Parallel and Distributed Simulation Systems

[illegible]

PARALLEL AND DISTRIBUTED SIMULATION SYSTEMS

Richard M. Fujimoto, PhD
Georgia Institute of Technology

A WILEY-INTERSCIENCE PUBLICATION

JOHN WILEY & SONS, INC.

New York • Chichester • Weinheim • Brisbane • Singapore • Toronto

CHAPTER 6

Time Parallel Simulation

As discussed in Chapter 2, one can view the simulation computation as determining the values of a set of state variables across simulation time. The approaches discussed thus far accomplish this task by partitioning the state variables defined in the simulation among a set of logical processes. They assign each LP the responsibility of computing the evolution of its state variables over the duration of the simulation. As illustrated in Figure 6.1*(a)*, this approach uses a *spatial decomposition* approach where the space-time diagram is partitioned into a set of horizontal strips, with each LP responsible for computing the values of the variables contained within that strip over simulation time.

Another approach is to use a *temporal decomposition* of the space-time diagram. Here, as shown in Figure *6.1(b)*, the space-time diagram is partitioned into a set of vertical strips, and a logical process is assigned to each strip. Each LP must perform a simulation of the entire system for the interval of simulation time covered by its strip of the space-time diagram.

Stated another way, the simulation computation constructs a *sample path* through the set of all possible states in which the system can reside across simulation time (see Fig. 6.2). A *time parallel* simulation partitions the simulation time axis into a sequence of nonoverlapping simulation time intervals $[T_0, T_1)$, $[T_1, T_2)$, ..., $[T_{n-1}, T_n)$. A logical process assigned to the *ith* window computes the portion of the sample path within that window.

This so-called time parallel approach to parallel simulation offers several attractive properties:

- *Massive parallelism*. The amount of parallelism in the simulation is potentially very large because simulations often extend over long periods of simulation time. Time parallel simulation algorithms typically run out of processors before they run out of parallelism within the simulation computation.
- *Independent logical processes*. Once a logical process begins the simulation of a vertical strip, it can proceed with this simulation independent of other logical processes, thereby avoiding expensive synchronization operations throughout much of the computation. Time parallel simulation algorithms typically only require coordination among the logical processes prior to beginning the computation of each strip. This is in sharp contrast to space-parallel algorithms

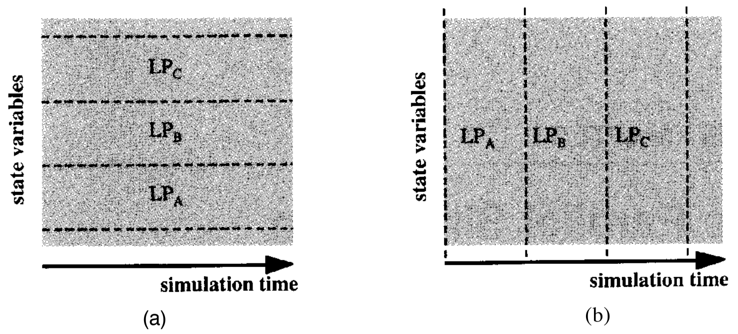


Figure 6.1 Space-time diagram of the simulation computation. *(a)* Space parallel approach; *(b)* time parallel approach.

which require continual synchronization among the logical processes throughout the entire computation.

The central problem that must be solved by the time parallel simulation algorithm is to ensure that the states computed at the "boundaries" of the time intervals match. This is referred to as the *state-matching problem*. Specifically, the state computed at the end of the ith interval must match the state at the beginning of the ith interval. But how can one compute the initial state of the ith interval without first performing the simulation computation for all prior intervals?

Several approaches to solving the state matching problem for specific simulation problems have been proposed. The three approaches to be discussed here are as follows:

- *Fix-up computations*. Logical process LP;, responsible for the ith time interval, "guesses" the initial state of the simulation in its time interval and performs a simulation based on this guess. In general, the final state computed for the **i - I** st interval will not match the initial guess, so LPi must perform a "fix-up"

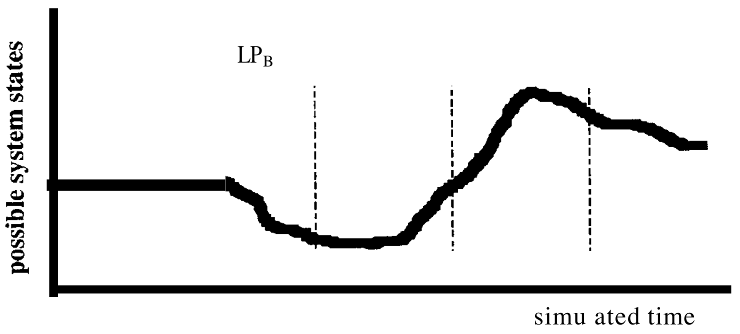


Figure 6.2 Sample path for a simulation computation.

