A Real-Time Traffic Simulation Using a Communication Latency Hiding Parallelization

Anthony Theodore Chronopoulos, *Senior Member, IEEE*, and Charles Michael Johnston

***Abstract*—This paper implements and analyzes a highway traffic-flow simulation based on continuum modeling of traffic dynamics. A traffic-flow simulation was developed and mapped onto a parallel computer architecture. Two algorithms (the one-step and two-step algorithms) to solve the simulation equations were developed and implemented. Tests with real traffic data collected from the freeway network in the metropolitan area of Minneapolis, MN, were used to validate the accuracy and computation rate of the parallel simulation system (with 256 processors). The execution time for a 24-h traffic-flow simulation over a 15.5-mi freeway, which takes 65.7 min on a typical single-processor computer, took only 88.51 s on the nCUBE2 and only 2.39 s on the CRAY T3E. The two-step algorithm, whose goal is to trade off extra computation for fewer interprocessor communications, was shown to save significantly on the communication time on both parallel computers.**

***Index Terms*—Communication latency hiding, real-time, simulation, traffic, two-step lax algorithm.**

I. Introduction

A VERY important component of the Intelligent Highway System is a traffic simulation system. Such a system uses computer hardware and software to simulate traffic on freeways and arterial networks. Input/output (I/O) devices provide real-time traffic data measurements from a network of traffic detectors (loops or cameras). The system uses a mathematical traffic-flow model to perform traffic-flow simulation and predict the traffic conditions in real time. These predictions can be used for real-time traffic control and vehicle guidance.

Fig.1 shows an architecture for such a traffic simulation system. It consists of three layers: a data layer, a simulation layer, and an analysis layer. The data layer is responsible for system I/O. It imports current traffic data from roadway sensors and provides access to databases containing roadway layout information. Above this, in the simulation layer, the traffic flow model is implemented to compute current and future traffic flow along the highway. At the top, the analysis layer combines simulation results with traffic management heuristics to make decisions about how best to control and/or route highway traffic. This information is then exported to both the traffic control systems and vehicle guidance systems via the data layer.

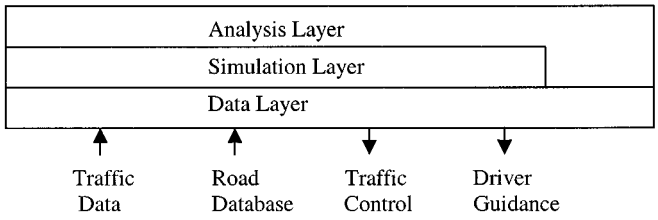

Fig. 1. Simulation system architecture.

Macroscopic or continuum traffic-flow models based on traffic density, volume, and speed have been proposed and analyzed in the past (see, for example, [1]–[6] and the references therein). These models involve partial differential equations defined on appropriate domains with suitable boundary conditions, which describe various traffic phenomena and road geometries.

The main objective of this paper is to demonstrate that the parallelization of the traffic-flow simulation component in a real-time system is feasible for macroscopic models. Some preliminary results on the issue of parallelizing computational fluid dynamics (CFD) methods for transportation problems were presented in [5]. Such a real-time simulation system can be designed using a parallel computer as its computational component. We design such a computational component by parallelizing a CFD method to solve the momentum conservation (macroscopic) model [4]–[6] and implementing it on the nCUBE2 and Cray T3E parallel computers.

Tests with real data from the I-494 freeway in Minneapolis, MN, were conducted. We run the simulations on a nCUBE2 and on a Cray T3E parallel computer. Processing elements (PEs) of the nCUBE2 are proprietary processors with rate of 20 MHz. Processors on the Cray T3E are DEC Alpha 21 164s with a clock speed of 450 MHz. Tests were run on the nCUBE2 at Sandia National Laboratory, and on the Cray T3E at the NPACI San Diego Supercomputing Center and at the Pittsburgh Supercomputing Center. The execution time for a 24-h traffic-flow simulation of a 15.5-mi freeway, which takes 65.65 min of computer time (on a 133-MHz single-processor Pentium computer), took only 88.51 and 2.39 s on the parallel traffic simulation systems implemented on nCUBE2 and Cray T3E, respectively.

We adopted the lax-momentum traffic model [11]. We next outline the continuous model equations. Let $k(x,t)$ and $q(x,t)$ be the traffic density and flow, respectively, at the space–time point (x,t). The generation term $g(x,t)$ represents the number

Manuscript received June 2000; revised October 2001. This work was supported in part by a Cray/SGI Grant, by the National Science Foundation (NSF) under ASC-9 634 775, by NSF under Cooperative Agreement ACI-9 619 020 through computing resources provided by the National Partnership for Advanced Computational Infrastructure, University of California San Diego, by NASA under Grant NAG 2-1383, and by Texas Advanced Research/Advanced Technology under Program ATP 003 658-0442-1999.

A. T. Chronopoulos is with the Division of Computer Science, University of Texas San Antonio, San Antonio, TX 78240 USA (e-mail: atc@cs.utsa.edu).

C. M. Johnston is with Concurrent Computer Corporation, Southfield, MI 48034 USA.

Publisher Item Identifier S 0018-9545(02)02490-8.

of cars entering or leaving the traffic flow in a freeway with entries/exits. The traffic flow, density, and speed are related by

$$q = k \cdot u.$$

The relaxation time T is assumed to vary with density k according to

$$T = t_0 \left(1 + \frac{rk}{k_{\text{jam}} - rk}\right)$$

where $t_0 > 0$ and $0 < r < 1$ are model parameters. The model equations are

$$\frac{\partial \vec{U}}{\partial t} + \frac{\partial \vec{E}}{\partial x} = \vec{Z}$$

where $\vec{U}, \vec{E}$, and $\vec{Z}$ are the following vectors in discretized form

$$\vec{U} = \begin{pmatrix} k \\ q \end{pmatrix}$$
$$\vec{E} = \begin{pmatrix} ku \\ u^2 k + \frac{v}{\beta+2} k^{\beta+2} \end{pmatrix}$$
$$\vec{Z} = \begin{pmatrix} g \\ \frac{k}{T}[u_f(x) - u] + gu \end{pmatrix}.$$

A typical choice of parameters is $u_f = 60, k_{\text{jam}} = 180, \beta = -1, v = 180, T = 50$, and $r = 0.80$. These parameters depend not only on the geometry of the freeway but also on the time of day and even on weather conditions.

We next outline the discrete model. Let Δt and Δx be the time and space mesh sizes. We use the following notation: k_j^i is the density (vehicles/mile/lane) at space node $j\Delta x$ and at time $i\Delta t$, q_j^i is the flow (vehicles/hour/lane) at space node $j\Delta x$ and at time $i\Delta t$, and u_j^i is the speed (mile/hour) at space node $j\Delta x$ and at time $i\Delta t$. At time $(i+1)\Delta t$, the density value k_j^{i+1} and volume value q_j^{i+1} are computed directly from the density and volume at the preceding time step i

$$\vec{U}_j^{i+1} = \frac{\vec{U}_{j+1}^i + \vec{U}_{j-1}^i}{2} - \frac{\Delta t}{\Delta x} \frac{\vec{E}_{j+1}^i - \vec{E}_{j-1}^i}{2} + \frac{\Delta t}{2}\left(\vec{Z}_{j+1}^i + \vec{Z}_{j-1}^i\right).$$

The method is of first-order accuracy with respect to Δt, i.e., the error is $O(\Delta t)$. To maintain numerical stability, time and space step sizes must satisfy the Courant–Friedrichs–Lewy (CFL) condition $(\Delta x/\Delta t) > u_f$, where u_f is the free-flow speed (see [4]). Typical choices for the space and time meshes of $\Delta x = 100$ ft and $\Delta t = 0.5$ s are recommended for numerical stability [6], [11].

Let P be the number of processors available in the system. The parallelization of the discrete model is obtained by partitioning the space domain (freeway model) into equal segments $\text{Seg}_0, \ldots, \text{Seg}_{P-1}$ and assigning each segment to the processors (PEs) $P_{j_0}, \ldots, P_{j_{p-1}}$. The choice of indexes $j_0, \ldots, j_{P-1}$ defines a mapping of the segments to the processors [3], [11].

The computations associated with each segment have as their goal to compute the density, volume, and speed over that segment. The computation in the time dimension is not parallelized.

TABLE I
PE ALLOCATION USING METHOD I

Total PEs (P)	Used PEs	Idle PEs	r_h or r_l	*rem*
1	1	0	426	0
2	2	0	213	0
4	4	0	106	2
8	8	0	53	2
16	16	0	27	21
32	31	1	14	6
64	61	3	7	6
128	107	21	4	2
256	213	43	2	0
512	426	86	1	0

At a fixed discrete time, this essentially means that the quantities k_j^i, q_j^i, and u_j^i are computed by processor p_{j_k}, iff the space node $j\Delta x$ belongs to the segment Seg_{j_k}. This segment-processor mapping must be such that the communication delays for data exchanges, required in the computation, are minimized. This segment-processor mapping is a linear array mapping onto a hypercube (nCUBE) and Torus (T3E).

