element (vertex), and a *path* is a sequence of adjacent links. A *cycle* is a path with the first and last elements the same. We will assume that the network G is *connected*; that is, there is at least one path between every pair of elements in G. A *tree* is a connected graph containing no cycles, and a *spanning tree* of the network $G = (V, E)$ is a tree containing every element in V.

Let the *latency*, k_{ij} represent the signal transmission latency from network element i to element j, shown in Fig. 3. Let the *offset*, x_{ij} represent the real number of cycles between the instant that ATC$_i$ first equals t, and the instant that ATC$_j$ equals t. If ATC$_i$ reaches t after ATC$_j$ reaches t, then $x_{ij} < 0$. Both k_{ij} and x_{ij} are real numbers (non-integer) because of the phase difference between the oscillator images driving elements i and j. Furthermore, because the oscillator phase difference affects k_{ij} and x_{ij} identically, $k_{ij} - x_{ij}$ is an integer.

Now define the *effective latency*, e_{ij} as follows. If element i transmits a signal when ATC_i is t_i, then element j receives this signal when ATC_j is $t_i + e_{ij}$. Then

$$e_{ij} = k_{ij} - x_{ij}. \tag{1}$$

This relationship is illustrated in Figure 3. The effective latency e_{ij} is thus the latency observed by (i.e., apparent to) the network element j on link ij. For example, link ij may be 5 clock cycles long, however element j will observe the latency as 2 cycles long if its counter ATC_j is lagging ATC_i by a count of 3. Effective latency is an integer quantity.

Equation 1 can be restated as

$$k_{ij} = e_{ij} + x_{ij}. \tag{2}$$

Naturally, latencies are additive when traversing a number of links in a path, P:

$$k_P = \sum_{(v_i,v_j)\in P} k_{ij} = \sum_{(v_i,v_j)\in P} (e_{ij} + x_{ij}) \tag{3}$$

In particular, a path that begins and ends at the same element v_i (a *cycle*, C) is of interest, because an observer (a hardware circuit) at v_i can only measure signal latencies from a single point. Cycle latency measurements also permit the formulation of equations containing latencies and effective latencies without offsets. Two lemmas about sums of offsets are introduced to derive this result.

Lemma 1 *The offset between two elements v_i and v_j is equal to the difference in their offsets with respect to a third "reference" element v_0.*

PROOF: Let t_0 be the instant at which ATC_0 reaches the value t. Consistent with our definition of offset, ATC_i will first become t at time $t_0 + x_{0i}$, and ATC_j will first become t at time $t_0 + x_{0j}$. Therefore,

$$x_{ij} = (t_0 + x_{0j}) - (t_0 + x_{0i}) = x_{0j} - x_{0i}$$

Lemma 1 leads to Lemma 2, which is similar to Kirchhoff's Voltage Law for electrical circuits.

Lemma 2 *The sum of the offsets around a cycle is zero.*

PROOF: Assume a cycle C formed by the edges (v_1, v_2), (v_2, v_3), $\ldots$, (v_{n-1}, v_n), (v_n, v_1). As in Lemma 1, assume a reference element v_0.

The sum of offsets around C is

$$\sum_{(v_i,v_j)\in C} x_{ij} = (x_{02} - x_{01}) + (x_{03} - x_{02}) + \ldots + (x_{0n} - x_{0,n-1}) + (x_{01} - x_{0n}) = 0$$

From Equation 3 and Lemma 2, it follows that

$$k_C = \sum_{(v_i,v_j)\in C} k_{ij} = \sum_{(v_i,v_j)\in C} e_{ij}. \tag{4}$$

Latency measurements performed for a network cycle thus permit one to formulate equations with effective latencies as the only unknowns. Because one can only observe latencies from only one position at a time in the system, network cycle measurements are also the *only* valid measurements obtainable (i.e. round-trip delays measured to/from the service processor.)

The SP2 machines impose an additional requirement on each link's effective latency due to specialized operations that are keyed to the ATC and that are also dependent upon link data arriving within an expected 16 cycle window [3]. For a link carrying signals from element i to element j, correct operation requires that

$$0 \leq e_{ij} \leq 15. \tag{5}$$

Thus, to sufficiently synchronize ATC's, the Worm must determine a set of effective latencies that satisfies Eq. 4 for every cycle in the network, and simultaneously satisfies Eq. 5 for every link in the network.

Every edge (v_i, v_j) in E forms a simple two edge cycle with its mirror edge (v_j, v_i), allowing the Worm to conduct a timing experiment with only (v_i, v_j) and (v_j, v_i) as unknowns. Thus, the expression $k_{ij} + k_{ji} - e_{ij}$ can always be substituted for e_{ji}, halving the number of independent effective latencies.

In practice, a bidirectional link (an edge and its mirror) has delays nearly equal in both directions due to equal cable lengths. Thus, we make the default effective latency assignment by dividing the roundtrip delay by 2. If the roundtrip delay $k_{ij} + k_{ji}$ of a link is odd, then for the opposite directions the effective latencies are assigned as $\lceil k/2 \rceil$ and $\lfloor k/2 \rfloor$, respectively. This "equal length" heuristic makes it easier to calculate a set of effective latencies that satisfy Equation (5).

A permissible set of effective latencies is determined while considering only a subset of the cycles in the network. The Worm, beginning with the service processor v_{sp}, visits each reachable element (each vertex in graph theory terms.) Such a search is accomplished by successively examining each *unexamined* link (edge) and element of the network G in breadth-first fashion, discarding links that form cycles with the links already selected. This search results in a *spanning tree* (V, T) of the network $G = (V, E)$ (where T is a subset of E) which contains the minimum number of links required to connect every element [17]. Any such spanning tree contains $|V| - 1$ edges, $(|T| = |V| - 1)$.

Let T_M be the union of the set of edges T in the spanning tree along with each of their associated mirror edges (T_M therefore contains twice as many edges as T). Any edge not in T_M will create exactly one cycle when combined with a subset of the edges in T_M. Let the set of latency equations formed by such cycles be called D_F. D_F contains $|E| - |V| + 1$ linear equations, and these equations are independent because each equation contains an edge not found in any other equation in D_F.

Theorem 1 *The set of all possible cycle equations D has rank $|E| - |V| + 1$.*

PROOF: The set of cycles associated with the equations in D_F is called a *fundamental set* of cycles for the spanning tree T [17]. Let this set be called F, where $F = \{C_1, C_2, \ldots, C_{|E|-|V|+1}\}$. Then all other cycles in G can be written as $((C_{i_1} \oplus C_{i_2}) \oplus \cdots) \oplus C_{i_t}$, where $\oplus$ represents the *symmetric difference* operation [17], and $A \oplus B = \{x | x \in A \cup B, x \notin A \cap B\}$. I.e., $A \oplus B$ includes all elements of A and B that are not in both A and B. Depending on the assumed direction (clockwise or counter-clockwise) of cycles A and B, either $k_A + k_B$ or $k_A - k_B$ will contain only the edges contained in $A \oplus B$. Thus $k_{A \oplus B}$ is linearly dependent upon the k_A and k_B latency equations. Therefore all cycle latency equations D are dependent upon D_F, which has rank $|E| - |V| + 1$.