computation to recompute the sample path for the ith interval using the final state computed by LP_{i-1} as the initial state for LP;. The fix-up computation may be to simply repeat the simulation using the new initial state. This process is repeated until the final state computed by each interval matches the initial state of the next interval. When a state match occurs across all of the intervals, the simulation is completed.
- *Precomputation of state at specific time division points*. It may be possible to determine the state of the simulation at specific points in simulation time without performing a detailed simulation of everything preceding that time. For example, as will be seen later, one may be able to guarantee a buffer will overflow in simulating a queue based on the rate of traffic entering the queue relative to the maximum rate of traffic departing. If this is the case, one may define the simulation time intervals so that the state of the simulation is known at the beginning of each time interval, thereby solving the state matching problem.
- *Parallel prefix computations*. If one can formulate the state of the simulation as a linear recurrence equation, a parallel prefix computation can be used to solve the equation over simulation time.

As will be seen momentarily, the solution to the state matching problem requires detailed knowledge of the simulation application and the statistics that are being computed. Thus time parallel simulation techniques do not provide a general approach to parallel or distributed simulation but rather define a methodology that can be applied to develop parallel simulation algorithms for specific simulation problems.

In the following we describe three time parallel simulation algorithms for simulating a cache memory system, an asynchronous transfer mode (ATM) multiplexer, and a G/G/1 queue.

6.1 TIME PARALLEL CACHE SIMULATION USING FIX-UP COMPUTATIONS

The time parallel simulation approach using fix-up computations executes the following steps:

1. Logical process LP_i is assigned an interval of simulation time $[T_i+_1, T_i)$ and selects an initial state $SO(T_{i-1})$ for its interval. More generally, $S^j(T_i)$ denotes the state of the system at simulation time T_i computed after the *jth* iteration (j=1,2, ...).
2. LP; simulates the system over the time interval $[T_{i-1}, T_i)$, computing a final state $S^j(T_i)$ for this interval. Each logical process can execute on a separate processor, and no interprocessor communications is required during this step.
3. LP_i sends a copy of the final state it just computed $S^j(T_i)$ to LPi+l'

4. If $S^j(T_{i-1})$ does not match the initial state $S^{j-1}(T_{i-1})$ used in the simulation for the interval $[T_{i-1}, T_i)$, then LP$_i$ sets its initial state to $S^j(T_{i-1})$ and recomputes the sample path for its interval. If during this recomputation the state of the new sample path that is being computed matches that of the previous iteration (i.e., $S^j(t)$ is identical to $S^{j-1}(t)$ for some simulation time t), the process can stop its computation because the remainder of its sample path will be identical to that derived from the previous computation.
5. Repeat steps 3 and 4 until the initial state used in each interval matches the final state computed for the previous interval for all logical processes.

This process is illustrated graphically in Figure 6.3. As shown in Figure *6.3(a)*, each logical process is assigned an interval in the simulation time axis, selects some initial state, and computes the sample path based on this initial state. The logical processes execute independent of each other during this phase. Only LPA computed a correct sample path because it is the only process that used the correct initial state (the initial state for the entire simulation). Each of the logical processes except LPA resets its initial state to the final state computed by the LP assigned the immediately preceding time window. In Figure *6.3(b)*, the new sample path converges to that

