# The Parallelization of a Highway Traffic Flow Simulation

Charles Michael Johnston — Concurrent Computer Corporation
Anthony Theodore Chronopoulos — University of Texas San Antonio

## Abstract

*This work implements and analyses 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 1-step and 2-step algorithms) to solve the simulation equations were developed and implemented. They were then tested on a Cray T3E, a 3-D torroidal mesh with very fast inter-processor communication (IPC) times. 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. The execution time for a 24-hour traffic-flow simulation over a 15.5-mile freeway, which takes 65.7 minutes on a typical single processor computer, took only 2.39 seconds on the Cray T3E. The 2-step algorithm, whose goal is to trade off extra computation for fewer IPC's, was shown to save more than 5% on computation time. This parallel implementation has proven potential for real-time traffic engineering applications.*

## 1: Introduction

A very important component of an Intelligent Highways' management System is a traffic simulation system. Such a system would consist of a traffic flow simulation that is able to simulate traffic on freeways and arterial networks on a computer system. This computer system consists of hardware and software. Input/output devices provide real-time traffic data measurements from a network of traffic detectors (loops or cameras) and data on the road geometry or other traffic characteristics. 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 drivers' guidance.

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, 2, 3, 4, 5, 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, 5, 6] and implementing it on the Cray T3E parallel computer.

Tests with real data from the I-494 freeway in Minneapolis were conducted. Each processing element (PE) of the Cray T3E is a 450MHz DEC Alpha 21164. Tests were run on the Cray T3E at the Pittsburgh Supercomputing Center. The execution time for a 24-hour traffic flow simulation of a 15.5-mile freeway, which takes 65.65 minutes of computer time (on a 133MHz single-processor Pentium computer), took only 2.39 seconds on the parallel traffic simulation system implemented on the Cray T3E.

We adopted the Lax-Momentum traffic model [11]. Let $\Delta t$ and $\Delta x$ 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 $\Delta t$, 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 $\frac{\Delta x}{\Delta t} > u_f$, where $u_f$ is the free flow speed (see [4]). Typical choices for the space and time meshes are $\Delta x = 100$ *feet* and $\Delta t = 0.5$ *sec* 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 $Seg_0, \ldots, Seg_{P-1}$ and assigning each segment to the processors (PE) $P_{j_0}, \ldots, P_{j_{P-1}}$. The choice of indices $j_0, \ldots, j_{P-1}$ 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. 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.

## 2: 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 $r = \frac{n}{P}$ nodes per PE. Obviously, the closest we can come is an allocation of $r_h = \lceil \frac{n}{P} \rceil$ and/or $r_l = \lfloor \frac{n}{P} \rfloor$.

Our first approach to this problem we call **Method I**: allocate $r_h$ 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 1, where *rem* is the number of space nodes that the last PE will need to process (i.e. the number remaining after the initial $r_h$ distribution):

| Total PEs (*P*) | Used PEs | Idle PEs | $R_h$ | *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 |

**Table 1. PE Allocation Using Method I**

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 PEs are idle – an unreasonably large number. In order 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:

$$r_h = \left\lceil \frac{n}{P} \right\rceil \tag{1}$$

$$r_l = \left\lfloor \frac{n}{P} \right\rfloor \tag{2}$$

$$h + l = P \tag{3}$$

$$r_h \times h + r_l \times l = n \tag{4}$$

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 allocates $r_l$ space nodes, and *h* PEs each allocates $r_h$ space nodes.