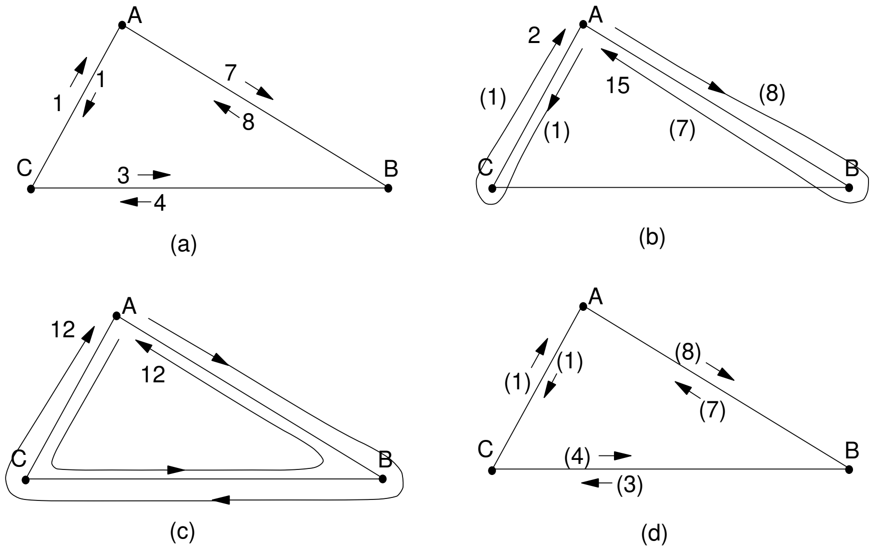

Figure 4: Calculation of effective latencies by the Worm. Actual latencies are shown in (a). The Worm makes round-trip measurements in (b) and (c), and assigns effective latencies—indicated with parentheses—as shown in (b) and (d).

The significance of Theorem 1 is that once we have observed the latencies for a fundamental set of cycles F, we can do no better; we have the maximum information obtainable about how the synchronize the clocks in the network. The set of equations D_F, along with Equation (5) applied to each link, forms an under-constrained system of linear equations with effective latencies as the variables. Once a set of effective latencies is found that satisfies these equations, the ATC of each element can be synchronized easily. In practice, using the "equal length" heuristic introduced above, the effective latencies chosen for spanning tree edges and the remaining effective latencies calculated from the fundamental cycle equations D_F typically satisfy equation (5) without further adjustment. Due to the uncertainty of assignment here, there may be up to 1 clock cycle time difference between two neighboring network elements on the spanning tree (relative to an absolute time reference). Currently, the largest SP2 network has a diameter of 8 hops. Therefore, the time difference between any two ATCs is at most 8 clock cycles (200 ns).

The example in Fig. 4 illustrates the entire procedure on a simple hypothetical network of three vertices. The actual latencies of the bidirectional links are shown in Fig. 4(a).

From node A, where the service processor is located, a spanning tree T that consists of the bidirectional links (A, B) and (A, C), is built as shown in Fig. 4(b) and round trip delays are measured as 15 and 2 cycles, respectively. Effective latencies (shown in parentheses as $e_{AB} = 8$, $e_{BA} = 7$, $e_{AC} = 1$, $e_{CA} = 1$) which satisfy the round trip delays are also assigned to the links using the equal length heuristic.

In the next step, shown in Fig. 4(c), the link (B, C) is added to T to form a network cycle, and the round trip delays are measured as 12 clocks in both directions of the network cycle. By using the network cycle delays, and effective latency assignments of step (b), and Eq. 4, effective latencies of the link (B, C) are determined as $e_{BC} = 3$ and $e_{CB} = 4$.

The Worm then initializes the ATCs assuming the calculated effective latencies. In the example, it sends a counter initialization command over the link from A to B at time $\mathrm{ATC}_A = t$, "set your counter to $t + 8$", which after 7 cycles node B receives and promptly executes. After this initialization, when $\mathrm{ATC}_A = t$, the other counters will be $\mathrm{ATC}_B = t + 1$ and $\mathrm{ATC}_C = t$. Therefore, although a message sent from node A at time t will arrive at node B 7 cycles later, node B will receive the message at time $t + 8$ according to its own time reference ATC_B, consistent with the effective latency assignment $e_{AB} = 8$. Although there is a 1 cycle difference between ATC_A and ATC_B relative to an absolute time refence, the nodes cannot observe this difference, since their ATC values are consistent with the effective latency assignments.

3 Synchronizing the Operating System Clocks

The RISC System/6000 processor nodes each have an independent oscillator, as illustrated in Fig. 2. Although processor nodes could derive their clock signal from the common 40 MHz network oscillator, system integration requirements made it impractical and costly to do so. Since oscillators are independent, the node clocks will drift apart if not periodically synchronized. The clock synchronization daemon `sptimed` uses the synchronized Absolute Time Counters (ATC) in the SP2 network as a global time reference to synchronize the node clocks. The ATCs are synchronized within 200 ns throughout the network as explained in Section 2, and each processor can access the ATC on its respective network interface [1, 3].

The tightly synchronized ATCs do not make the processor node clocks obsolete. Because there are many existing user and system applications and the OS itself that are tied to the node clock. Synchronizing the node clocks via the network counters enables those applications to reap the benefits of having a synchronized clock. An example application and motivation for having synchronized node clocks is discussed in the next section. We then describe `sptimed` in Section 3.2. Note that in the following discussions, we use the terms "node clock" and "OS clock" interchangeably.

3.1 Tracing Parallel Applications

Parallel programmers rely on tools to help them understand the behavior and performance of their programs. One popular approach to performance monitoring is the use of execution traces. Traces are necessary for program visualization [18] and trace-driven simulation of computer systems [19], and can be useful for debugging programs and statistical analysis. On a distributed system, typically traces are generated on each processor individually and the individual traces are merged later during analysis. This is the case on the SP systems when the tracing package UTE (Unified Trace Environment) is used.

UTE is a tracing environment built on top of the AIX tracing facility built in to the operating system. By enabling tracing, the user may collect trace information from his program without any changes to the source code. UTE supports tracing of MPI [20] and the IBM Parallel File System library routines [21]. UTE produces a trace record at the beginning and end of each routine. For example, if a processor issues a non-blocking send in MPI, there will be a trace record at the beginning of the send (*b_Mpi_Send*) and a trace record at the end of the send (*e_Mpi_Send*). In addition, since UTE uses the AIX tracing facility, system events can also be traced. By using these traces the user can gain insight into his program's behavior. Each trace record generated by UTE contains a timestamp that corresponds to the time of the node clock.