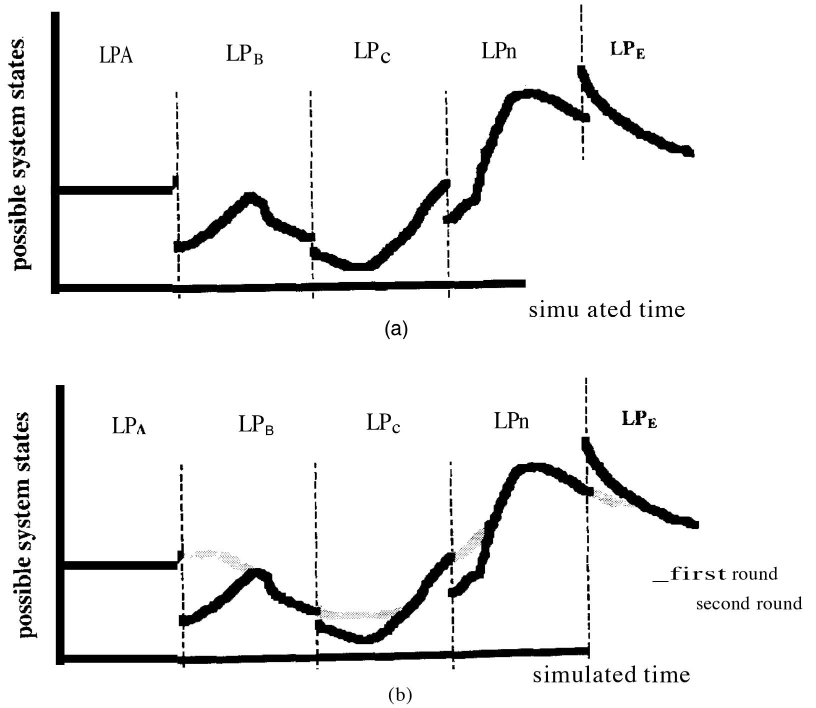


Figure 6.3 Time parallel simulation using fix-up computations. *(a)* Computation performed in first round; *(b)* fix-up computation in second round.

computed in the previous iteration for each logical process, so the computation completes after only two iterations. In general, however, more passes may be required to construct the complete sample path.

Using this approach, the computation for the first interval will always be completed by the end of the first iteration, the second interval computation will be completed after the second iteration, and so on. Thus, in the worst case N iterations will be required to complete the sample path for N intervals' or approximately the same time for the sequential execution if overhead computations required in the time parallel execution are negligible. The algorithm will perform well if the final state for each interval computation does not depend on the initial state. When this is the case (as in Fig. 6.3), the final state computed in the first iteration for each interval computation will be correct despite the fact that an incorrect initial state was used.

This time parallel simulation approach can be applied to simulating a cache memory in a computer system using a least recently used (LRU) replacement policy. A cache is a high-speed memory that holds recently referenced memory locations (data and instructions). The goal is to store frequently used data and instructions in the cache that can be accessed very quickly, typically an order of magnitude faster than main memory. Because the cache memory has a low access time, it is expensive, so the computer system may only contain a limited amount of cache memory. If data or instructions are referenced by the CPU that do not reside in the cache, the information must be loaded into the cache, displacing other data/instructions from the cache if there is no unused memory in the cache. Main memory is partitioned into a collection of fixed-size blocks (a typical block size is 64 bytes), and some set of blocks are maintained within the cache. The cache management hardware includes tables that indicate which blocks are currently stored in the cache.

The cache *replacement policy* is responsible for determining which block to delete from the cache when a new block must be loaded. A commonly used policy is to replace the block that hasn't been referenced in the longest time, based on the premise that recently referenced blocks are likely to be referenced again in the near future. This approach is referred to as the *least recently used* (LRU) replacement policy. Due to implementation constraints, cache memories typically subdivide the cache into *sets* of blocks, and use LRU replacement within each set.

Time parallel simulation can be effective in simulating cache memories, particularly those using LRU replacement because the final state of the cache is seldom dependent on the cache's initial state for "reasonably long" strings of memory references. This is because subsequent references will tend to load new blocks into the cache, eventually displacing the cache's original contents.

The input to the cache simulation is a sequence of memory references. Each reference indicates which block of memory is being referenced. The sequence is partitioned into N subsequences, one for each logical process (i.e., each processor) participating in the simulation.

The state of the cache is a list of the blocks that are currently stored in the cache. These blocks are stored in a data structure known as the LRU stack. When a block is referenced, the LRU stack is searched to determine if the block already resides in the cache. If it does, a cache "hit" is said to occur, and the block is removed and placed on top of the stack. Removing a block from the stack causes the blocks above it to move down one position in the stack, much like removing a tray from the middle of a stack of trays in a cafeteria. Thus blocks that have been referenced recently will be near the top of the stack, while those that have not been referenced recently tend to sink toward the bottom of the stack. The block that has not been referenced for the longest time, that is, the least recently used block, will be at the bottom of the stack.

If the referenced block is not in the stack, the block does not reside in the cache and a miss is said to occur. The block must now be loaded into the cache. To make room, the block at the bottom of the stack (the LRU block) is deleted from the stack, causing all blocks to slide down one position. The newly referenced block is placed on top of the stack.

For example, Figure 6.4 shows the execution of a time parallel simulation for a single set of a cache memory system containing four blocks. The addresses listed across the top of the figure indicate the blocks that are referenced by successive memory references. A time parallel simulation using three logical processes is used. Each LP initially assumes the cache is empty as its initial state. LPA first references blocks I and 2, with each causing a miss. Block I is then referenced again, causing it to be moved to the top of the LRU stack. Blocks 3 and 4 are referenced and loaded into the stack. When block 6 is referenced, the LRU block (block 2) at the bottom of the stack is deleted. The time parallel simulation divides the input trace of memory references into three segments, and each LP independently processes its trace, assuming that the cache is initially empty. Figure *6.4(a)* shows the sample path computed by each LP during the first round of the simulation.