The contents of this paper are organized as follows. In Section II, a processor allocation scheme is presented. In Sections III, the parallelization of the traffic flow model is discussed. In Sections IV and V, the implementation of the model on the nCUBE2 and the Cray T3E is described. In Section VI, the simulation tests are shown. In Sections VII–IX, a performance study of the simulations is presented. In Section X, conclusions are drawn.

II. PE SCHEDULING

All the space nodes in the simulation must be allocated to PEs. This is also known as PE *scheduling*. For a given number of PEs, P, it is desirable to allocate them in such a way that all PEs are utilized in the most efficient way possible, while keeping in mind any load-balancing requirements. Assume that there are n space nodes. Then ideally, we allocate $r = (n/p)$ nodes per PE. Obviously, the closest we can come is an allocation of $r_h = \lceil (n/p) \rceil$ (ceiling operation) or $r_l = \lfloor (n/p) \rfloor$ (floor operation), where "floor" equals the integer division and "ceiling" equals "floor"+1 if n/p is greater than r_l.

Our first approach to this problem we call Method I: allocate r_h or r_l space nodes to as many PEs as possible, with the remainder going to the last utilized PE. This algorithm has some side effects. Namely, as P gets larger, so does the number of idle PEs. As an example, consider the case where $n = 426$ and P is an increasing power of two. We are faced with the situation in Table I, where *rem* is the number of space nodes that the last PE will need to process (i.e., the number remaining after the initial r_h or r_l distribution).

This scenario is clearly not desirable from a load-balancing perspective. It does, however, free up some PEs for some other potentially useful work. In a real-time environment, these PEs could be doing I/O, or applying algorithms to the simulation results in order to effectively manage the traffic flow. However,

at the largest P values, where the lowest run-times are obtained, nearly 17% of the PE's are idle—an unreasonably large number. To obtain better load balancing, a second scheduling method was developed that was used in our implementation.

Method II seeks to improve load balancing by distributing the space nodes as evenly across the PEs as possible. This will require some combination of r_h and r_l. If h and l are the number of PEs allocated r_h and r_l space nodes, respectively, then we know

$$r_h = \left\lceil \frac{n}{P} \right\rceil \quad (1)$$

$$r_l = \left\lfloor \frac{n}{P} \right\rfloor \quad (2)$$

$$h + l = P \quad (3)$$

$$r_h \times h + r_l \times l = n. \quad (4)$$

Given (1) and (2), we combine (3) and (4) to solve for l and h and obtain

$$l = r_h \times P - n$$
$$h = P - l \,(\text{from } 3).$$

Therefore, l PEs each allocate r_l space nodes, and h PEs each allocate r_h space nodes.

In addition to knowing how many space nodes to allocate to each PE, we must devise a mapping of a subset of space nodes to each PE. Given that each PE knows its identity via a uniquely valued parameter $P_i, i = 0 \ldots P-1$, then the mapping is defined via the following pseudocode:

```
if P_i < h then
  upstream-node = P_i × r_h + 1
  downstream-node = (P_i + 1) × r_h
else
  temp = r_h × h
  offset = P_i − h
  upstream-node = offset × r_l + 1 + temp
  downstream-node = (offset + 1) × r_l + temp
end if
```

We introduce here the concept of *upstream* and *downstream* space nodes. If we think of the traffic as moving from upstream to downstream, then an upstream node is the leftmost or lowest index node within a segment, and the downstream node is the rightmost or highest index node within a segment.

Note that the first h PEs were chosen to allocate r_h space nodes. It could just as easily have been the first l PEs allocating r_l space nodes. The following example helps clarify this procedure:

Given $P = 3$ and $n = 16$, then
$r_l = \lfloor 16/3 \rfloor = 5 \;\; r_h = \lceil 16/3 \rceil = 6$
$l = 6 \times 3 - 16 = 2 \;\; h = 3 - 2 = 1$
$\cdots$ two PE's allocate 5 space nodes, and one PE allocates 6.
For P_0: upstream-node $= 0 \times 6 + 1 = 1$
  downstream-node $= (0+1) \times 6 = 6$
P_1: offset $= 1 - 1 = 0$
  temp $= 6 \times 1 = 6$
  upstream-node $= 0 \times 2 + 1 + 6 = 7$
  downstream-node $= (0+1) \times 5 + 6 = 11$
P_2: offset $= 2 - 1 = 1$
  temp $= 6 \times 1 = 6$
  upstream-node $= 1 \times 5 + 1 + 6 = 12$
  downstream-node $= (1+1) \times 5 + 6 = 16$

We should also note at this point that this allocation method places some constraints upon the simulation. Clearly, there are situations where either r_h or r_l can be zero. This is an indication that there are more processors than are necessary for the given number of space nodes (i.e., some PEs would be idle). This situation is flagged as an error and the program terminated. This allocation mechanism is used for all algorithms that follow.

III. The One- and Two-Step Algorithms

In this section, we first explain the existing algorithm for the lax-momentum computation. In the *one-step* algorithm, one time step is executed before an interprocessor communication (IPC) is required. In the *two-step* algorithm, two time steps are executed before an IPC is required.

The core of the lax-momentum computations, for a given time step and space node, is quite simple. The general form of the computations is as follows (expressed in a C-like pseudocode). In this notation, the *odd* and *even* references correspond to successive time steps. The *evenk*, *evenq*, and *evenu* variables are the previous time steps values for density, flow, and speed, respectively, and the *oddk*, *oddq*, and *oddu* variables are the current time step density, flow, and speed for which a new solution is being sought:

```
for each segment Seg_i on processor P_i do
  for each space node j within Seg_i do
    oddk[j] = k[j] + (evenk[j + 1] +
      evenk[j − 1] − C*(evenq[j + 1] −
      evenq[j − 1]))/2                    (5)
    oddq[j] = q[j]+ (evenq[j + 1] +
      evenq[j − 1])/2 −
      D*(evenu[j + 1]*evenu[j + 1]*
      evenk[j + 1] +
      V*evenk[j + 1] −
      evenu[j − 1]*evenu[j − 1]*evenk[j − 1] −
      V*evenk[j − 1]) + E*((evenk[j + 1]*F −
      evenq[j + 1])/T[j + 1] + (evenk[j − 1]*F −
      evenq[j − 1])/T[j − 1])             (6)
    oddu[j] = oddq[j]/oddk[j]             (7)
  end for
end for
```

Here C, D, E, F, and V are constants across all processors, and $oddk[j]$, $oddq[j]$ and $oddu[j]$ are the $\boldsymbol{k_j^i}$, $\boldsymbol{q_j^i}$ and $\boldsymbol{u_j^i}$ described in Section I. In a straightforward implementation, once the *odd* values are computed and distributed to the neighboring segments (on neighboring PEs), the *even* variables are loaded with the odd values, and the computation resumes for the next time step.

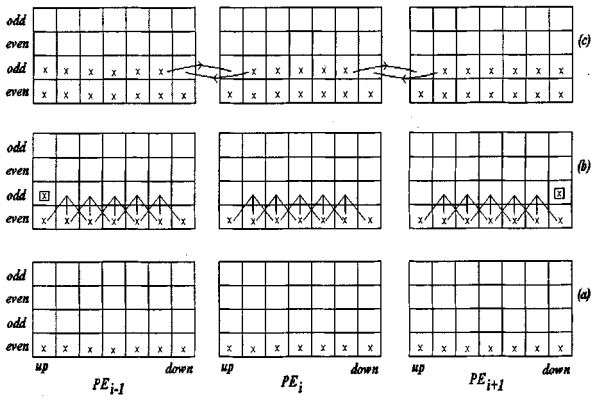

Fig. 2. The one-step algorithm.

The major steps of the one-step algorithm are demonstrated in Fig. 2, which depicts a hypothetical situation with three PEs and 15 space nodes. Each segment can be viewed as a boundary value problem whose upstream and downstream nodes contain new boundary values for each time step. Thus, after partitioning the space nodes into segments of size $\boldsymbol{k}$ *nodes* each (five in our example), each processor will allocate space for $\boldsymbol{k+2}$ nodes to hold the boundary value as well.

The flow of the algorithm (time) begins at the bottom of the figure and proceeds upward [Fig. 2(a)–(c)]. We begin the algorithm at some arbitrary time t_n with all even variables set to some initial value [step (a)]. From (5) and (6) in the algorithm, we note that the new (odd) values for node j are computed from previous (even) values from nodes $j-1$, j, and $j+1$ [step (b)]. After new values are computed, each PE exchanges values with its neighbors to update their boundaries [step (c)]. Note that the upstream boundary on the farthest upstream segment, and the downstream boundary on the farthest downstream segment, will be determined by other means [marked by a box in step (b)]. These data are either prerecorded roadway data (our case) or could be obtained, in real time, from sensors strategically placed upon an actual roadway. Once the new (odd) values have been moved into the old (even) variables, the algorithm is ready to proceed with step t_{n+1}.