Ideally, the traces would be merged by sorting the trace events by timestamp. However, if the node clocks are not synchronized, merging the traces by timestamp alone results in a confusing program behavior, such as a message being received by a processor before it was sent by another processor. On the SP systems, when NTP alone is used to synchronize the clocks, accuracy of synchronization is not sufficient to prevent this anomalous behavior. Therefore, when using UTE traces it was necessary to write a merge program that performs matching operations between related events (e.g. a send and it's receive) and adjusts the timestamps accordingly [22].

Our main concern in merging trace files from different processors is to maintain a logical ordering of events. Certain events, such as blocking receives and waits (*Mpi_Brecv* and *Mpi_Wait* in MPI) imply a fixed ordering. For example, one processor cannot complete a blocking receive call until the message has been sent. Therefore, the merge program identifies matching sends and receives in the trace files (in the trace record of a blocking receive or wait, the source processor number is indicated, so the merge program can identify which processor should have issued the send.) The merge program imposes the condition that a receive cannot complete before a send has been executed. If such a situation is detected, the time of the receive event is adjusted to be the time of the send event plus the minimum network latency. This is an approximation that assumes a lightly loaded network which may not be accurate in general. However, it guarantees logical ordering of events.

This technique is similar to the one described by Lamport in [5]. There, a distributed algorithm for synchronization is given which yields a total ordering of events in a distributed system. Since we do not process traces in real–time, we do not have to use a distributed algorithm. However, since tracing is transparent to the user (i.e. we do not require processors to send additional messages for synchronization purposes), there is no way to synchronize the events in sections of the program in which no communication occurred.

The matching technique works relatively well for point-to-point communication events in correctly written programs. The problem of merging trace files becomes more complicated when it is extended to *collective* communication events such as *broadcast* or *synchronize*. In this version of UTE, the underlying point-to-point events within a collective communication event are not traced, since tracing these events increases size of the trace files and tracing overhead [22, 20]. Without these additional events, a more sophisticated algorithm is needed for merging by logical ordering of events. Rules are developed for the causality relationships in each of the collective communication routines. For example, in a broadcast,

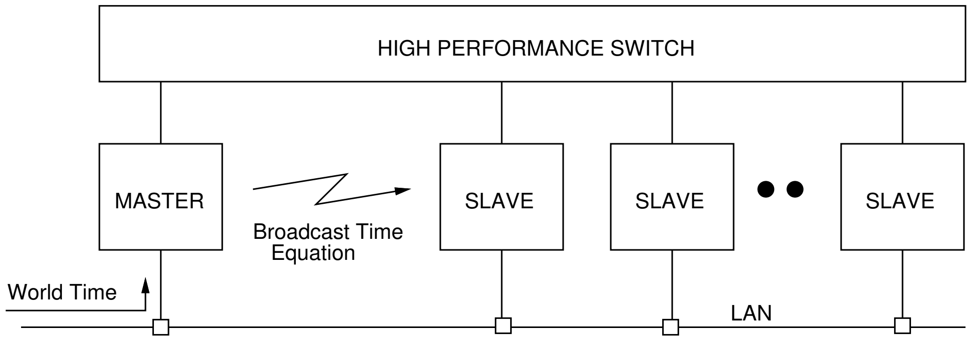

Figure 5: The master broadcasts constants of the linear time equation $T = at + b$. The slaves determine the OS time T, by reading the synchronous switch time t and computing $T = at + b$. The master may also loosely track an external time server.

the begin broadcast (*b_Mpi_Bcast*) event of the source node must precede the end broadcast (*e_Mpi_Bcast*) event of all participating nodes. For a program synchronization event, all the *begin sync* events must precede any *end sync* events. Thus, in the merge program when an end sync event is encountered on one processor, the program must check to see if all other processors participating in the sync have issued their begin sync events. In addition, the time adjustments that are made are approximate, and this may affect any statistics that are derived from analyzing the merged trace.

3.2 Clock Synchronization Software `sptimed`

The primary function of the `sptimed` daemon is to keep the operating system clocks tightly synchronized on each node and also track the Internet time-of-day. The `sptimed` daemon is based on the Master/Slave concept shown in Fig. 5. In this scheme, the node designated as the Master periodically broadcasts to the slave nodes the constants a, b of the linear time equation

$$T = a \cdot t + b \tag{6}$$

where t is the SP2 network counter value and T is what the OS time should be at time t. The constants a, b are determined by the master node such that they relate the SP2 network time t to the master node's OS time T, which in turn is related to the Internet time-of-day as will be explained in the following.

The computed value of T represents a globally synchronous time reference, because broadcast values of a and b are identical on all the nodes and t, the network counter values, are identical on all ATCs at any given time. Thus, slave nodes infer the master node's clock value T by reading t from their respective ATC in the network and then using Eq. 6. After computing the T value, each slave node adjusts it's clock accordingly. Using this linear time equation concept, we were able to synchronize all the node clocks within 5 μs of each other on the SP2 parallel system. This result compares favorably to the 1–3 ms synchronization error that we observed for the standard clock synchronization daemon. In the following, we describe this scheme in more detail.

Our objectives for clock synchronization were (a) to tightly synchronize the OS clocks in the entire cluster within few microseconds of each other, and (b) to make the entire cluster track the Internet time-of-day loosely, for example by a difference of few tens of milliseconds so that manual clock adjustments are not necessary. Here, the Internet time-of-day refers to the time periodically provided by a time server on the Internet. In our scheme, the master node shown in Fig. 5 serves as a "time gateway" that uses NTP to track the Internet time. NTP periodically adjusts master node clock in response to the information received from external time servers [6].

Along with NTP, the master `sptimed` daemon also runs on the master node, but it doesn't make any clock adjustments. Therefore, it is in "read–only" mode, in which it computes the master node's time T as a function of the SP2 network time (counter value) t. That is, the master `sptimed` daemon computes the constants a and b of Eq.6, such that the hypothetical reference clock represented by Equation 6 runs synchronously with the master node clock. The constants a and b are calculated by the method of least squares, a well known curve fitting method: `sptimed` takes timestamps (t_i, T_i) periodically, where t_i is the network time, and T_i is the node clock value during the i–th timestamp. Using the most recently taken M timestamps, `sptimed` finds constants a and b of the least square line represented by Eq. 6 passing through the M points

$$(t_{i-M+1}, T_{i-M+1}) \ , \ (t_{i-M+2}, T_{i-M+2}) \ , \ \ldots \ , \ (t_{i-1}, T_{i-1}) \ , \ (t_i, T_i)$$

We set empirically the timestamp period to 10 seconds and $M = 1000$ in the current implementation. Therefore, Eq. 6 contains a 10,000 second (approx. 2.8 hours) history of the master node clock. Here, the least square method acts like a low-pass filter, which removes the "time noise" from the master node clock. Time noise is oscillations, jitter, and other abnormal changes in time. In practice, some type of filtering is needed because external time servers and NTP on the master node tracking those servers, make abrupt clock adjustments that we don't want slave nodes to follow too closely. See Fig. 7 for examples of such abrupt adjustments on a collection of RS/6000 workstations; The method of least squares will average out the jagged edges of the curves and will result in a straight line which runs parallel to the envelope of the jagged curves.