In the second round of this computation LPA is idle because it correctly computed its portion of the sample path in the first pass. LP_B recomputes its sample path using the final state computed by LPA as the initial state for its cache. Similarly LPc

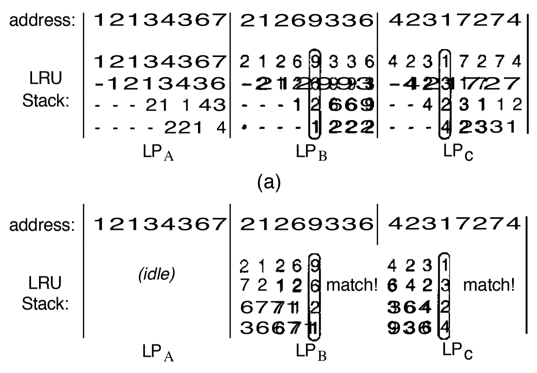


Figure 6.4 Example execution of time parallel cache simulation. *(a)* Execution during the first round using empty caches as the initial state; *(b)* execution during the second round.

recomputes its sample path using LPB's final state after the first iteration. After simulating the fifth memory reference in its trace, LP_B observes that the state of the cache is now identical to what it had computed after the fifth memory reference in the first round. Therefore the remainder of its sample path will be identical to that computed in the first round, so there is no need to recompute it. Similarly LPc's recomputation becomes identical to that computed in its first round after only four memory references, so it can also stop. LP_B and LP_C need only replace the first four and three stack states, respectively, in the first round simulation with the new values computed in the second round to reconstruct the entire sample path.

In general, the LRU replacement policy guarantees that if the number of blocks in the set is *k,* the state of the simulation will be independent of the initial state of the cache after *k* different blocks have been referenced. Thus, if each subsequence of memory references used by the logical processes references at least *k* different blocks, the time parallel simulation will require only two steps to compute the complete sample path.

6.2 SIMULATION OF AN ATM MULTIPLEXER USING REGENERATION POINTS

The points where the time axis was partitioned in the cache simulation could be made arbitrarily, so time intervals were defined with an equal number of memory references in each interval assigned to each logical process in order to balance the workload among the processors. The time parallel simulation algorithm described next selects the time division points at places where the state of the system can be easily determined. Specifically, the simulation time axis is broken at regeneration points; these are points where the system returns to a known state.

Asynchronous transfer mode (ATM)25 networks are a technology that has been developed to better support integration of a wide variety of communication service-voice, data, video, and faxes-all within a single telecommunication network. These so-called Broadband Integrated Services Digital Networks (B-ISDN) are expected to provide high bandwidth and reliable communication services in the future. Messages sent into ATM networks are first divided into *fixed-size* cells that then form the atomic unit of data that is transported through the network.

A multiplexer, depicted in Figure 6.5, is a component of a network that combines (concentrates) several streams of incoming traffic (here, ATM cells) into a single output stream. The incoming lines might represent phone lines to individual customers, while the out-going line represents a high-bandwidth trunk line carry traffic for many customers. The bandwidth of the outgoing line is usually smaller than the sum of the bandwidths of the incoming lines. This means that cells will accumulate in the multiplexer if the total incoming traffic flow exceeds the capacity of the output link. For this purpose the multiplexer contains a certain amount of

25 An unfortunate acronym. ATM networks are not to be confused with automated teller machines used by banks!

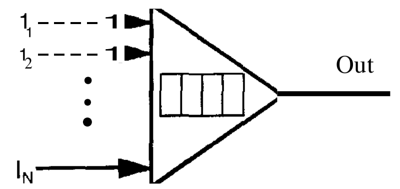


Figure 6.5 An ATM multiplexer with N inputs. The circuit contains a fixed sized buffer. Data (cells) are lost if the buffer overflows.

buffer memory to hold cells waiting to be sent on the outgoing link. To simplify its design, if the buffer memory overflows (i.e., a new cell arrives but the outgoing link is busy and there is no unused buffer space in which to store the cell), the cell is simply discarded. A goal in designing a multiplexer is to provide sufficient buffer memory to ensure that the expected number of lost cells is below some design goal, typically on the order of only one lost cell per 10^9 successfully transmitted cells. Since cell losses are rare, very long simulation runs are required to capture a statistically significant number of cell losses.

Assume that each of the incoming links can transmit B cells per second, and the outgoing link is of bandwidth C x B cells per second, where C is an integer. Link bandwidths are normalized so that the input link has unit bandwidth, and the output link has bandwidth C. Here, the unit of time that is used is the amount of time required to transmit a single cell over an input link. This quantity is referred to as a *cell time*. C cells may be transmitted by the output link during a single cell time.