In addition to knowing how many space nodes to allocate to each PE, we must also 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...P-1*, then the mapping is defined via the pseudo-code:

```
if $P_i < h$ then
    upstream-node = $P_i \times r_h + 1$
    downstream-node = $(P_i + 1) \times r_h$
else
    temp = $r_h \times h$
    offset = $P_i - h$
    upstream-node = offset $\times r_l + 1 +$ temp
    downstream-node = (offset + 1) $\times r_l +$ 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

$r_l = \left\lfloor \frac{16}{3} \right\rfloor = 5 \qquad r_h = \left\lceil \frac{16}{3} \right\rceil = 6$

$l = 6 \times 3 - 16 = 2 \qquad h = 3 - 2 = 1$

$\therefore$ two PEs 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 × 1 = 6
      upstream-node = 0 × 2 + 1 + 6 = 7
      downstream-node = (0 + 1) × 5 + 6 = 11
P_2:  offset = 2 - 1 = 1
      temp = 6 × 1 = 6
      upstream-node = 1 × 5 + 1 + 6 = 12
      downstream-node = (1 + 1) × 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 both the **1-step** and **2-step** algorithms. The **2-step** algorithm places one additional constraint on this 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 (this will become obvious when the algorithm is more fully explained).

## 3: The 1- and 2-Step Algorithms

The core of the Lax-Momentum computations, for a given time step and space node, are quite simple. The general form of the computations is as follows (expressed in a C-like pseudo-code). In this notation, the *odd* and *even* references correspond to time. 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 $\boldsymbol{k}^i_j$, $\boldsymbol{q}^i_j$ and $\boldsymbol{u}^i_j$ described in Section 1. 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. Since one time step is computed for each interprocessor communication (IPC), we call this the **1-Step** algorithm.

The major steps of the algorithm are demonstrated in Figure 1, 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 ***k* nodes** each (five in our example), each processor will allocate space for ***k+2* nodes** to hold the boundary value as well.

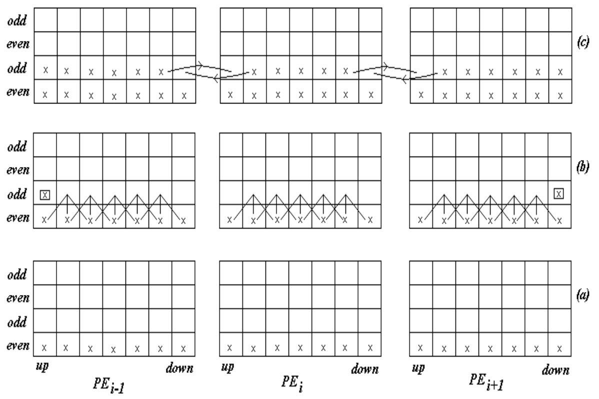

**Figure 1. The 1-Step Algorithm**

The flow of the algorithm (time) begins at the bottom of the figure and proceeds upward ((*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) 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 algorithm 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:

```
oddk[j] = evenk[j-1]/2 - C/2*evenq[j-1]
```

```
             + k[j]
             + C/2*evenq[j+1] + evenk[j+1]/2
    oddq[j] = E*evenk[j-1]*F/T[j-1]
             + E*evenq[j-1]/T[j-1] + evenq[j-1]/2
             + D*evenu[j-1]*evenu[j-1]*evenk[j-1]
             + D*V*evenk[j-1]
             + q[j]
             + evenq[j+1]/2 + E*evenk[j+1]*F/T[j+1]
             + evenq[j+1]/T[j+1]
             + D*evenu[j+1]*evenu[j+1]*evenk[j+1]
             + D*V*evenk[j+1]
```

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 a 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, plus 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 2-step iteration. This is the **2-Step** algorithm. It incurs a very small overhead in 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 6.

Returning once again to our example, and Figure 2, we can see the major steps of this algorithm.

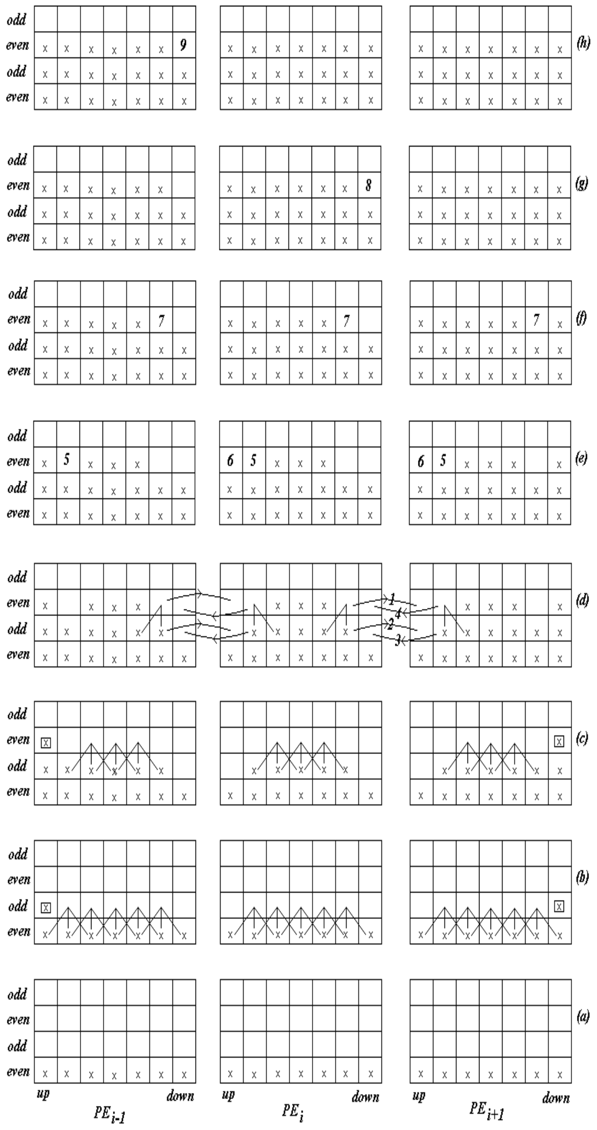

**Figure 2. The 2-Step Algorithm**

Steps (*a*) and (*b*) are exactly as they were in the 1-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 is the boundary value from the first time step sent (step (*d*), items *2* and *3*), but also the additional data necessary for the neighbor to compute the "missing" information 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 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 $t_{n+2}$.

## 4: Implementation on the Cray T3E

The Cray T3E (model 900) is a distributed shared-memory MIMD architecture with a 3-D torus topology and bi-directional channels. Each PE consists of a DEC Alpha 21164 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 3D toroidal mesh, as seen in Figure 3. 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 bandwidth of 600 MB/s, with data bandwidths of 100 to 480 MB/s after protocol overheads.

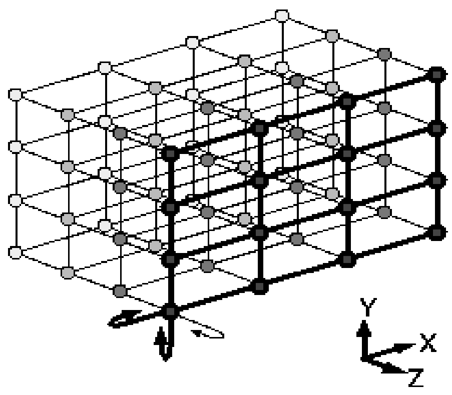

**Figure 3. The Cray T3E Toroidal Mesh**

The DEC Alpha 21164 processor is capable of issuing four instructions per clock period, giving it a theoretical peak rate of 900 MFLOPS. Each PE supports 128MB 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 between 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. Here are the options as benchmarked (these benchmarks were made using the complete simulation code, but on a smaller data set size for expediency):

| 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 |

**Table 2. Compiler Optimization Results**

All pairs of options were also tried. The most obvious being *pipeline3* with *scalar3*, since they both resulted in improvements. This combination resulted in a run time of 5.211 seconds (a ratio of 0.923). But a better combination was discovered: *pipeline3* with *unroll*, resulting in a run time of 5.170 seconds (a ratio of 0.915), and the best runtime of all the options/combinations tested. Note that no three-option combinations were tested due to time constraints. 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.

## 5: Simulation Testing

The final simulation was run on the Cray T3E at the Pittsburgh Supercomputing Center in Pittsburgh, Pennsylvania. This Cray T3E is composed of 256 300MHz-clock PEs and 256 450MHz-clock PEs. We only used the fastest 256 PEs for these tests to insure that there was not any variation in computation time between PEs in a given run due to PE clocking.

As a test site, a multiple entry/exit section of the I-494 highway was chosen in the metropolitan Minneapolis, Minnesota, area. This section of Eastbound I-494 extends from I-394 in the West to Nicollet Avenue in the East. It is 15.5 miles long, with 17 exit and 19 entry ramps. Data for the simulation were recorded on April 9, 1997, and spans a 24-hour period beginning at midnight of that day. To test the simulation, the time and space mesh sizes were $\Delta t$ = 0.5 second and $\Delta x$ = 100 feet. The discrete model contained 814 space nodes. We used the same error metrics as in [1, 11]. The deviation of our computer model values for volume and speed were at most 10 percent from the real traffic measurement values, and agreed with the error range in the 1-PE and [1, 11] runs.

## 6: 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 millions of floating point operations per second (MFLOPS). In order to derive this measure, an estimate of the number of floating point operations (FLOP) is needed for the algorithm in question. Upon examining equations (5) - (7), we see there are some 32 floating point operations contained within the main simulation loop. There are, in actuality, 34 such operations (this pseudo-code was somewhat simplified for purposes of clarity). In general, each space node computation requires 34 FLOP. If we rewrite the 1-step algorithm pseudo-code 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 $\Delta$t seconds
    end for
end for
```