As mentioned previously, in any system with high IPC latency, the designer must structure the algorithm so that large amounts of computation are performed between communication steps. The only possible way to reduce the number of IPCs between processing elements here is to see if more than one computation can be done before an IPC is necessary. Whether this can be done or not depends upon the functional form of the computation. If we rewrite (5) and (6) and reorder the terms, we can see more clearly the data interdependencies between the current node and its neighbors

$$
\begin{aligned}
\text{oddk}[j] &= \text{evenk}[j-1]/2 - C/2^{*}\text{evenq}[j-1] \\
&\quad + k[j] \\
&\quad + C/2^{*}\text{evenq}[j+1] + \text{evenk}[j+1]/2 \\
\text{oddq}[j] &= E^{*}\text{evenk}[j-1]^{*}F/T[j-1] \\
&\quad + E^{*}\text{evenq}[j-1]/T[j-1] + \text{evenq}[j-1]/2 \\
&\quad + D^{*}\text{evenu}[j-1]^{*}\text{evenu}[j-1]^{*}\text{evenk}[j-1] \\
&\quad + D^{*}V^{*}\text{evenk}[j-1] \\
&\quad + q[j] \\
&\quad + \text{evenq}[j+1]/2 + E^{*}\text{evenk}[j+1]^{*}F/T[j+1] \\
&\quad + \text{evenq}[j+1]/T[j+1] \\
&\quad + D^{*}\text{evenu}[j+1]^{*}\text{evenu}[j+1]^{*}\text{evenk}[j+1] \\
&\quad + D^{*}V^{*}\text{evenk}[j+1]
\end{aligned}
$$

We ignore *oddu* because it is simply a function of *oddk* and *oddq* and is easily obtained once they are known. We can see that each new computation has inputs from only *adjacent* (both upstream and downstream) and *current* nodes from the previous (even) time step. In a sense, each new (odd) computation is independent for each node, given that the neighboring nodes' data are known. Therefore, it should be possible to do a *second computation* on at least *part* of the nodes within a segment before incurring the cost of an IPC. The challenging question now is what to do with the upstream and downstream boundaries and their neighbors. During each IPC, we will send both the boundary values from the first complete time step and the inputs necessary for the neighbor to *compute* the second time steps' boundary values. When these computations are complete, we will be ready to begin the next two-step iteration. This is the two-step algorithm. It incurs a very small overhead in central processing unit (CPU) time, but the IPC time is cut almost in half. The assumption is that the extra computations in completing the second time steps boundary are more than made up for by the saved IPC. We will quantify these savings in Section VI. Finally, the two-step algorithm places one additional constraint on our PE allocation technique: both r_h and r_l must be three or more in order for there to be enough nodes for the partial second step to be performed.

Returning once again to our example and Fig. 3, we can see the major steps of this algorithm.

Steps (a) and (b) are exactly as they were in the one-step algorithm. In step (c), the "inner" space nodes are computed for the second time step. Step (d) is the new IPC. Note that not only the boundary value from the first time step [step (d), items 2 and 3] but also the additional data necessary for the neighbor to compute the "missing" information are sent so it can complete the second time step [step (d), items 1 and 4]. Several computations are performed in step (e). Once the first time steps' boundary has been loaded [step (d), item 2], the PE can then complete the second time step for its own upstream node [step (e), item 5]. With the partial information delivered in step (d), item 1, it can complete the upstream boundary for the second step [step (e), item 6]. Similar processing is applied to the data from step (d), items 3 and 4, to compute a new downstream node [step (f), item 7)] and boundary value [step (g), item 8 and step (h), item 9]. The algorithm is now ready to proceed with step t_{n+2}.

IV. Implementation on the nCUBE2

The nCUBE2 is a multiple-instruction multiple-data (MIMD) hypercube parallel processing computer. Each PE contains a proprietary processor with a 20-MHz clock and 4 MB of local memory. Peak theoretical performance is 4.1 millions of floating point operations per second (MFLOPS) per PE. The hypercube architecture is a distributed-memory message passing architecture. In a hypercube of dimension d, there are $P = 2^d$ processors, labeled $0, 1, \ldots, P-1$. Two processors

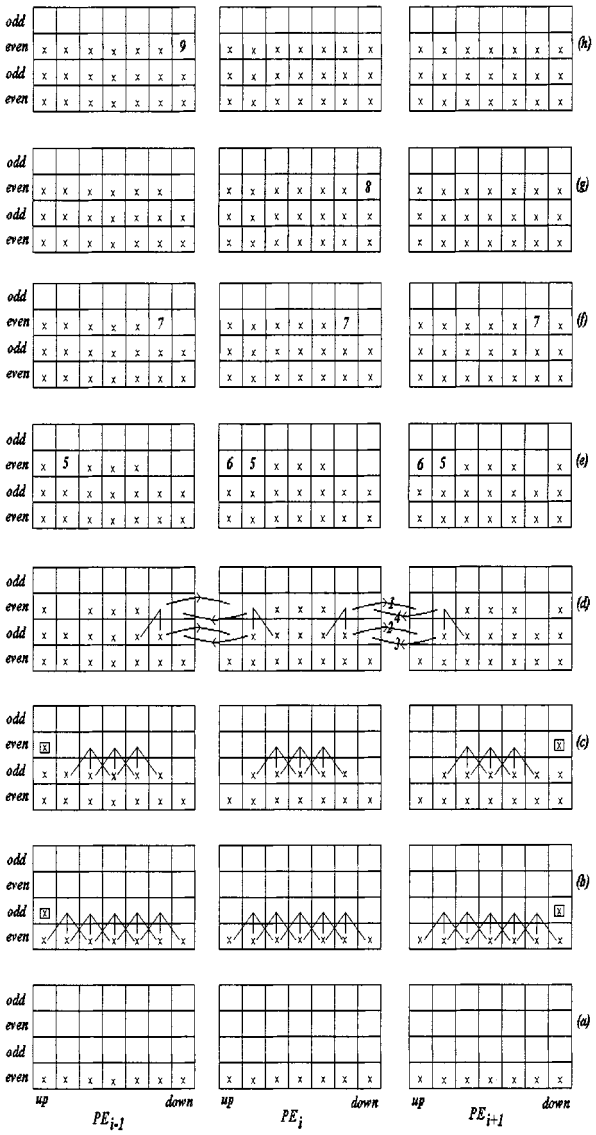

Fig. 3. The two-step algorithm.

P_j and P_k are directly connected (neighbors) iff the binary representations of j and k differ in exactly one bit. Each edge of a hypercube graph represents a direct connection between two processors. Thus, any two processors in a hypercube are separated by at most d other processors. Fig. 4 illustrates a hypercube graph of dimension $d = 4$. The number of processors to be allocated to a job is chosen by the user, but must be a power of two.

Table II summarizes interprocessor communication times for neighbor processors and basic floating-point operation times for the nCUBE2 [13]. We see that communication even between neighboring processors is many times slower than floating-point operations.

In an architecture with high communication latencies (such as nCUBE2), the algorithm designer must structure the algorithm so that as much computation as possible is done between communication steps.

Two important factors that influence the delivered performance on this machine are load balancing and reduction of communication overhead.

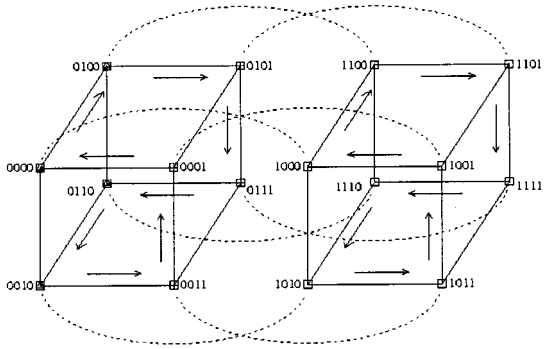

Fig. 4. Hypercube of dimension four with gray code mapping of linear arrays in its subcubes.

TABLE II
COMPUTATION AND COMMUNICATION TIMES (nCUBE)

Operation	Time(μSEC)	Comm/Comp
8-byte transfer	111	-
8-byte add	1.23	90
8-byte multiply	1.28	86

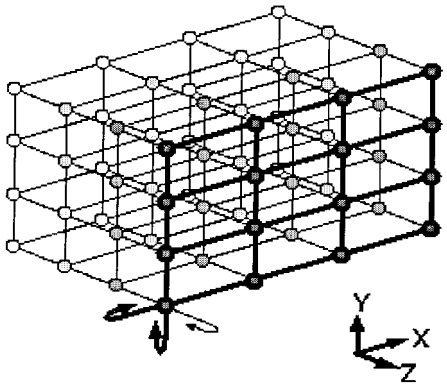

Fig. 5. The Cray T3E toroidal mesh.