A traffic generator (called a source) transmits cells on each incoming link. A separate source is attached to each incoming link of the multiplexer. The traffic produced by each source can be viewed as a sequence of "on" and "off" periods. When the source is on, it transmits cells, one per cell time, over the link. When the source is off, no cells are transmitted. This behavior is depicted in Figure 6.6.

The stream of incoming cells to the multiplexer can be characterized as a sequence of tuples $(Ai' \delta_i)$ (i = 1,2,3, ...), where Ai denotes the number active or "on" (transmitting cells) sources, and δ_i denotes the length of time that exactly this number of sources remain active. For example, in Figure 6.6 the input is characterized by the tuples (1,4), (4,2), (3,4), and so on, meaning initially one source is active for four units of time, then four are active for two units of time, then three for four units of time, and so on.

The simulation problem is formulated as follows: Consider a multiplexer with N input links of unit capacity, an output link with capacity C, and a FIFO queue containing K buffers, with each buffer able to hold a single cell. Any cells arriving when the queue is full are discarded. The simulation must determine the average utilization and number of discarded cells for incoming traffic characterized by the sequence of tuples $(Ai' \delta_i)$, as depicted in Figure 6.6.

Let T_i denote the simulation time corresponding to the end of the ith tuple, with To denoting the beginning of the simulation which is equal to zero. The state of the simulation model at any instant of simulation time is simply the number of cells

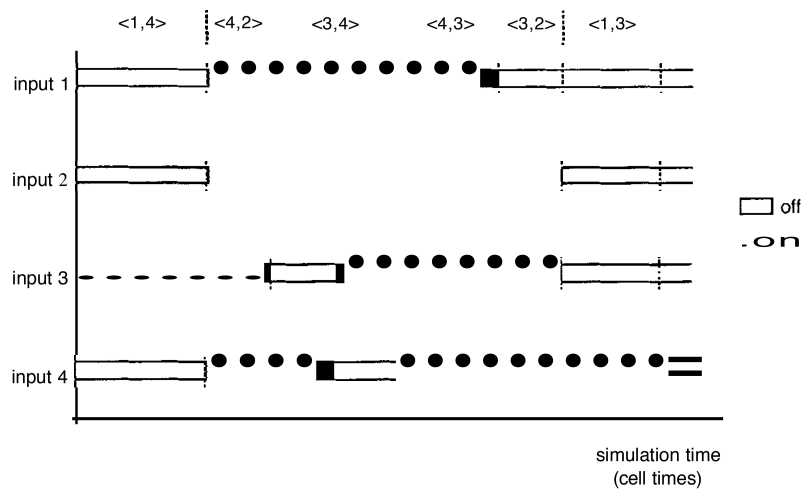


Figure 6.6 Onjoffperiods of four sources. The input to the multiplexer is characterized by a sequence of tuples $(Ai' \delta_i)$, where A indicates the number of on sources and δ_i indicates the duration (in cell times) that this number of sources are active.

stored in the queue. Let $Q(T_i)$ denote the number of cells stored in the queue at time T_i. Let $S(T_i)$ denote the total number of cells serviced (transmitted on the output link) and $L(T_i)$ denote the number of cells lost up to time T_i. The notation used for this simulation is summarized in Table 6.1. All of these quantities are integers.

During each time interval the multiplexer will be in one of two possible situations:

1. *Underload:* $Ai \leq$ C. The number of active sources is less than or equal to the output link capacity; that is, the rate that cells are flowing into the multiplexer is less than or equal to the outgoing rate, so the length of the FIFO queue (Qi) is either decreasing or remains the same. No cell losses can occur during an underload period.

TABLE 6.1 Symbols used in multiplexer simulation

N	Number of input links
C	Capacity of the output link
K	Number of buffers
$A;$	Number of active sources during ith time interval
δ_i	Length of ith time interval
$T;$	Time marking the end of the ith interval
$Q(T;)$	Length of the queue at time $T;$; $Q(T_o) = 0$
$SeT;)$	Total number of cells transmitted on output at time $T;$; $S(To) = 0$
$L(T_i)$	Total number of cell lost up to time $T;$; $L(T_o) = 0$

2. *Overload:* $Ai > C$. The rate of cells entering the multiplexer exceeds the capacity of the outgoing link, so the length of the FIFO queue is increasing. All cell losses occur during overload periods.