3.2.1 Meaning of constants a and b

The constant a is the rate of change of the master node clock—averaged over time due to the filtering—relative to the rate of change of the SP2 network clock. For example, suppose the master node clock is running faster than the network clock by a drift of 1μs/s. It means that for every 1 second elapsed on the network clock, the master has 1.000001 seconds elapsed. Therefore the method of least square will produce $a = 1.000001$. The constant will a ensure that the slave node clocks run at the same speed with the master node clock.

The constant b is the seconds elapsed from the UNIX operating system time zero (0 hour, January 1st, 1970) to the network clock time zero which occurs every 81 days. To detect a network clock overflow the master time daemon always compares current time t_i to the previously read time t_{i-1}. If $t_i < t_{i-1}$, it means that the network clock had an overflow, and the master `sptimed` daemon corrects the constant b accordingly. If slave nodes detect an overflow they don't attempt to adjust time, but they wait for the next broadcast from the master which sends the correct values of a and b.

3.2.2 Master Node Algorithm

The master time daemon broadcasts constants a and b to the slaves periodically. Because a, b values may change—slowly in our scheme due to the filtering. For example, if the master clock changes its frequency due to clock adjustments or temperature and voltage variations.

Likewise the SP2 network oscillator frequency may change due to temperature and voltage variations, causing t to run faster or slower, therefore causing a, b values to change. In the current implementation we empirically chose the broadcast period equal to the timestamp period; each time the master obtains a new timestamp, it recalculates a, b using the most recent M timestamps and then broadcasts a, b. We implemented broadcasting using the UDP/IP (User Datagram Protocol) communication protocol [10]. The UDP/IP protocol allows for broadcasting on LAN segments such as the ethernet shown in Fig. 5.

3.2.3 Slave Node Algorithm

Slave `sptimed` daemons keep a local copy of the most recently broadcast constants a and b. Periodically, each slave daemon wakes up and reads time t from it's ATC and computes T using Eq. 6. The daemon then reads the node clock T', and compares it to T. If $T > T'$, it means that the node clock is running slower, and if $T < T'$, it means that the node clock is running faster than it should be. The daemon tells OS to advance the node clock by $T - T'$ and then goes to sleep again. In the current implementation, the adjustment period is 1 seconds. Since our time reference t is globally synchronous within less than 200 nanoseconds throughout the parallel system, now the synchronization accuracy of node clocks is dramatically improved.

Time adjustment requests are submitted to the OS by a routine called “usadjtime” that we developed. The magnitude of adjustment requests can be as small as 1 μs with usadjtime. In many UNIX based operating systems, time adjustment requests are made by a system call named “adjtime” that truncates adjustment requests to the nearest 1 ms integral value. “Adjtime” has other deficiencies that we will not detail here. Therefore we developed “usadjtime” as a replacement. Note that there is a routine called “tickadj” distributed in the NTP source code which can reduce the truncation value of “adjtime” to as low as 40 μs. However, it was reported that below this value of tickadj the adjtime routine does not work properly.

3.3 Additional Improvements in Synchronization Accuracy

In many computer systems, the clock itself is often a source of error in timing measurements and operations. The error may occur due to the hardware, such as oscillator instability, or due to the poor software implementation of the OS–hardware clock interface. Having a second independent clock—in our case the SP2 network counters—allows one to quantify and possibly compensate these errors, which we describe in this section. We show two possible sources of error and show how to eliminate them. Mills provides a more exhaustive list [7].

An improvement in synchronization accuracy in a parallel system is possible by considering the execution overhead of a clock adjustment individually on each node. The node clock loses time by the amount of overhead during each adjustment. We measured this overhead in the range of 13 to 43 μs for the AIX 3.2 operating system, the overhead depending on the processor speed. Typically a clock adjustment comprises 5 main steps: (1) an adjustment request Δ is submitted to the OS by the clock synchronization daemon, (2) the OS reads the current hardware clock value into a temporary variable T, (3) updates T with the adjustment value Δ ($T \leftarrow T + \Delta$), (4) updates clock dependent data structures such as alarm queues, (5) writes back the new value of T to the hardware clock. Between the reading (2) and writing (5) of T, there is some execution delay not accounted for; therefore the OS clock loses time. The execution overhead is a function of the processor speed, and could be ignored if all the nodes were identical in speed. However, an SP2 system may have a mixture of processor speeds and types. Therefore the overheads must be considered to improve the synchronization accuracy. In our method, each node determines its adjustment overhead v by the following experiment during initialization of the clock daemon. The daemon takes a timestamp (t_s, T_s), where t_s is the network time and T_s is the OS time, and then submits a request to put the OS clock forward by Δ μs, where Δ is some arbitrary nonzero number. The daemon waits for some time (10 ms in our implementation) for OS to complete the adjustment, and takes another timestamp (t_e, T_e). The equation

$$v = t_e - t_s - (T_e - T_s) + \Delta \tag{7}$$

which ideally would be equal to 0, gives the time v lost during the adjustment. To increase

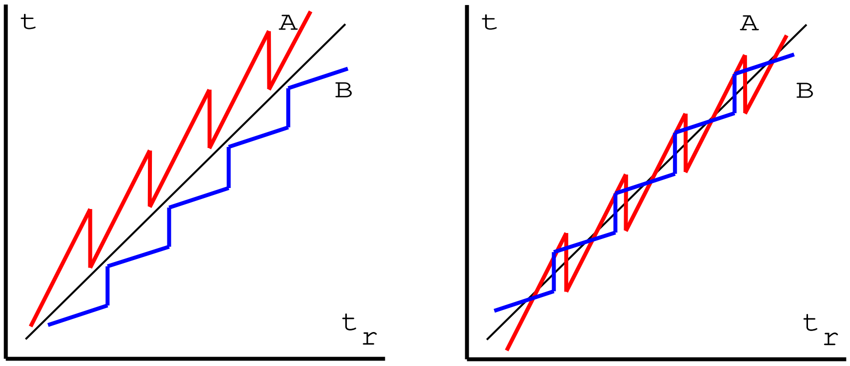

Figure 6: Reducing the maximum clock difference by 50 percent

the accuracy, the experiment is repeated few hundred times and the median result is used. In the strict sense, Eq. 7 is an approximation because the network clock and the OS clock may not be running at the same speed (i.e. $a \neq 1.0$). However, since we set the timestamp period $t_e - t_s$ to 10 ms in this experiment, the error due to the approximation is negligibly small; with a drift of ε μs/s, T will be off from t by no more than 0.01ε μs due to drift, which is in the nanoseconds range.