V. IMPLEMENTATION ON THE CRAY T3E

The Cray T3E (model 900) is a distributed shared-memory MIMD architecture with a three-dimensional (3-D) torus topology and bidirectional channels. Each PE consists of a DEC Alpha 21 164 processor, a system control chip, local memory, and a network router. The custom-made control chip implements the distributed shared memory, which consists of all the local memories in the PEs. Each processor is connected to six other processors in a 3-D toroidal mesh, as seen in Fig. 5. All PEs in opposite "faces" of the mesh are connected to each other. The T3E supports low-latency, high-bandwidth communications via this mesh and is capable of delivering a 64-bit word every system clock in all six directions, for a raw

TABLE III
COMPILER OPTIMIZATION RESULTS

Option	T_1 (sec)	T_1/B
None	B = 5.652	1.00
-h pipeline3	5.178	0.931
-h unroll	5.838	1.033
-h split	7.004	1.239
-h inline3	5.808	1.028
-h vector3	5.765	1.020
-h scalar3	5.647	0.999
-O3	5.714	1.011
-O2	5.721	1.012
-O1	6.083	1.076

bandwidth of 600 MB/s, with data bandwidths of 100–480 MB/s after protocol overheads.

The DEC Alpha 21 164 processor is capable of issuing four instructions per clock period, giving it a theoretical peak rate of 900 MFLOPS. Each PE supports 128 MB of local memory.

Optimization on the T3E is not a straightforward process. The user is given fine-grained control over what aspects of optimization they wish to control. There are no less than 26 selectable options, most having more than one level of control. Without in-depth knowledge of the exact relationships among the compiler, the source code, and the objectives of the application, it is nearly impossible to know which combination of optimizations would be optimal. Most were selected based on their descriptions, together with some basic benchmarking of those whose effects could not be predicted beforehand. Some combinations yielded better performance, others worse. Table III shows the options as benchmarked (these benchmarks were made using the complete simulation code, but on a smaller data set size for expediency).

All pairs of options were tried, the most obvious being *pipeline3* with *scalar3*, since they both resulted in improvements. This combination resulted in a run time of 5.211 s (a ratio of 0.923). But a better combination was discovered: *pipeline3* with *unroll*, resulting in a run time of 5.170 s (a ratio of 0.915), and the best run time of all the options/combinations tested. Note that no three-option combinations were tested due to time constraints. The *unroll* directive instructs the compiler to unroll all loops generated by the compiler. The *pipeline* directive instructs the compiler to aggressively pipeline the software to the CPU, including speculative loads and operations. Both of these compiler directives will result in longer compilation times but faster execution times.

Because the T3E maps a linear array of PEs into a mesh, the mapping of highway segments to processors is a straightforward linear mapping. For performance reasons, it is important that neighboring segments are mapped to neighboring processors. The most problematic implementation issue is the paradigm with which distributed memory is implemented. The T3E has several different IPC mechanisms to choose from. At the highest level are standard message passing interfaces (like the standard *PVM* application programming interface) to the lower level (and faster) interfaces built around shared-memory operations. It is at this level that an IEEE POSIX-like shared memory interface is defined which is much more efficient than PVM. Because of this, the Cray *shmem_get*() and *shmem_put*() shared-memory routines were selected.

VI. SIMULATION TESTING

The simulation was implemented on the 1024-node nCUBE2 computer located at the Massively Parallel Computer Research Laboratory, Sandia National Laboratories, Albuquerque, NM. The simulation was also run on the Cray T3E at the San Diego NPACI, San Diego, CA, and Pittsburgh PSC Supercomputing Center, Pittsburgh, PA.

As a test site, a multiple entry/exit section of the I-494 highway was chosen in Minneapolis, MN. This section of eastbound I-494 extends from I-394 in the west to Nicollet Avenue in the east. It is 15.5 mi long, with 17 exit and 19 entry ramps. Data for the simulation were recorded on April 9, 1997, and span a 24-h period beginning at midnight of that day. The Appendix shows a sample of the data. To test the simulation, the time and space mesh sizes were chosen as $\Delta t = 0.5$ s and $\Delta x = 100$ ft. The discrete model contained 814 space nodes. The tests were analyzed in two ways: comparison with real data and computational performance.

Traffic data are collected at the *upstream/downstream boundaries* of the freeway section and at *check-station sites* (check-nodes) inside the freeway section. Figs.12–15 show plots drawn after the collected data. Let N be the number of discrete time points at which real traffic-flow data are collected. We compare the simulated traffic flow volume and speed data with those from the check-stations. There were a total of 23 check-stations. The following error moduli are used to measure the effectiveness of the simulation in comparison with the actual data:

$$\text{Max Absolute Error} = \text{MAX}_{1\le j\le N}|\text{Observed}_j - \text{Simulated}_j|$$

$$\text{Max Relative Absolute Error} = \text{MAX}_{1\le j\le N}\frac{|\text{Observed}_j - \text{Simulated}_j|}{\text{Observed}_j}$$

$$\text{Mean Absolute Error} = \frac{1}{N}\sum_{j=1}^{N}|\text{Observed}_j - \text{Simulated}_j|$$

$$\text{Mean Relative Error} = \frac{1}{N}\sum_{j=1}^{N}\frac{|\text{Observed}_j - \text{Simulated}_j|}{\text{Observed}_j}$$

$$\text{Relative Error with 2-Norm} = \sqrt{\frac{\sum_{j=1}^{N}(\text{Observed}_j - \text{Simulated}_j)^2}{\sum_{j=1}^{N}\text{Observed}_j^2}}$$

$$\text{Standard Deviation} = \sqrt{\frac{1}{N-1}\sum_{j=1}^{N}(\text{Observed}_j - \text{Simulated}_j)^2}.$$

The error statistics are sampled in Tables IV and V. The relative errors (*Rel. 2-Norm*) are at a level of about 10% for the volume but lower for the speed measurements. These error levels are consistent with past simulations carried out by simulation systems based on a single-processor computer (see [1] and [11]).

TABLE IV
ERROR STATISTICS FOR TRAFFIC-FLOW VOLUME

Volume Error (vehicles/5-min)						
Site	*Max. Abs.*	*Max. Rel.*	*Mean Abs.*	*Mea Rel.*	*Rel. 2-Norm*	*Std. Dev.*
1	97.7	0.55	20.2	14.7	0.14	27.2
2	84.5	1.30	18.3	14.7	0.13	25.3
3	64.1	1.22	11.4	11.8	0.09	16.0
4	77.2	0.68	17.3	13.7	0.12	23.5
5	72.6	4.97	17.6	21.9	0.15	24.0
6	63.4	1.69	12.1	13.2	0.10	17.2
7	72.5	3.48	12.0	14.1	0.10	17.4

TABLE V
ERROR STATISTICS FOR TRAFFIC-FLOW SPEED

Speed Error (mph)						
Site	*Max. Abs.*	*Max Rel.*	*Mean Abs.*	*Mea Rel.*	*Rel.-2Nrm*	*Std. Dev.*
1	7.4	0.19	1.3	2.6	0.03	1.7
2	6.4	0.16	1.3	2.5	0.03	1.7
3	5.9	0.10	1.4	2.6	0.03	1.8
4	6.5	0.15	1.7	3.2	0.04	2.1
5	7.4	0.14	1.7	3.0	0.04	2.1
6	8.1	0.14	1.6	3.0	0.04	2.1
7	9.9	0.17	2.0	3.6	0.05	2.5

VII. PERFORMANCE STUDY

In general, the serial (or single PE) computational performance of a given algorithm implemented on a given computer architecture is expressed in terms of MFLOPS. In order to derive this measure, an estimate of the number of floating-point operations (FLOPS) is needed for the algorithm in question. Upon examining (5)–(7) in the algorithm, we see there are some 32 floating-point operations contained within the main simulation loop. There are, in actuality, 34 such operations (this pseudocode was somewhat simplified for purposes of clarity). In general, each space node computation requires 34 FLOP. If we rewrite the one-step algorithm pseudocode in terms of the number of FLOPS performed, we arrive at the following:

```
for ns 5-minute time steps do
  for 300 seconds do
     for each space node on this PE do
       ⟨34 FLOP⟩
     end for
     IPC
     advance time Δt seconds
  end for
end for
```

TABLE VI
MFLOPS RESULTS (nCUBE)

Algorithm	T_1 (sec)	MFLOPS
1-step	6729.9	0.71
2-step	5739.9	0.83

TABLE VII
ONE-STEP PERFORMANCE (nCUBE)

