# A Communication Latency Hiding Parallelization of a Traffic Flow Simulation

Charles Michael Johnston — Concurrent Computer Corporation
Anthony Theodore Chronopoulos — Division of Computer Science, 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 an nCUBE2, a 1024 node hypercube with very slow inter-processor communication (IPC) times. Tests with real traffic data collected from the freeway network in the metropolitan area of Minneapolis, Minnesota 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 88.51 seconds on the nCUBE2. The 2-step algorithm, whose goal is to trade off extra computation for fewer IPC's, was shown to save more than 19% on total execution time.*

## 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 nCUBE2 parallel computer.

Tests with real data from the I-494 freeway in Minneapolis were conducted. Each processing element (PE) of the nCUBE2 is a proprietary processor running at 20MHz. Tests were run on the nCUBE2 at the Massively Parallel Computing Research Laboratory at the Sandia National Labs. 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 88.51 seconds on the parallel traffic simulation system implemented on the nCUBE2.

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 is chosen according to the Gray code mapping of a linear array onto a hypercube [8].

## 2: 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 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 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.

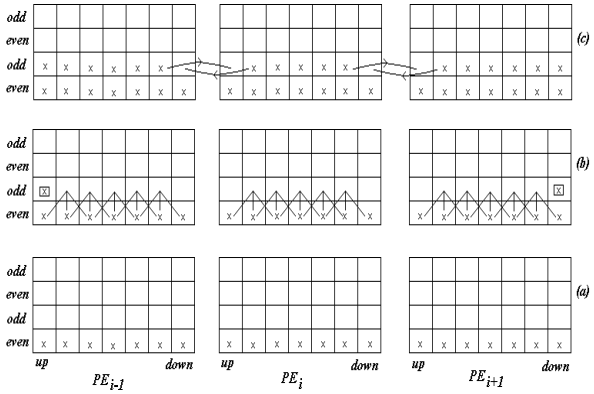


**Figure 1. The 1-Step Algorithm**

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 IPC's 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.

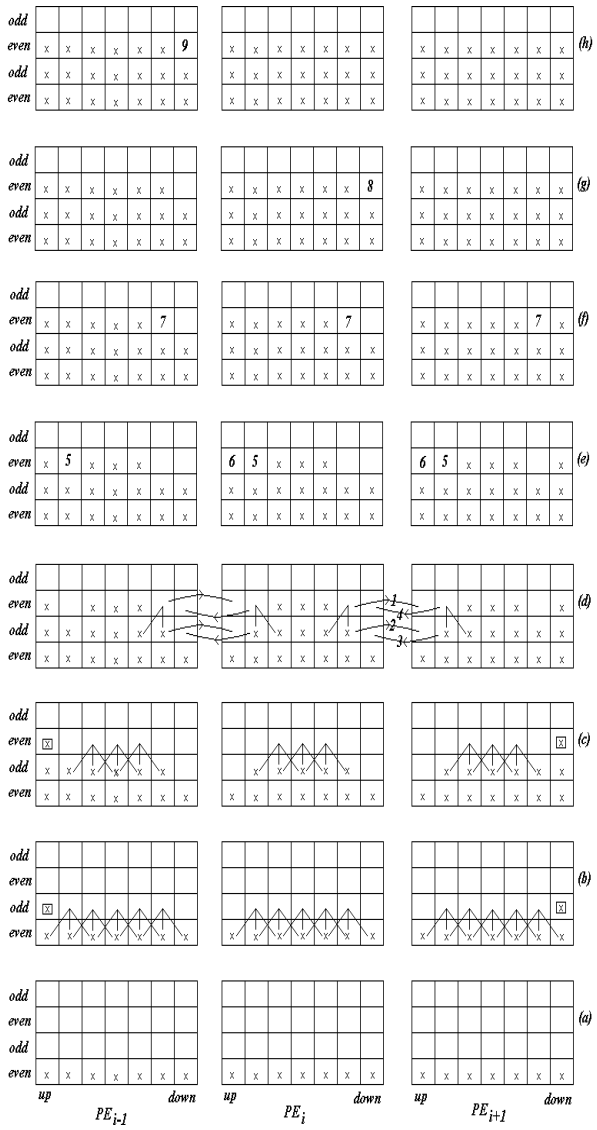


**Figure 2. The 2-Step Algorithm**

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 $t_{n+2}$.