Each slave may then add the overhead correction amount

$$C = a \cdot v \tag{8}$$

to the adjustment value when submitting a clock adjustment to the OS, where a is the constant received from the master, v is the overhead slave node has determined for itself.

Synchronization accuracy may be further improved by considering drift of the node clocks individually. Since node clocks each are based on an independent oscillator, they may be running at different speeds. If a node clock is running faster (slower) than reference time Eq. 6, then the node clock may be put slightly backward (forward) so that the synchronization error is reduced, as illustrated in Fig. 6(a)–(b). For example, assume that node A and B have drifts of $\varepsilon_A = 10$ μs/s and $\varepsilon_B = -10$ μs/s, and both nodes have the same time initially. After 1 second, node A time will be 20 μs ahead of node B time. However, if initially node A time was put backward 5 μs and node B time was put forward 5 μs, then after 1 second, node A time would be only 10 μs ahead of node B time. Therefore, if each node adds an offset of $-n \cdot \varepsilon/2$ to the time adjustment value, where n is the adjustment period and ε is the drift of the node clock relative to the reference time Eq. 6, then the upper bound for time difference between any two nodes will be reduced by half. Each slave daemon may determine its node clock drift ε_S relative to the network clock by measuring what the node clock would have read without any adjustments were made; during initialization each slave node takes two timestamps (t_s, T_s) and (t_e, T_e). Then,

$$\varepsilon_S = (T_e - T_s)/(t_e - t_s) - 1 \tag{9}$$

gives the drift relative to the network clock where t is the network time and T is the node clock value. The drift relative to the reference time given in Eq. 6 is $\varepsilon = \varepsilon_S - (a-1)$ where $a-1$ is the drift of Eq.6 relative to the network clock.

With the correction for the drift included, C in Eq. 8 becomes

$$C = a \cdot v - n \cdot (\varepsilon_S - a + 1)/2 \tag{10}$$

where a is the constant received from the master, and v, ε_S are the constants each slave daemon has determined for itself, and n is the clock adjustment period. As described before, each slave daemon needs to add correction C to the adjustment value during the periodic clock adjustments.

4 Results

We present here the results obtained for `sptimed` and NTP. We installed `sptimed` on a cluster of 42 MHz Model 550 RISC System/6000 processors interconnected by the SP network. In the first set of experiments the master node did not track the Internet time; it's clock was free running; We compare the results to the synchronization achieved by NTP on another SP cluster which consisted of four 62.5 MHz Model 370 RISC System/6000 processors.

We made the measurements via a program running on each node which takes timestamps (t, T) every 0.25 seconds from the network and the OS clock, and saves them to a file. The timestamps are later post-processed to find the difference between the node clock values for any given network time.

Fig. 8 shows synchronization by `sptimed`. The OS clock values of the four processor nodes are plotted for an interval of 10000 seconds. Fig. 9 shows a small section of Fig. 8 on a finer interval of 10 seconds. In these figures, the x–axis denotes the synchronous network time. The y–axis denotes the node time minus the synchronous network time. This subtraction allows one to see the difference between the node clock values in finer detail, on a y–scale of a few tens of microseconds. Without this subtraction, the curves would have approximately a 45 degree slope, similar to those in Fig. 6, and y–scales would be 10000 and 10 seconds long for Figs. 8 and 9, respectively. Intersection of a vertical line with the curves gives the difference between the node clocks at a given time.

The main result of the `sptimed` experiments is shown in Figs. 8 and 9. In this experiment, the master node clock was free running. The dips and peaks in the master node curve in Fig. 8 —small ones as well as the bigger ones at 16000 and 23000 seconds— indicate that the frequency of the master node clock relative to the SP2 network clock frequency was not constant but varying, indicative of oscillator instability. Oscillator instability is typically caused by temperature variations in the operating environment. In Fig. 9, the approximately 1 microsecond variations in the master node curve are not due to oscillator instability, but rather an artifact of the timestamping program that makes sampling errors. In Fig. 8, we see that the 3 slave node clock values reside in a 5 microsecond envelope throughout the 10000 second interval, demonstrating the amount of synchronization achieved by our method. We see in finer detail in Fig. 9, the clock adjustments `sptimed` makes once a second, apparent from the jagged shape of the curves. Positive slope of the curves indicate that slave node oscillators each run faster than the reference time of Eq. 6. However, the `sptimed` daemon periodically puts the slave node clocks back, evidenced by the sharp negative slopes occurring once a second.

Notice from the smooth shape of the slave node curves of Fig. 8, the least–square filter in action (as described in Section 3). The filter eliminates the "high frequency noise", the small dips and peaks of the master node clock observed in the figure. When analyzed in the frequency domain, the least–square filter is a low pass filter that eliminates the high frequency components and leaves only the fundamental frequency, which is in this case the curious sine wave shape of the slave node curves in Fig. 8. As we expected, while slave nodes are tightly synchronized among themselves, they track the master node loosely, evidenced by the time lag between the peaks of the master node and slave node curves at 15500 and 16000 seconds, respectively.

We compared the performance of `sptimed` to the standard time daemon NTP available on SP2 systems. Fig. 7 shows the amount of synchronization achieved by NTP on 4 nodes. Difference between the node clock values of the four nodes is plotted as a function of the synchronized SP2 network time. Note that this figure is unique in that we used the synchronized counters (ATCs) of the SP2 network to measure the performance of NTP; without the globally synchronized ATCs, it would be impossible to measure the actual time difference between the nodes with the sub–microsecond accuracy presented here. Fig. 7 shows that the maximum time difference among the nodes is about 3 ms. As stated before, the "adjtime" routine has a deficiency of truncating time adjustment requests to the nearest 1 ms integral value, which is apparent from the sawtooth shaped curves with 1 ms jumps along the y-axis. Note that the deficiency exist in the AIX operating system implementation of adjtime, and therefore results do not necessarily represent the performance of NTP on all hardware/software platforms. Mills reports synchronization accuracies better than 1 ms for NTP on other hardware/software platforms [9]. Mills also reports special implementations of adjtime with an overhead of as small as 3–7 microseconds on Sun Sparc-20 and DEC Alpha workstations [23]. For the 4.3BSD `timed` [6] daemon, we observed node clocks differing as much as 5 ms, and time adjustments were coarser grained, as much as 5 ms per adjustment.