#Pes	T_P (sec)	S_P	E_P	IPC Time (sec)	IPC % of T_P
1	6729.9	N/A	N/A	N/A	N/A
2	3427.2	1.96	0.98	77.26	2.25
4	1757.4	3.83	0.96	78.20	4.45
8	936.1	7.19	0.90	97.62	10.43
16	520.0	12.94	0.81	100.95	19.41
32	304.8	22.08	0.69	100.76	33.06
64	196.3	34.28	0.54	97.49	49.66
128	146.3	46.01	0.36	88.36	59.59
256	120.53	55.84	0.22	96.52	80.08

In this testing, recall that $\Delta t = 0.5$ s. If the number of space nodes operated on by this PE is N, then the one-step single PE total number of operations is

$$\text{ns} \cdot 600 \cdot N \cdot 34 = 20\,400 \cdot \text{ns} \cdot N.$$

The two-step algorithm is somewhat more complicated. Its pseudocode looks like the following:

```
for ns 5-minute time steps do
  for 300 seconds do
    for each space node in 1st step on this PE
    do
      ⟨34 FLOP⟩
    end for
    advance time Δt seconds
    for each space node in 2nd step on this
    PE do
      ⟨34 FLOP⟩
    end for
    IPC
    ⟨34 FLOP⟩◇
    ⟨34 FLOP⟩◇
    ⟨34 FLOP⟩ ◇ ◇
    ⟨34 FLOP⟩ ◇ ◇
    advance time Δt seconds
  end for
end for
```

TABLE VIII
TWO-STEP PERFORMANCE (nCUBE)

#PEs	T_P (sec)	S_P	E_P	IPC Time (sec)	IPC % of T_P
1	5739.9	N/A	N/A	N/A	N/A
2	2911.2	1.97	0.99	51.26	1.76
4	1481.4	3.88	0.97	40.91	2.76
8	778.2	7.38	0.92	57.81	7.43
16	425.2	13.50	0.84	59.82	14.07
32	242.7	23.65	0.74	57.61	23.73
64	153.6	37.37	0.58	57.86	37.67
128	109.4	52.47	0.41	55.21	50.46
256	88.5	64.85	0.25	55.09	62.24

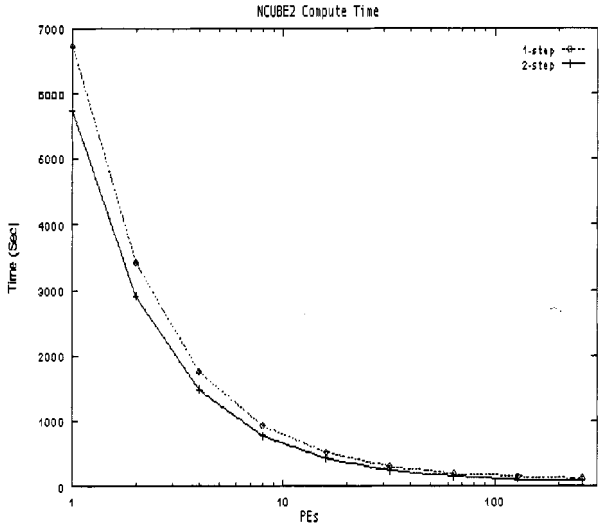

Fig. 6. Compute time of one- and two-step algorithms.

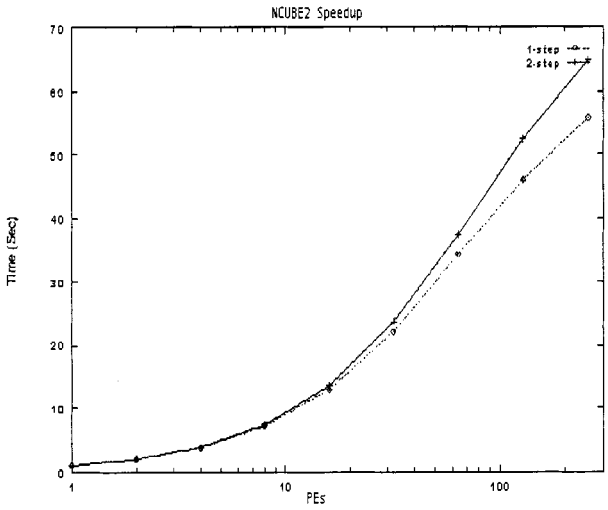

Fig. 7. Speedup of one- and two-step algorithms.

Some explanation is in order. The space node loops certainly make sense in terms of the number of nodes that are being solved for in both the first and second steps of the algorithm. Plus it

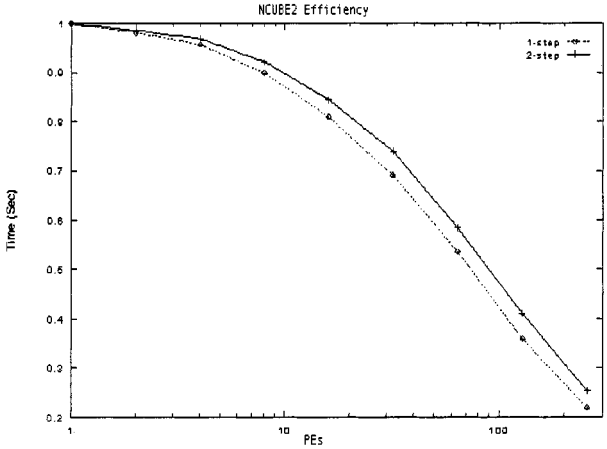

Fig. 8. Efficiency of one- and two-step algorithms.

TABLE IX
TWO-STEP TO ONE-STEP COMPARISON (nCUBE)

# PEs	T_P Gain $\left(\frac{1\text{-step}}{2\text{-step}}\right)$	IPC Time Gain $\left(\frac{1\text{-step}}{2\text{-step}}\right)$
1	1.18	N/A
2	1.18	1.52
4	1.19	1.92
8	1.20	1.69
16	1.22	1.69
32	1.25	1.75
64	1.28	1.69
128	1.33	1.61
256	1.37	1.75

TABLE X
MFLOPS RESULTS (Cray T3E)

Algorithm	T_1 (sec)	MFLOPS
1-step	73.676	64.91
2-step	67.040	71.42

makes sense that there would be two additional nodes solved for (marked $\diamond$) since the second step does not operate on all the space nodes that the first step does (two less). In actuality, the two-step algorithm requires slightly more computation than the one-step. In the one-step case, the segment end-nodes are exchanged between neighboring PEs during the IPC to be used as segment boundary values for the next compute cycle. This cannot happen in the two-step case, since the segment end-nodes have yet to be calculated for the second step. This forces neighboring PEs to both compute the boundary values, but for different purposes: one as the segment end-node and the other as the boundary value for the next compute cycle. Thus, we must perform two additional node computations (marked $\diamond\diamond$) for a total number of operations of

$$\text{ns} \cdot 300(34N + 34(N-2) + 34 \cdot 4) = 20\,400 \cdot \text{ns}(N+1).$$

TABLE XI
ONE-STEP PERFORMANCE (Cray T3E)

# PEs	T_P (sec)	S_P	E_P	IPC Time (sec)	IPC % of T_P
1	73.68	N/A	N/A	N/A	N/A
2	37.42	1.97	0.98	0.74	1.93
4	19.63	3.75	0.94	1.17	5.94
8	10.98	6.71	0.84	1.13	10.29
16	6.61	11.15	0.70	1.12	16.98
32	4.29	17.16	0.54	1.18	27.43
64	3.42	21.55	0.34	1.12	32.85
128	2.82	26.13	0.20	1.12	39.61
256	2.36	31.18	0.12	1.14	48.16

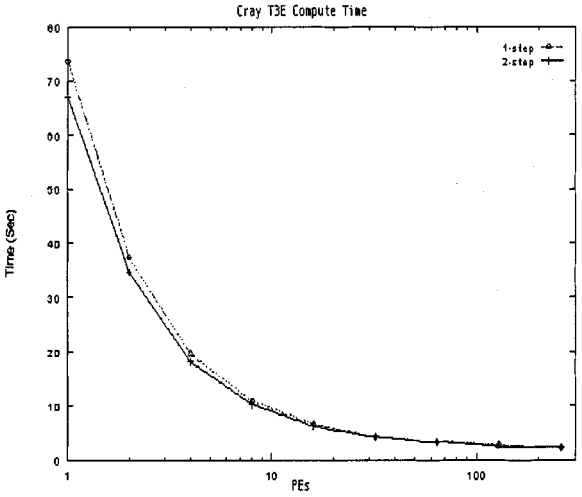

Fig. 9. Compute Time of one- and two-step algorithms.