## 3: 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 20MHz clock, and 4MB of local memory. Peak theoretical performance is 4.1 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,…, P-1*. Two processors $P_j$ and $P_k$ are directly connected (neighbors) iff the binary representation 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. Figure 3 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 2.

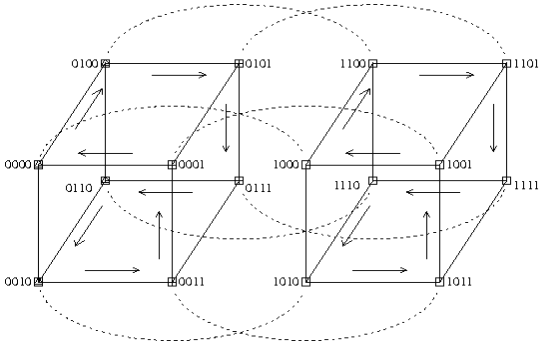


**Figure 3. Hypercube of Dimension 4 with Gray Code Mapping of Linear Arrays in its Subcubes**

Table 1 summarizes inter-processor communication times for neighbor processors and basic floating-point operation times for the nCUBE2 [13]. We see that communications even between neighboring processors is many times slower than floating point operations.

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

**Table 1. Computation & Communication Times**

In an architecture with high communication latencies (such as the 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.

## 4: Simulation Testing

The simulation was implemented on the 1024-node nCUBE2 computer located at the Massively Parallel Computer Research Laboratory at Sandia National Labs in Albuquerque, New Mexico.

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. Test 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. Let *N* be the number of discrete time points at which real traffic flow data are collected. We compare the simulation computed traffic flow volume and speed data with the check-station sites' data. 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:

$$\text{Max Absolute Error} = MAX_{1\le j\le N} \mid Observed_j - Simulated_j \mid$$

$$\text{Max Relative Absolute Error} = MAX_{1\le j\le N} \frac{\mid Observed_j - Simulated_j \mid}{Observed_j}$$

$$\text{Mean Absolute Error} = \frac{1}{N}\sum_{j=1}^{N} \mid Observed_j - Simulated_j \mid$$

$$\text{Mean Relative Error} = \frac{1}{N}\sum_{j=1}^{N} \frac{\mid Observed_j - Simulated_j \mid}{Observed_j}$$

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

$$\text{Standard Deviation} = \sqrt{\frac{1}{N-1}\sum_{j=1}^{N}(Observed_j - Simulated_j)^2}$$

The error statistics are summarized in Tables 2 and 3. The relative errors (*Rel. 2-Norm*) are at a level about 10 percent for the volume but is 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]).

| Volume Error (vehicles/5-min) | | | | | | |
|---|---|---|---|---|---|---|
| *Site* | *Max. Abs.* | *Max. Rel.* | *Mean Abs.* | *Mean 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 2. Error Statistics for Traffic Flow Volume**

| Speed Error (mph) | | | | | | |
|---|---|---|---|---|---|---|
| *Site* | *Max. Abs.* | *Max. Rel.* | *Mean Abs.* | *Mean Rel.* | *Rel. 2-Norm* | *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 |