In the next set of experiments, we configured the master node to track the Internet time using NTP, unlike the free running master node clock used in the previous experiment. Fig. 10 shows synchronization by `sptimed` and the least–square filter in action. The sawtooth shaped curve of the master node clock, with 1 millisecond jumps along the y–axis, shows the clock adjustments made by NTP. There are 3 other curves, overlapping each other, and running through the master node curve. These curves belong to the 3 tightly synchronized slave nodes. As explained in Section 3, the least–square filter in the master `sptimed` daemon eliminates the high frequency components, i.e. the sawtooth shape, of the master node clock which manifests itself as smoother slave node curves. To summarize, Fig. 10 shows that the slave nodes are immune to abrupt changes of master node clock and follow it loosely.

5 Summary and Conclusions

In this paper, we presented hardware and software techniques for accurately synchronizing clocks of a large scale multicomputer. We described the distributed counters of the synchronous SP2 interconnection network, and we presented an algorithm for synchronizing the counters to within few clock cycles of each other and consistent with the effective communication latencies between the counters. We then presented a clock synchronization daemon which can continuously track the Internet time and synchronize the independent clocks of the processor nodes typically within 5 microseconds, which is 2–3 orders of magnitude more accurate than the previous method used in the SP2 systems. We presented experimental data illustrating the synchronization achieved by NTP and the vast improvement provided by the SP2 hardware and the `sptimed` software.

Multiple clocks, each driven by an independent oscillator, need periodic synchronization because of oscillator instabilities. A highly accurate system of clocks is possible by making use of stable frequency sources such as atomic clocks, temperature controlled crystals, or GPS satellite receivers. However, those schemes do not justify the per node cost in a general purpose multicomputer. Our scheme of synchronization may be more cost effective: (a) use a common oscillator to drive a set of distributed counters, (b) a master daemon formulates the Internet time as a function of the network counter values, (c) slave nodes use the counters and master node information to derive a highly accurate time reference.

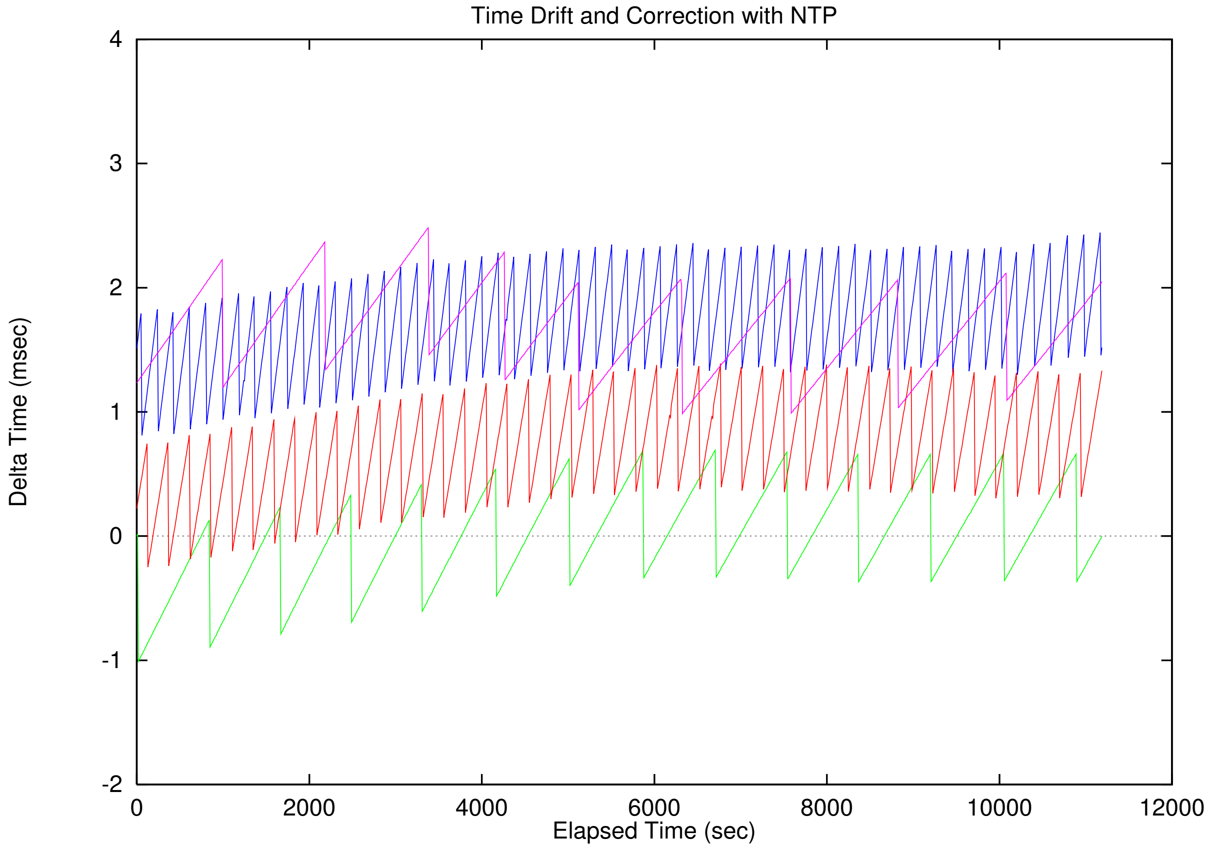

Figure 7: Clock synchronization with the Network Time Protocol (NTP) on AIX 3.2 operating system. The measurement is made on a cluster of four SP nodes and made possible by the synchronous network clock. The plot shows the difference between the node clock values on the four nodes as a function of the synchronous network time. Maximum difference between the node clocks is about 3 milliseconds.