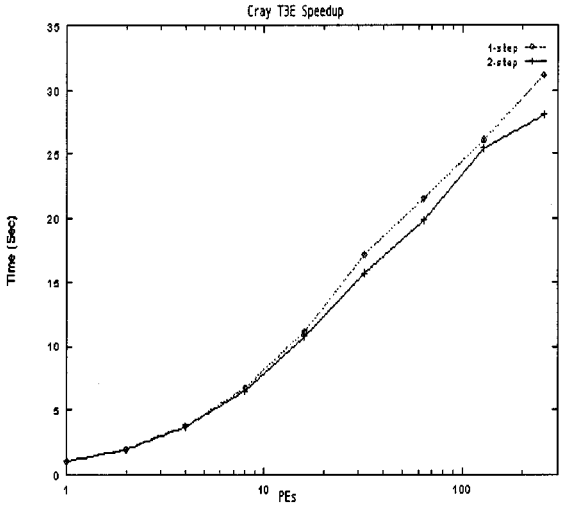

Fig. 10. Speedup of one- and two-step algorithms.

To derive the desired MFLOPS value, we need only divide the total number of operations by both the single PE execution time and 10^6. For this simulation, ns $= 288$ and $N = 814$.

VIII. nCUBE2

Table VI summarizes the MFLOPS results for nCUBE2.

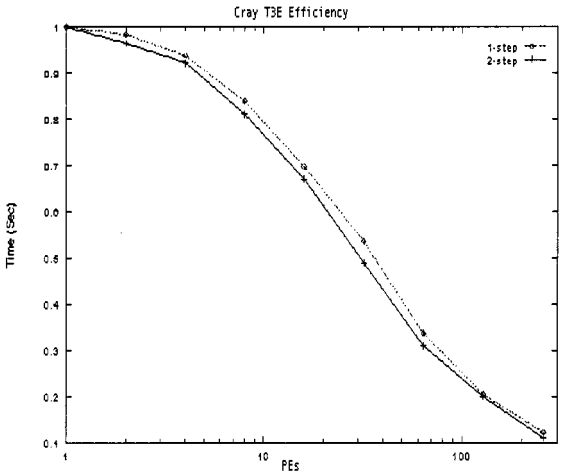

Fig. 11. Efficiency of one- and two-step algorithms.

TABLE XII
TWO-STEP PERFORMANCE (Cray T3E)

# PEs	T_P (sec)	S_P	E_P	IPC Time (sec)	IPC % of T_P
1	67.04	N/A	N/A	N/A	N/A
2	34.75	1.93	0.97	0.55	1.58
4	18.15	3.69	0.92	0.57	3.14
8	10.34	6.48	0.81	1.10	10.64
16	6.26	10.72	0.67	1.10	17.55
32	4.27	15.69	0.49	1.12	26.09
64	3.38	19.84	0.31	1.10	32.49
128	2.63	25.45	0.20	1.12	42.41
256	2.39	28.05	0.11	1.12	46.97

TABLE XIII
TWO-STEP TO ONE-STEP COMPARISON (Cray T3E)

# PEs	T_P Gain $\left(\frac{\text{1-step}}{\text{2-step}}\right)$	IPC Time Gain $\left(\frac{\text{1-step}}{\text{2-step}}\right)$
1	1.10	N/A
2	1.08	1.35
4	1.09	2.04
8	1.06	1.03
16	1.05	1.02
32	1.00	1.05
64	1.01	1.02
128	1.08	1.00
256	0.99	1.01

Here, we see the two-step algorithm outperforming the one-step algorithm when the software is run on a single PE, as mentioned at the end of Section VII.

For the parallel performance analysis (for both the nCUBE2 and T2E), we evaluate the following measures: the serial execution time (T_1), the parallel execution time (T_P), the parallel speedup (S_P), and the parallel efficiency (E_P). Additionally, T_P can be broken down further to component measures of *input*, *computation*, and *output*. The computation time is simply the

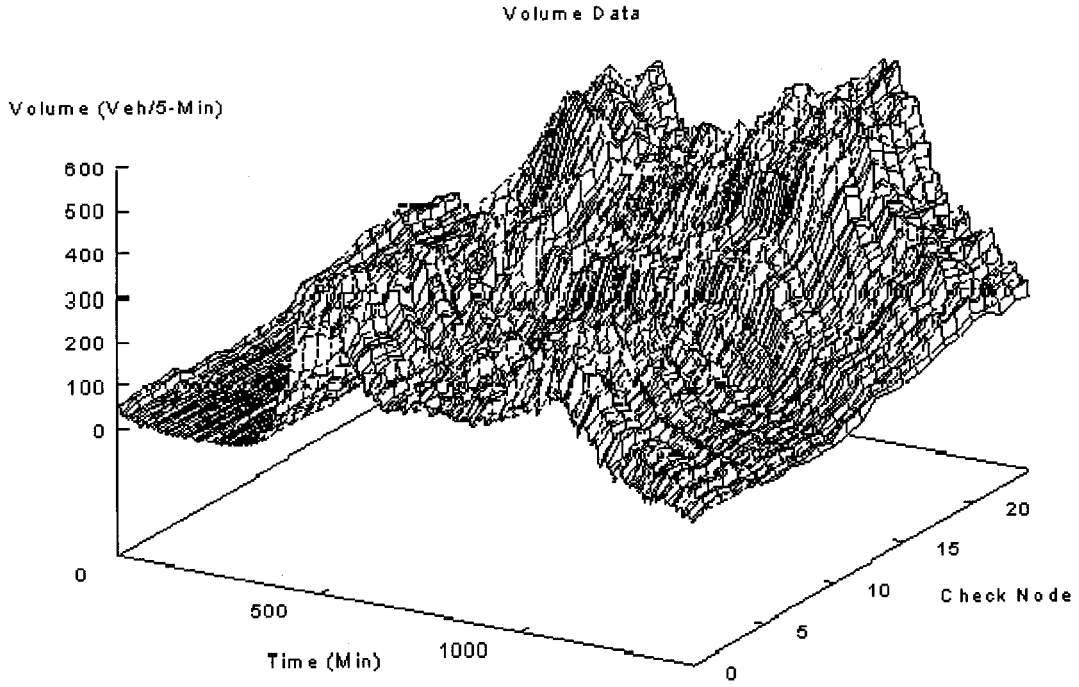

Fig. 12. Occupancy data.

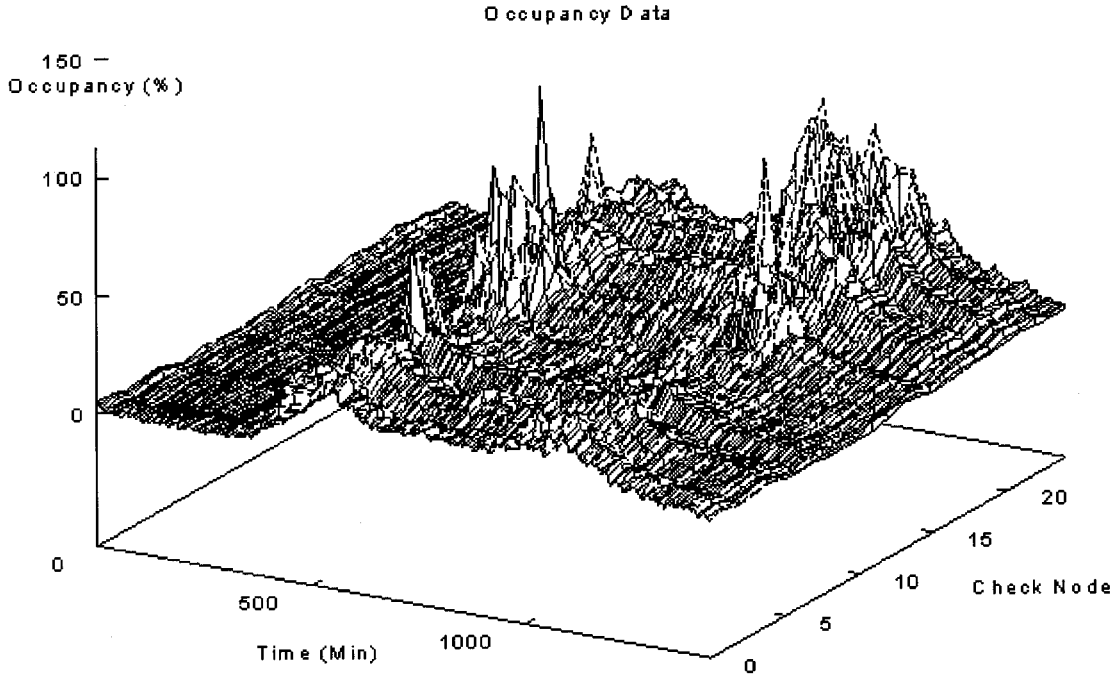

Fig. 13. Volume data for I-494.

time for the discrete model computations. This time can be further decomposed into *calculation time* and *IPC time*.

NCUBE2 performance data are presented first as Tables VII and VIII, then as Figs. 6–8. Table IX contains the two-step over the one-step gains.