During each time unit Ai cells are received on the input links and C cells are transmitted on the output link. During an overload period the queue fills at a rate of $(Ai - C)$ cells per unit time, unless the queue becomes full. If the queue becomes full, the queue length remains fixed at K, and the additional cells that could not be placed into the queue are lost. Conversely, during an underload period the queue is drained at a rate of $(C - Ai)$ cells per unit time, unless the queue becomes empty. Thus the length of the queue can be computed as

$$\begin{aligned} Q(T_i) &= min(Q(T_{i-i}) + (Qi - C)\delta_i, K) && \text{if} Ai > C \text{ (overload)} \\ &= max(Q(T_{i-i}) - (C - A_i)\delta_i, 0) && \text{if } Ai \le C \text{ (underload).} \end{aligned} \tag{6.1}$$

The number of serviced and lost cells can be computed by accounting for all cells during each time period, which is the time represented by a single tuple. Specifically, at the beginning of the ith time period there are $Q(Ti=i)$ cells buffered in the multiplexer. During this time period $A_i\delta_i$ additional new cells are received by the multiplexer. At the end of the time period, there are $Q(T_i)$ cells remaining. The difference between these two quantities, $(Q(T_{i-i}) + A_i\delta_i) - Q(T_i)$, corresponds to cells that were either serviced or lost during the time period.

First, let us compute the number of serviced cells. During an underload period no cells are lost. Thus all of the $(Q(T_{i-i}) + A_i\delta_i) - Q(T_i)$ cells derived in the previous paragraph represent cells that were serviced. Now consider an overload period. Observe that the output link transmits C cells per unit time so long as the queue is not empty. Because the queue cannot become empty during an overload period, the number of serviced cells increased by $C\delta_i$. In other words,

$$\begin{aligned} S(T_i) &= S(T_{i-i}) + C\delta_i && \text{if } Ai > C \text{ (overload)} \\ &= S(T_{i-i}) + ((Q(T_{i-1}) + A_i\delta_i) - Q(T_i) && \text{if } AiC \text{ (underload).} \end{aligned} \tag{6.2}$$

The utilization of the output link is computed as the total number of cells serviced during the simulation divided the number of cells that could have been serviced during the length of the simulation. Specifically, if the simulation ends at time T_M (i.e., the simulation includes M intervals, or M tuples), the output link utilization is $S(TM)/CT_M$.

Now consider lost cells. No cells are lost during underload. During overload, the number of cells either lost or serviced is $(Q(T_{i-1}) + A_i\delta_i) - Q(T_i)$, as discussed earlier. The number of serviced cells is easily computed because as was observed in computing $S(t_i)$, C cells will be serviced per unit time during overload. The

difference between these two quantities is the number of lost cells during the ith time period. Thus we have

$$\begin{aligned} L(T_i) &= L(T_{i-i}) + (Q(T_{i-i}) + A_i\delta_i) - Q(T_i) - C\delta_i && \text{if } Ai > C \text{ (overload)} \\ &= L(T_{i-i}) && \text{if } Ai \le C \text{ (underload).} \end{aligned} \tag{6.3}$$

These equations for computing $Q(T_i)$, $S(T_i)$, and $L(T_i)$ enable one to simulate the behavior of the multiplexer. To perform a *time parallel* simulation of the multiplexer, the sequence of tuples is partitioned into P subsequences, where P is the number of processors available to perform the simulation, and a subsequence is assigned to each one. The principal question that must be answered is again the state-matching problem, or here, What is the initial state (the initial length of the queue) for each subsequence assigned to each processor?

This state-matching problem can be solved if the length of the queue at certain points in simulation time can be computed during a (parallel) precomputation phase. One could then partition the tuple sequence at points in time where the state of the multiplexer is known. Two key observations are used to determine the state of the multiplexer at specific points in time:

1. *Guaranteed overflow.* Consider a tuple defining an overload period. If the length of the overload period is sufficiently long that even if the queue were empty at the beginning of the tuple's period, the buffer is guaranteed to be full by the end of the period, then it would be known that the length of the queue is K at the end of the tuple period. A tuple with this property is referred to as a *guaranteed overflow* tuple.
2. *Guaranteed underflow.* Similarly consider a tuple defining an underload period. In this case the queue is being drained. If the duration of the tuple is sufficiently long that the queue will be empty at the end of the tuple's period even if the queue were full at the beginning of the period, then the tuple is said to be a guaranteed underflow tuple, and the queue must be empty at the end of the tuple's period.