In this testing, recall that $\Delta t$ = 0.5 second. If the number of space nodes operated on by this PE is $N$, then the 1-step single PE total number of operations is:

$$ns \cdot 600 \cdot N \cdot 34 = 20400 \cdot ns \cdot N$$

The **2-step** algorithm is somewhat more complicated. It's pseudo-code looks like:

```
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 $\Delta$t 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 $\Delta$t seconds
    end for
end for
```

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 1st and 2nd steps of the algorithm. Plus it makes sense that there would be two additional nodes solved for (marked ♦) since the 2nd step does not operate on all the space nodes that the 1st step does (two less). In actuality, the **2-step algorithm requires slightly more computation than the 1-step**. In the 1-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 can not happen in the 2-step case, since the segment end-nodes have yet to be calculated for the 2nd step. This forces neighboring PEs to **both** compute the boundary values, but for different purposes: one as the segment end-node, the other as the boundary value for the next compute cycle. Thus we must perform two additional node computations (marked ♦♦) for a total number of operations of:

$$ns \cdot 300\left(34N + 34(N-2) + 34 \cdot 4\right) = 20400 \cdot ns(N+1)$$

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. Table 5 summarizes the results.

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

**Table 5. MFLOPS Results**

One may wonder why the 2-step algorithm outperforms the 1-step algorithm when the software is run on a single PE. This is a side effect of the implementation of the 2-step algorithm. Let us recall from (5) - (7) 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, is prefixed with *even*. In the 1-step algorithm, after the *odd* data are computed, the data is simply copied into the *even* variables for use in the next time step. However, in the 2-step algorithm the computations are done "in place," as it were, so that the 1st step is stored into the *odd* variables, and the 2nd 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

For the parallel performance analysis, 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 time for the discrete model computations. This time can be further decomposed into **calculation time** and **IPC time**.

Cray T3E performance data are presented first as Tables 6 and 7, and then as Figures 4 through 6.

| # 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 |

**Table 6. 1-Step Performance**

| # 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 7. 2-Step Performance**