Based on the single PE timings, the restructuring of the code, which eliminates the odd and even swapping, saves a total of 15%. Even assuming that this fraction remains constant across PE sizes, the two-step algorithm, by way of halving the number of IPCs, still saves an additional 12% at the 256 PE size, where IPC times are the highest (as a fraction of total compute time). On average, the IPC time is reduced by 42% by the two-step algorithm. A theoretical expected peak value would be 50%, but in practice cannot be obtained. The data content of the IPC for the two-step algorithm is more than twice that of the one-step algorithm, which will result in slightly longer IPC times. Overall,

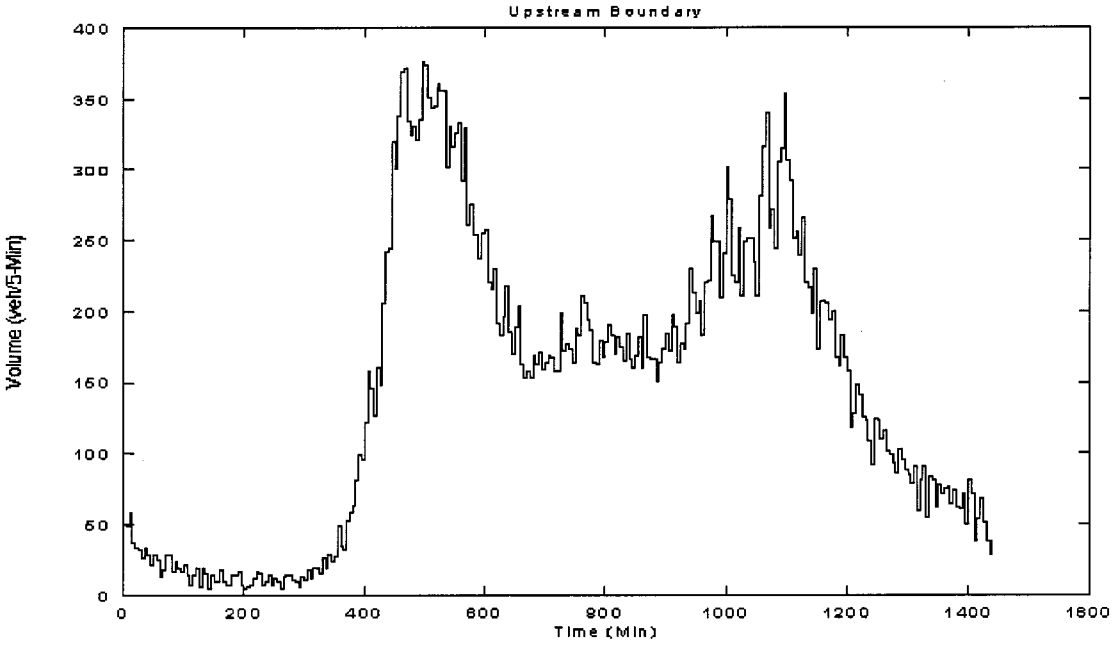

Fig. 14. I-494 upstream boundary data.

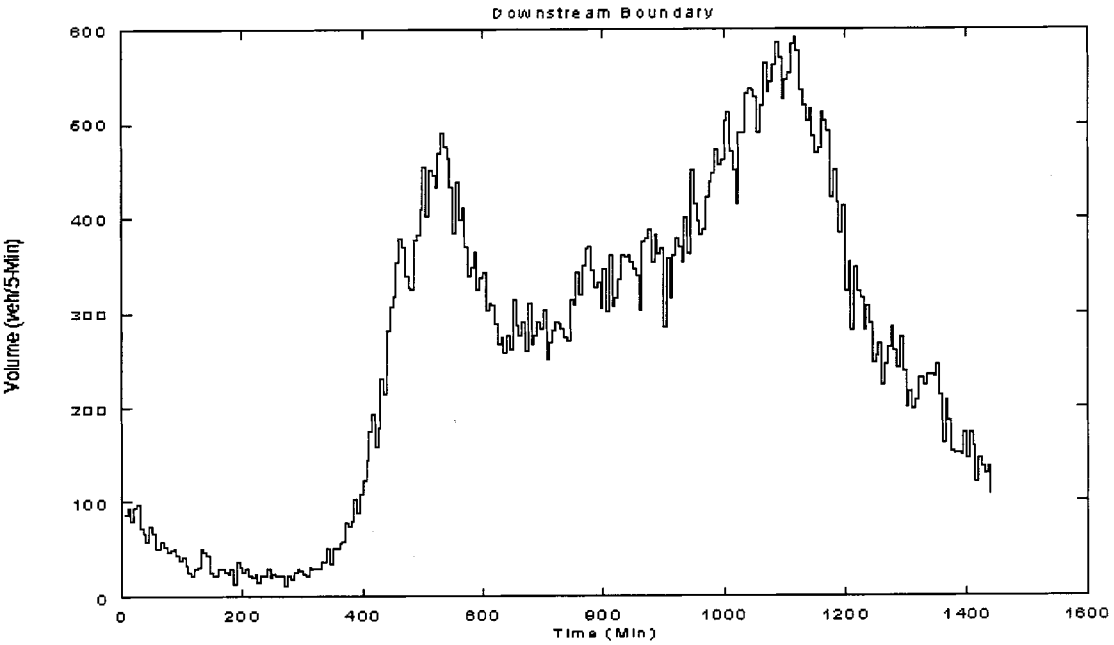

Fig. 15. I-494 downstream boundary data.

these data show that the two-step algorithm is ideally suited to the nCUBE2 architecture, where IPCs are quite costly compared to computation.

IX. Cray T3E

Table X summarized the MFLOPS results for T3E.

One may wonder why the two-step algorithm outperforms the one-step algorithm when the software is run on a single PE. This is a side effect of the implementation of the two-step algorithm. Let us recall from (5)–(7) in the algorithm that the space nodes currently being solved for have their data stored into locations prefixed with *odd*, while the data for the same space node, but for the previous time step, are prefixed with *even*. In the one-step algorithm, after the *odd* data are computed, the data are simply copied into the *even* variables for use in the next time step. However, in the two-step algorithm, the computations are done "in place," as it were, so that the first step is stored into the *odd* variables and the second time step is stored into the *even* variables, thus avoiding the overhead of the copy operation. This has the benefit of lower computation times but the disadvantage of approximately doubling the size of the core computational section of the code.

Cray T3E performance data are presented first as Tables XI and XII, then as Figs. 9–11. Table XIII contains the two-step over one-step gains.

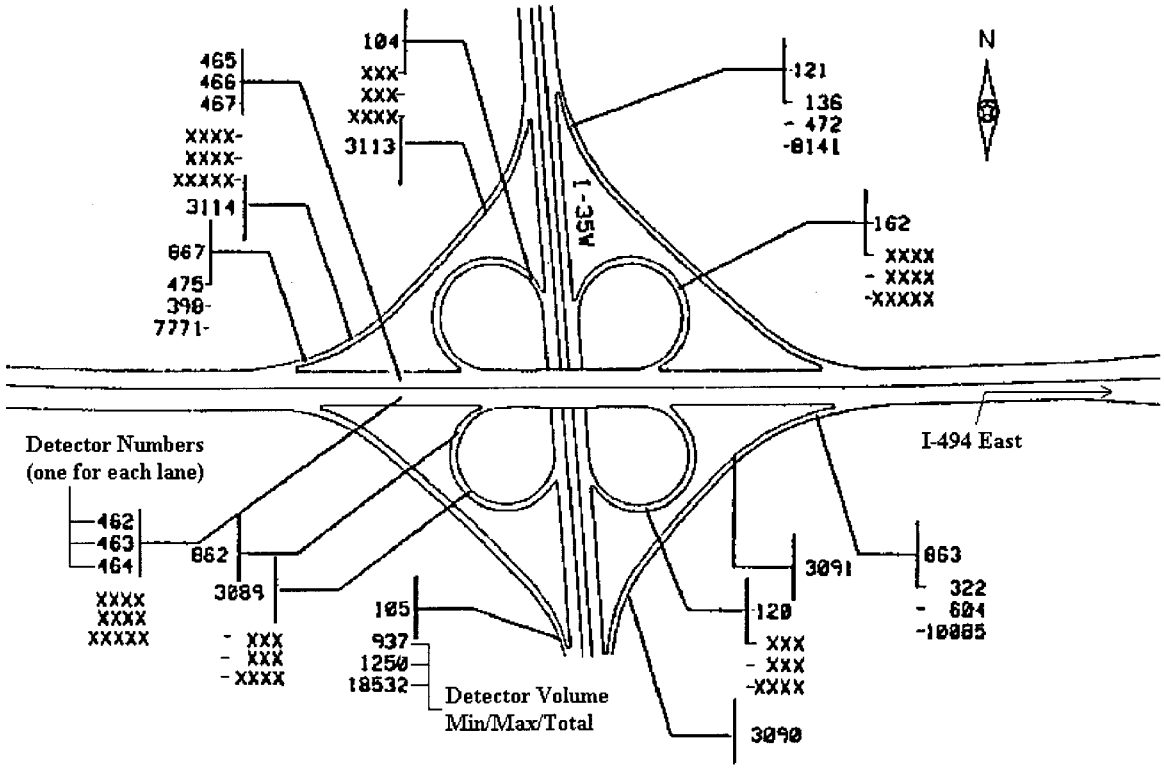

Fig. 16. I-494 and I-35 intersection near Minneapolis.