More precisely, the conditions for $\langle Ai' \delta_i \rangle$ to be a guaranteed overflow (underflow) tuple are

Guaranteed overflow:

$$\begin{aligned} &\text{if} && (Ai - C)\delta_i \ge K, \\ &\text{then} && Q(T_i) = K. \end{aligned} \tag{6.4}$$

Guaranteed underflow:

$$\begin{aligned} &\text{if} && (C - A_i)\delta_i \ge K, \\ &\text{then} && Q(T_i) = 0. \end{aligned}$$

The time parallel simulation algorithm for the ATM multiplexer operates as follows. Given a sequence of tuples (A_i, δ_i), $i = 1,2, \ldots$:

1. Identify a set of guaranteed overflow or underflow tuples G by applying conditions (6.4). This can be accomplished by assigning an equal length subsequence to each processor, and by having each processor search from the beginning of its subsequence for a guaranteed overflow or underflow tuple. The tuples in G define the time division points at which the simulation time axis is broken, as shown in Figure 6.2.
2. For each processor, if the ith tuple is found to be a guaranteed overflow tuple, set $Q(T_i)$ to K; if the ith tuple is a guaranteed underflow tuple, set $Q(T_i)$ to 0.
3. For each processor, compute $Q(T_i)$, $S(T_i)$ and $L(T_i)$ using equations (6.1), (6.2), and (6.3) defined above for each tuple, starting with the tuple following the guaranteed overflow (or underflow) tuple. Assume that S and L are initially zero for each subsequence; that is, each processor only computes the number of serviced and lost cells for the subsequence assigned to it (not a cumulative total for the entire simulation). When this has been completed, send the queue length at the end of the subsequence to the processor assigned the next subsequence. Upon receipt of this information, the processor can simulate the tuples assigned to it that preceded the guaranteed overflowlunderflow tuple.
4. Compute the total number of serviced and lost cells by summing the values computed for these quantities by each processor.

This algorithm relies on being able to identify a guaranteed overflow or underflow tuple in the subsequence assigned to each processor. The algorithm fails if any processor does not locate a guaranteed underflow or overflow tuple. In general, it is impossible to guarantee such a tuple will be found. In practice, because cell losses are so rare, there will usually be an abundance of guaranteed underflow tuples. An alternative approach to this problem is to examine short *sequences* of tuples in order to identify a sequence that results in a guaranteed underflow (overflow), even though individual tuples within the sequence could not be guaranteed to result in an underflow (overflow). There is again no guarantee, however, that such a sequence can always be identified.

The central advantage of this algorithm compared to the approach described in the previous section for cache memories is no fix-up computation is required. The central disadvantages are the need for a precomputation to compute the time division points, and the possibility the algorithm may fail if such time division points cannot be identified.

6.3 SIMULATION OF QUEUES USING PARALLEL PREFIX

A third approach to time parallel simulations utilizes parallel prefix computations to determine the state of the simulation across simulation time. A *prefix computation*

computes the N initial products of N variables $X_1, X_2, \ldots, X_N$:

$$\begin{aligned} P_1 &= X_1 \\ P_2 &= X_1 * X_2 \\ P_3 &= X_1 * X_2 * X_3 \\ &\vdots \\ P_N &= X_1 * X_2 * \cdots * X_N \end{aligned}$$

where the asterisk (*) is an associative operator. This set of equations can be rewritten more compactly as the linear recurrence $P_i = P_{i-1} * X_i$, $i = 1, 2, \ldots, N$, where P_0 is the identity element. As will be discussed momentarily, prefix computations are of interest because efficient algorithms exist for performing these computations on a parallel computer.

The simulation of a *G/G/1* queue[26] where the service time does not depend on the state of the queue can be recast as a prefix computation. Specifically, let r_i denote the interarrival time of the ith job at the queue, and s_i denote the service time assigned to the ith job. These values can be trivially computed in parallel because they are independent random numbers. The simulation computation must compute the arrival time of the ith job A_i, and the departure time of the ith job D_i for $i = 1,2, \ldots, N$. The arrival time of the ith job can immediately be rewritten as a linear recurrence:

$$A_i = A_{i-1} + r_i \; (= r_1 + r_2 + \cdots + r_i),$$

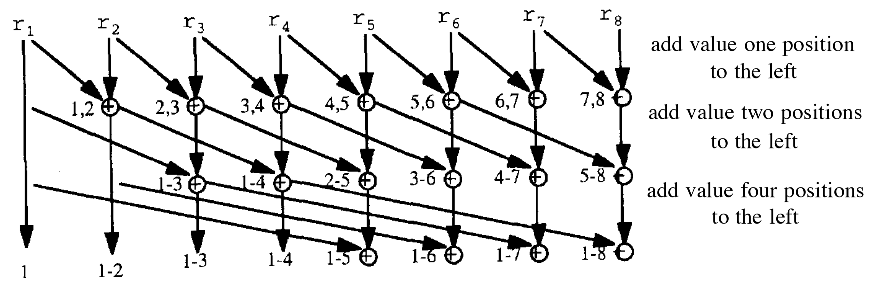
so it can be immediately solved using a parallel prefix computation. The departure times can also be written as a linear recurrence. Specifically, the ith job begins service either when it arrives, or when the i-1st job departs, which ever is later. Thus $D_i = \max(D_{i-1}, A_i) + s_i$. This can be rewritten as the following linear recurrence:

$$\begin{pmatrix} D_i \\ 0 \end{pmatrix} = \begin{pmatrix} s_i & A_i + s_i \\ -\infty & 0 \end{pmatrix} \cdot \begin{pmatrix} D_{i-1} \\ 0 \end{pmatrix},$$

where the matrix multiplication is performed using max as the additive operator (with identity $-\infty$), and + as the multiplicative operator (identity 0). Rewriting the departure time equation in this form puts it into the proper format for a parallel prefix computation.