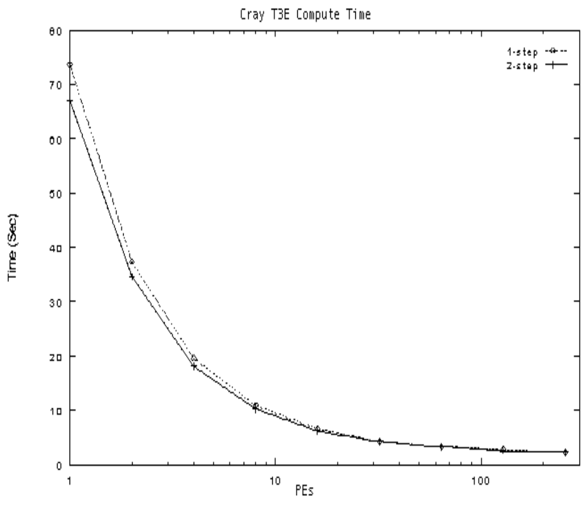

**Figure 4. Compute Time 1- & 2-Step Algorithms**

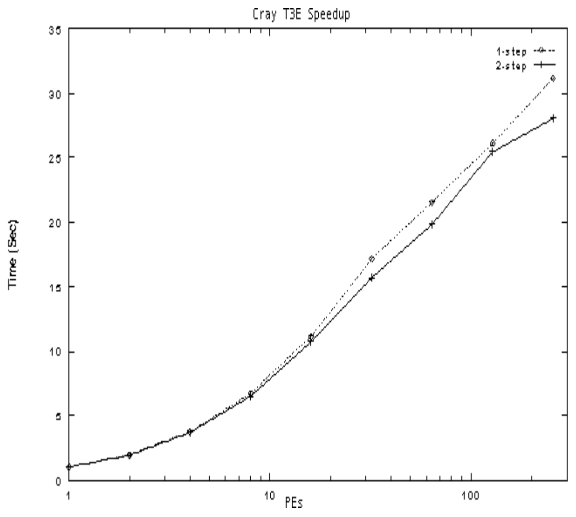

**Figure 5. Speedup 1- & 2-Step Algorithms**

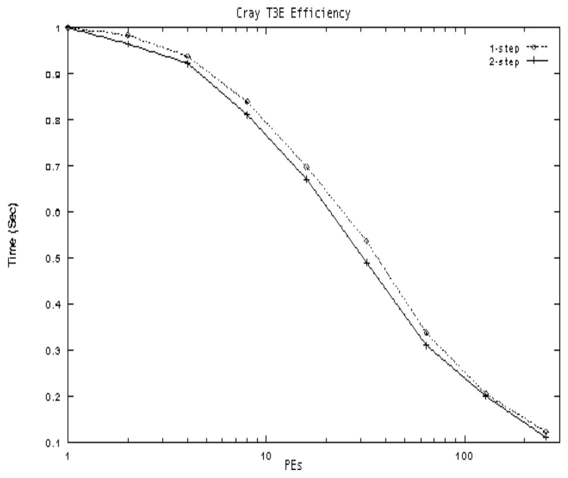

**Figure 6. Efficiency 1- & 2-Step Algorithms**

| # 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 | 1.10 | 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 |

**Table 8. 2-Step to 1-Step Comparison**

We see that the 2-step algorithm is faster than the 1-step algorithm for $P \leq 32$. For $P \geq 64$ the 1-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 is slightly more computations in the 2-step algorithm than the 1-step. Thus there is a breakpoint (in $P$) for the gain of the 2-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 2-step algorithm will be even faster.

## 7: Conclusions

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. 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 Cray T3E parallel computer. Tests were run with real traffic data to validate the accuracy and computational rate of the system. A 24-hour, 15.5-mile simulation, with real traffic data, took 2.39 seconds on the Cray T3E versus 65.65 minutes on a typical single processor system (a 133MHz Pentium). Two algorithms were implemented offering tradeoffs in execution time, IPC time and memory size. The 2-step algorithm, when compared to the 1-step, reduced computation time an average of 5.4% on the Cray T3E.

## Acknowledgments

We would like to acknowledge partial funding by a Cray/SGI grant, by NSF ASC-9634775, and access to the Cray T3E provided by the Pittsburgh Supercomputing Center, Pittsburgh, PA (1996-1997) and the San Diego Supercomputing Center, San Diego, CA (1998).

## References

[1] A.T. Chronopoulos et. al., “Traffic Flow Simulation Through High Order Traffic modeling”, *Mathematical Computing Modeling*, Vol. 17, No. 8, pp. 11-22, 1993.

[2] A.T. Chronopoulos et. al., “Efficient Traffic Flow Simulation Computations”, *Mathematical and Computer 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”, *Vol.2, John Wiley and Sons*, 1988.

[5] A.S. Lyrintzis et al., “Continuum Modeling of Traffic Dynamics”, *Proc. of the 2nd Int. Conf. on Appl. of Advanced Tech. in Transportation Eng., Aug. 18-21, Minneapolis, Minnesota*, pp. 36-40, 1991.

[6] P. Yi et al., “Development of an Improved High Order Continuum Traffic Flow Model”, *Transp. Res. Rec.*, 1365, pp. 125-132, 1993.

[7] B. Dixon and J Sallow, “High-Performance Sorting Algorithms for the Cray T3D Parallel Computer”, *The Journal of Supercomputing*, Vol. 10, pp. 371-396, 1997.

[8] E. Anderson et al., “Performance of the CRAY T3E Multiprocessor”, http://www.cray.com/products/systems/crayt3e/1200/performance.html, 1997.

[9] G. Cameron and G. Duncan, “PARAMICS – Parallel Microscopic Simulation of Road Traffic”, *The Journal of Supercomputing*, Vol. 10, pp. 25-53, 1996.

[10] I. Angus et al., “Solving Problems On Concurrent Processors Volume II”, *Prentice Hall*, pp. 104-113, pp. 126-128.

[11] A. Chronopoulos and C. Johnston, “A Real-Time Traffic Simulation System”, *IEEE Transactions on Vehicular Technology*, Vol. 47, No. 1, 1998.

[12] E. Anderson et al., “The Benchmarker’s Guide to Single-processor Optimization for CRAY T3E Systems”, *Cray Research*, 1997.