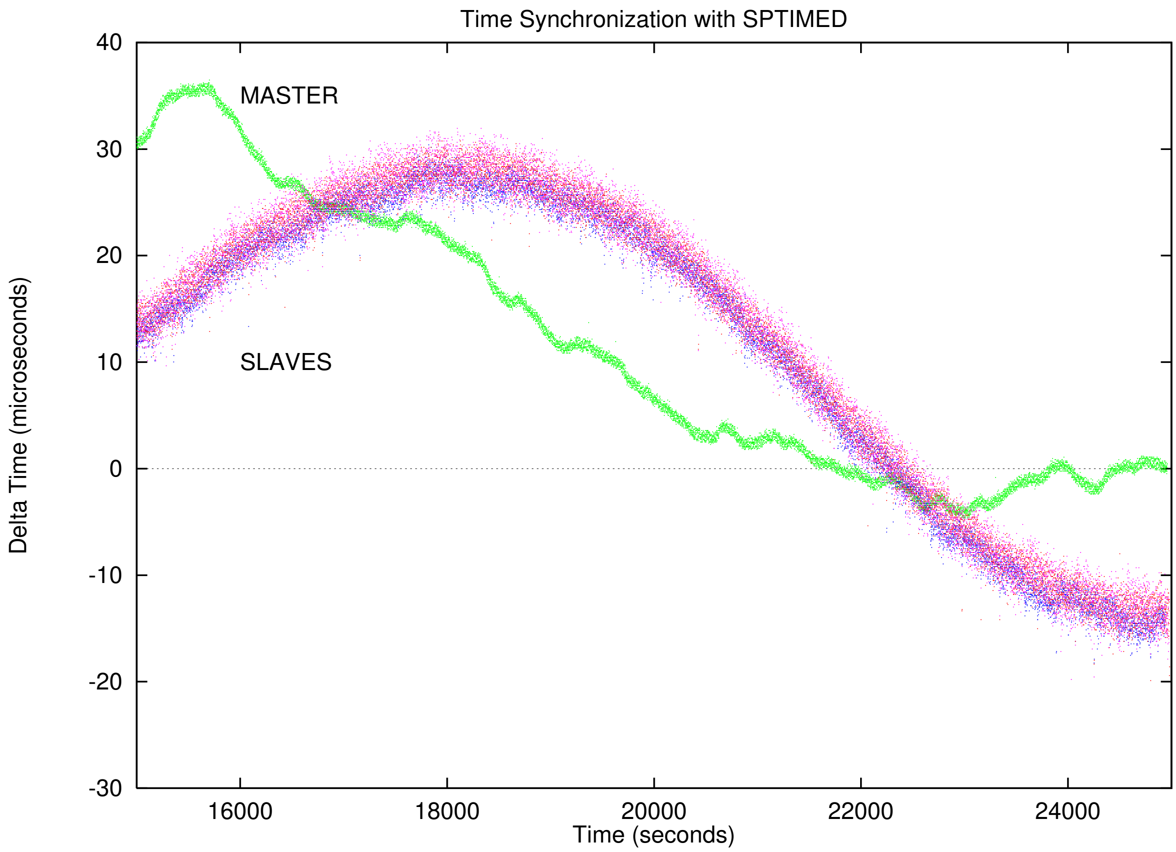

Figure 8: Synchronization with `sptimed`. The measurement is made on a cluster of four SP nodes. The scatter plot shows the difference between the node clock values on four nodes as a function of the synchronous network time for a 10000 second interval. The difference between the three slave node curves is typically within 5 microseconds. The least-square filter removes the "noise" of master node's clock as apparent from smoothing of the slave node curves. Slave clocks also follow the master loosely as expected.

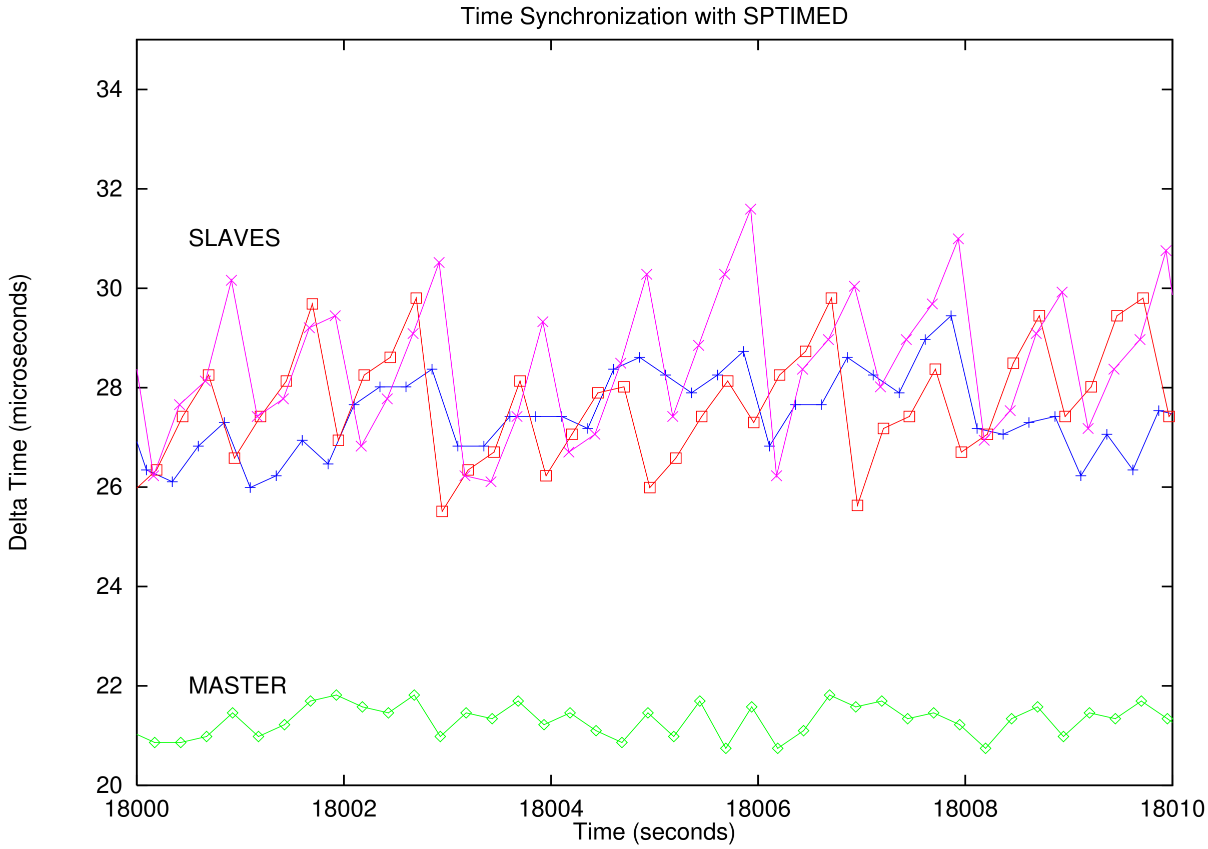

Figure 9: Synchronization with `sptimed`. The plot shows the difference between the node clock values on four nodes as a function of the synchronous network time for a 10 second interval. Periodic adjustments made once a second are apparent from the jagged form of the slave node curves.