We see that the two-step algorithm is faster than the one-step algorithm for $P \leq 32$. For $P \geq 64$, the one-step is faster because a) the IPC time is very fast on the T3E, and even halving it does not lead to great savings overall and b) there are slightly more computations in the two-step algorithm than the one-step. Thus, there is a breakpoint (in P) for the gain of the two-step, and it occurs at $P = 32$. This breakpoint is so small here because the number (N) of road space nodes assigned to each PE in our simulation is very small (e.g., 12 or 13 nodes per PE for $P = 64$). In realistic simulations, we expect this N to be much larger and, thus, the two-step algorithm will be even faster.

X. Conclusion

A very important component of an intelligent highway management system is a traffic simulation system. The design of a real-time traffic simulation system is a challenging problem. In this paper, the design of a parallel (macroscopic) traffic simulation system is demonstrated. This system could be used as a component of a real-time simulation system. This parallel system was implemented on the NCUBE2 and on the Cray T3E parallel computer. Tests were run with real traffic data to validate the accuracy and computational rate of the system. A 24-h 15.5-mi simulation, with real traffic data, took 2.39 s on the Cray T3E and 88.51 s on the NCUBE2 versus 65.65 min on a typical single processor system (a 133-MHz Pentium). Two algorithms were implemented, offering tradeoffs in execution time, IPC time, and memory size. The two-step algorithm, when compared to the one-step, reduced significantly the communication time on both parallel computers and would be of considerable savings when larger highway segments (e.g., entire beltway systems) are modeled. With increase in problem size, even higher efficiencies are to be expected.

Appendix
Sample Highway Schematic and Data File

Fig. 16 is a small section of a schematic from I-494 near Minneapolis (see Sections V and VII). Specifically, this is the interchange at I-494 and I-35W. It should be treated as example data only. The detector locations correspond to simulation site locations, e.g., detector 120 corresponds to exit ramp 14, located at space node 382 in this study. The values that show XXX or XXXX indicate a detector failure. Some ramps contain multiple detector locations. For example, detectors 3090, 3091, and 863 all point to the ramp from northbound I-35W to eastbound I-494. But only detector 863 data were contained in the data files (please see the equation at the top of the next page). This is a fragment from an original data file, exactly as delivered by MNDOT. This fragment is one of nine total files that covered the 24-h period beginning midnight April 9 through midnight April 10 and covering the 15.5-mi section of highway simulated. A three-digit line number has been added to the beginning of each line for discussion purposes. The "..." at the end of most lines indicate that additional columns have been removed for brevity. The column containing the detector number contains the volume data for that detector.

Acknowledgment

The authors would like to thank D. Berg, a traffic engineer from the Minnesota Department of Transportation, for pro-

001 VOLUME \ OCCUPANCY
002
003 494ER2
004 Wednesday, April9, 1997
005 Printed : 11/13/97
006

007		842	840	841	853	852	105	···						
008		494/FRANEX		494/NFRAEM		494/PENEM		494	···					
009			494/SFRAEM		494/PENEX		35W/E494SM		···					
010	———	···												
011	00 : 05	9	0	19	2	16	1	4*	0	15	1	19	2	···
012	00 : 10	9	0	12	1	14	1	7*	0	16	1	37	4	···
013	00 : 15	14	1	17	2	13	1	8	0	13	1	33	3	···
014	00 : 20	3	0	11	1	13	1	6	0	11	1	32	3	···
015	00 : 25	10*	0	6	0	11	0	5	0	14	1	32	5	···
016	00 : 30	6	0	8	1	12	1	3*	0	6	0	19	2	···
017	———	···												
018	00 : 35	4	0	9	1	8	0	3*	0	4	0	26	3	···
019	00 : 40	7*	0	11	1	6	0	2*	0	8*	0	19	2	···
020	00 : 45	5	0	14	2	6	0	4*	0	11	1	23	3	···
021	00 : 50	9*	0	5	0	18*	1	4*	0	15	1	18	1	···
022	00 : 55	2	0	9	1	9	0	3*	0	5	0	24	3	···
023	01 : 00	3*	0	17	2	10	0	3*	0	5	0	12	1	···
024	———	···												
025	01 : 05	4*	0	9	1	3	0	3*	0	10	1	16	2	···

viding them with the real traffic data, the Massively Parallel Computing Research Laboratory, Sandia National Laboratories, Albuquerque, NM, for providing access to the nCUBE2, and the Pittsburgh Supercomputing Center, Pittsburgh, PA (1996–1997), and the NPACI San Diego Supercomputing Center, San Diego, CA (1998–2001), for access to the Cray T3E. Finally, the authors thank the reviewers of this document, whose comments greatly enhanced the quality of its presentation.

REFERENCES

[1] A. T. Chronopoulos *et al.*, "Traffic flow simulation through high order traffic modeling," *Math. Comput. Modeling*, vol. 17, no. 8, pp. 11–22, 1993.
[2] A. T. Chronopoulos *et al.*, "Efficient traffic flow simulation computations," *Math. Comput. Modeling*, vol. 16, no. 5, pp. 107–120, 1992.
[3] A. Chronopoulos and G. Wang, "Traffic flow simulation through parallel processing," *Parallel Comput.*, vol. 22, pp. 1965–1983, 1997.
[4] C. Hirsch, *Numerical Computation of Internal and External Flows*. New York: Wiley, 1988, vol. 2.
[5] A. S. Lyrintzis *et al.*, "Continuum modeling of traffic dynamics," in *Proc. 2nd Int. Conf. Applied of Advanced Techniques in Transportation Engineering*, Minneapolis, MN, Aug. 18–21, 1991, pp. 36–40.
[6] P. Yi *et al.*, "Development of an improved high order continuum traffic flow model," *Trans. Res. Rec.*, vol. 1365, pp. 125–132, 1993.
[7] T. Junchaya and G. Chang, "Exploring real-time traffic simulation with massively parallel computing architecture," *Trans. Res. C*, vol. 1, no. 1, pp. 57–76, 1993.
[8] V. Kumar *et al.*, *Introduction to Parallel Computing Design and Analysis of Algorithms*. Redwood City, CA: Benjamin/Cummings, 1994.
[9] G. Cameron and G. Duncan, "PARAMICS—Parallel microscopic simulation of road traffic," *J. Supercomput.*, vol. 10, pp. 25–53, 1996.
[10] I. Angus *et al.*, *Solving Problems on Concurrent Processors*. Englewood Cliffs, NJ: Prentice-Hall, vol. II, pp. 104–113.
[11] A. Chronopoulos and C. Johnston, "A real-time traffic simulation system," *IEEE Trans. Veh. Technol.*, vol. 47, pp. 321–331, Feb., 1998.
[12] E. Anderson *et al.*, "The benchmarker's guide to single-processor optimization for CRAY T3E systems," *Cray Res.*, 1997.
[13] S. K. Kim and A. T. Chronopoulos, "A class of lanczos-like algorithms implemented on parallel computers," *Parallel Comput.*, vol. 17, pp. 763–778, 1991.
[14] L. Mikhailov and R. Hanus, "Hierarchical control of congested urban traffic—Mathematical modeling and simulation," *Math. Comput. Sim.*, vol. 37, pp. 183–188, 1994.
[15] A. Bachem *et al.*, "Microscopic traffic simulations of road networks using high-performance computers," *HPCN Eur.*, pp. 306–311, 1996.
[16] T. Ishikawa, "Development of a road traffic simulator," *IEEE Trans. Veh. Technol.*, vol. 47, pp. 1066–1071, Aug., 1998.

Anthony Theodore Chronopoulos (M'87–SM'97) received the Ph.D. degree from the University of Illinois at Urbana-Champaign in 1987.

He has published more than 30 journal and more than 30 refereed conference proceedings publications in the areas of scientific computation, parallel and distributed computing, and simulations. His work is cited in more than 130 noncoauthors' research articles.

Dr. Chronopoulos has received 11 federal/state government research grants.

Charles Michael Johnston received the B.Sci. degree in mathematics from the University of Michigan, Ann Arbor, in 1980 and the M.Sci. degree in computer science from Wayne State University, Detroit, MI, in 1998.

He has worked on numerous real-time simulation systems for NASA and General Motors. He is currently a Senior Analyst with Concurrent Computer Corporation, Southfield, MI.