Because the simulation computation can be specified as a parallel prefix computation, the only question that remains concerns the parallel prefix computation itself. The parallel prefix for N initial products can be computed in $O(\log N)$ time. Consider computation of the arrival times A_i. Suppose that the N data values r_1,

[26] The notation G/G/1 means a general distribution is used to select the interarrival time and service time of jobs arriving at the queue, and the 1 denotes the fact that there is one server.



r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8
1,2 2,3 3,4 4,5 5,6 6,7 7,8
1-3 1-4 2-5 3-6 4-7 5-8
1 1-2 1-3 1-4 1-5 1-6 1-7 1-8
add value one position to the left
add value two positions to the left
add value four positions to the left

Figure 6.7 Binary tree for performing a parallel prefix computation.

rz, ... , rN are assigned to different processors. The ith initial partial product *Ai* can be computed in parallel by defining a binary tree, as shown in Figure 6.7. **In** the first step, each data value *ri* is added to the data value one position to the left *(ri-l)'* **In** the next step, a new cumulative sum is formed by adding in the value two elements to the left, then four to the left, eight, and so on. If each processor repeats these steps, all *N* initial products will be performed in rlogNl steps, as shown in Figure 6.7.

More precisely, the parallel prefix computation is defined as follows:

```
FOR j=O to rlog Nl - 1 DO
  FOR ALL i E (2^j+1, 2^j+2, ..., N) DO IN PARALLEL
    NewR[i] .= r[i-2^j] + r[i];
  END-FOR ALL
  FOR ALL i E (2^j+1, 2^j+2, ..., N) DO IN PARALLEL
    r[i) := NewR[i);
  END-FOR ALL
END-FOR
```

The FOR ALL statement performs the iterations of the loop in parallel, one iteration per processor. The second FOR ALL loop is used to copy intermediate results back into the r array for the next iteration. When the above program completes, r [i] will hold the arrival time for the ith job.

In practice, there will usually be many more partial products than processors, so one would aggregate groups of the values onto individual processors and perform computations involving data values on the same processor sequentially. This will improve the efficiency of the parallel algorithm because there will be more local computation between interprocessor communications, which are time-consuming relative to the time to perform an addition.

6.4 SUMMARY

Time parallel algorithms are currently not as robust as space parallel approaches because they rely on specific properties of the system being modeled, such as

specification of the system's behavior as recurrence equations and/or a relatively simple state descriptor. This approach is currently limited to a handful of important applications, such as queuing networks, Petri nets, cache memories, and statistical multiplexers. Space parallel simulations offer greater flexibility and wider applicability, but concurrency is limited to the number of logical processes. **In** some cases both time and space parallelism can be used together.

Time parallel algorithms do provide a form a parallelization that can be exploited when there isn't spatial parallelism available in the application. The three examples described in this chapter are all examples where there is very little space parallelism in the original simulation problem. The fact that the time parallel algorithms can exploit massive amounts of parallelism for these problems highlights the utility of the time parallel approach, provided suitable algorithms can be developed.

6.5 ADDITIONAL READINGS

Time parallel simulation for trace-driven simulations is described in Heidelberger and Stone (1990); this is perhaps the first proposal for using this technique. Extensions to this method to simulate caches are described in Nicol, Greenberg et al. (1992). The algorithm using regeneration points to simulate ATM multiplexers is described in Andradottir and Ott (1995) and Fujimoto, Nikolaidis et al. (1995), and extension of this method to simulate cascaded multiplexers is described in Nikolaidis, Fujimoto et al. (1994). Time parallel simulation of queues using parallel prefix algorithms were first reported in Greenberg, Lubachevsky et al. (1991). Related work in using time parallel simulation to simulate queues and Petri networks are described in Lin and Lazowska (1991), Ammar and Deng (1992), Wang and Abrams (1992), and Baccelli and Canales (1993). Other algorithms have been proposed to simulate telephone switching networks (Gaujal, Greenberg et al. 1993) and Markov chains (Heidelberger and Nicol 1991).