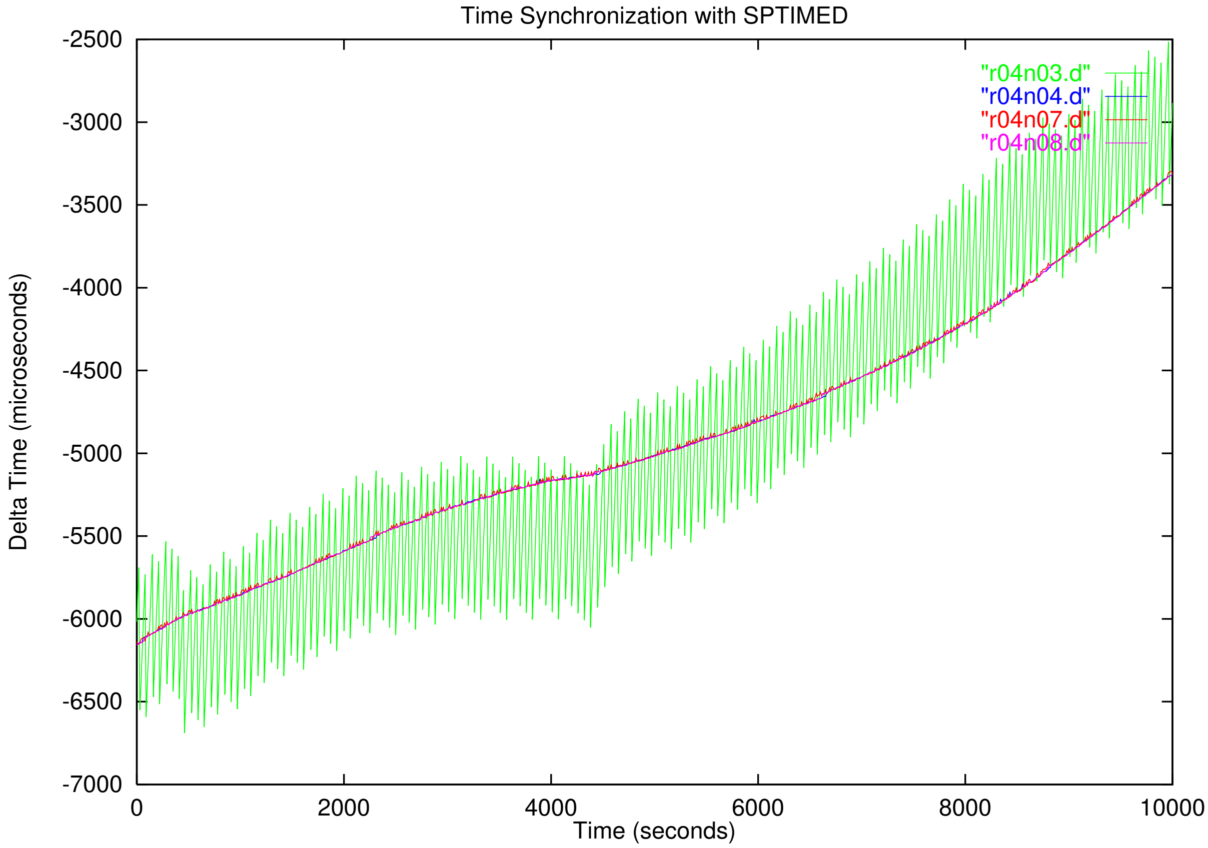

Figure 10: Synchronization with `sptimed` and the least–square filter in action. The master node is tracking the Internet time by NTP, observed here as the sawtooth shaped curve. The 3 other curves, overlapping each other and running through the master node curve, belong to the 3 tightly synchronized slave nodes. The least–square filter in the master `sptimed` daemon results in smoother slave node curves.

References

[1] C. B. Stunkel *et al*, "The SP2 High Performance Switch," *IBM Systems Journal*, vol. 34, no. 2, pp. 185–204, 1995.

[2] C. B. Stunkel, D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao, "The SP1 High-Performance Switch," in *Proc. 1994 Scalable High-Performance Computing Conference*, pp. 150–157, May 1994.

[3] C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, M. Tsao, and P. R. Varker, "Architecture and implementation of Vulcan," in *Proc. 8th Int. Parallel Processing Symp.*, pp. 268–274, April 1994.

[4] B. Abali and C. Aykanat, "Routing Algorithms for IBM SP1," *Lecture Notes in Computer Science, Springer–Verlag*, vol. 853, pp. 161–175, 1994.

[5] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[6] D. L. Mills, "Internet Time Synchronization: The Network Time Protocol," *IEEE Trans. Commun.*, vol. 39, pp. 1482–1493, Oct. 1991.

[7] D. L. Mills, "Precision Synchronization of Computer Network Clocks," *Computer Comm. Review*, vol. 24, pp. 28–43, Apr. 1994.

[8] D. L. Mills, "Improved Algorithms for Synchronizing Computer Network Clocks," *Computer Comm. Review*, vol. 24, pp. 317–327, Oct. 1994.

[9] D. L. Mills, "Improved Algorithms for Synchronizing Computer Network Clocks," *IEEE/ACM Trans. Netw.*, vol. 3, pp. 245–254, Jun. 1995.

[10] W. R. Stevens, *UNIX Network Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1990.

[11] C.-F. Wu, Y. H. Liu, C. D. Benveniste, C.-L. Chen, and W.-H. Chiang, "Trace-Based Analysis and Tuning for Distributed Parallel Applications," in *Proc. 1994 Int. Conf. on Parallel and Distrib. Sys.*, pp. 716–723, 1994.

[12] E. Maillet and C. Tron, "On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems," *J. Parallel Distrib. Comput.*, vol. 28, pp. 84–93, 1995.

[13] R. Mraz, "Reducing the Variance of Point–to–point Transfers for Parallel Real–Time Programs," *IEEE Parallel Distrib. Technol.*, vol. 2, no. 4, pp. 20–31, 1994.

[14] D. G. Feitelson and L. Rudolph, "Gang Scheduling Performance Benefits for Fine Grain Synchronization," *J. Parallel Distrib. Comput.*, vol. 16, pp. 306–318, Dec. 1992.

[15] G. J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons.* New York, NY: Wiley & Sons, 1987.

[16] I. D. Scherson and C.-H. Chien, "Least Common Ancestor Networks," in *Proc. 7th Int. Parallel Processing Symp.*, pp. 507–513, 1993.

[17] E. M. Reingold, J. Nievergelt, and N. Deo, *Algorithms and Complexity.* Englewood Cliffs, NJ: Prentice-Hall, 1977.

[18] D. N. Kimelman and T. A. Ngo, "The RP3 program visualization environment," *IBM Journal of Research and Development*, vol. 35, no. 5/6, pp. 635–651, 1991.

[19] C. E. Wu, J. P. Prost, and C. B. Benveniste, "The design of a timing simulator for distributed applications," in *Proc. 1992 Int. Conf. on Parallel and Distributed Systems*, pp. 50–57, Dec. 1992.

[20] H. Franke, C. E. Wu, P. Pattnaik, and M. Snir, "MPI programming environment for the IBM SP1/SP2," in *Proc. 1995 International Conference on Distributed Computing Systems*, IEEE, 1995.

[21] P. F. Corbett, D. G. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T. Morgan, , and A. Zlotek, "Parallel File Systems for the IBM SP Computers," *IBM Systems Journal*, vol. 34, no. 2, pp. 222–248, 1995.

[22] C. E. Wu, Y. H. Liu, and Y. Hsu, "Timestamp consistency and trace-driven analysis for distributed parallel systems," in *Proc. 9th Int. Parallel Processing Symp.*, April 1995.

[23] D. L. Mills. *private communication*, Mar. 1995.