**Table 3. Error Statistics for Traffic Flow Speed**

## 5: 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 Δt 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 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
```

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:

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

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 4 summarizes the results.

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

**Table 4. 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 computation and communication (IPC) times.

NCUBE2 performance data are presented first as Tables 5 and 6, and then as Figures 4 through 6.

| #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 |

**Table 5. 1-Step Performance**

| #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 |

**Table 6. 2-Step Performance**

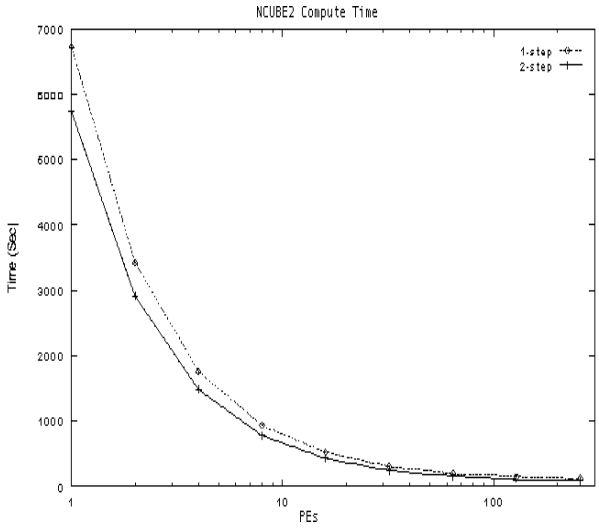


**Figure 4. Compute Time 1- & 2-Step Algorithms**

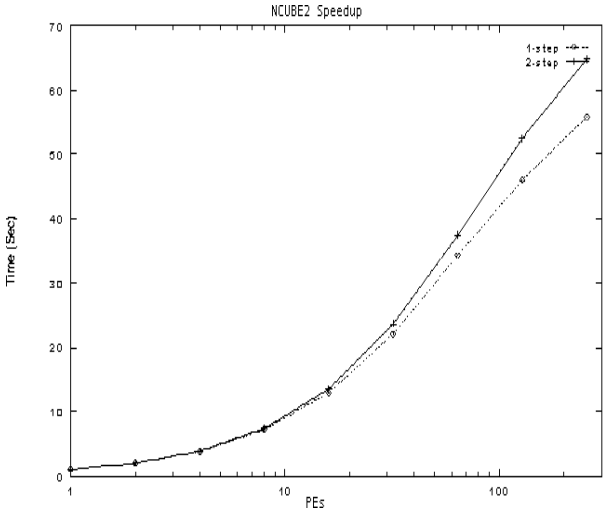


**Figure 5. Speedup 1- & 2-Step Algorithms**

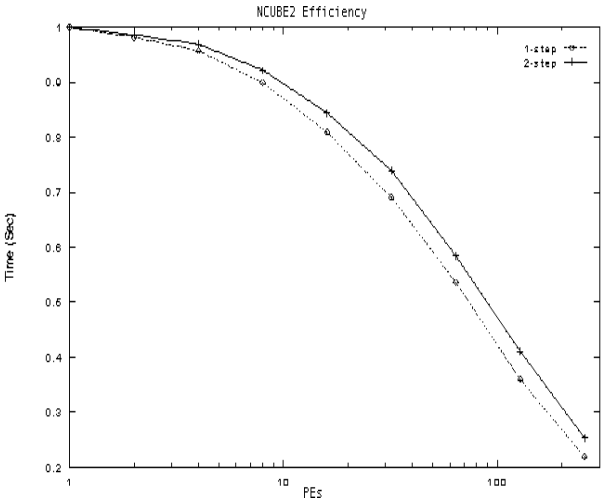


**Figure 6. Efficiency 1- & 2-Step Algorithms**

| # 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 | 1.18 | 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 7. 2-Step to 1-Step Comparison**

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 2-step algorithm, by way of halving the number of IPC's, 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 2-step algorithm. A theoretical expected peak value would be 50%, but in practice, can not be obtained. The data content of the IPC for the 2-step algorithm is more than twice that of the 1-step algorithm, which will result in slightly longer IPC times. Overall, these data show that the 2-step algorithm is ideally suited to the nCUBE2 architecture, where IPC's are quite costly compared to computation.

## 7: Conclusions

The design and implementation of a parallel (macroscopic) traffic simulation system is demonstrated. This parallel system was implemented on the nCUBE2 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 88.51 seconds on the nCUBE2 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 19.4% on the nCUBE2.

## Acknowledgments

We acknowledge Mr. D. Berg, a traffic engineer from the Minnesota Department of Transportation for providing us with the real traffic data and the Massively Parallel Computing Research Laboratory at Sandia National Labs, Albuquerque, New Mexico, for providing access to the nCUBE2.

## Contacting the Authors

The authors may be contacted at the following addresses: Mr. Johnston at chasj@ibm.net, and Dr. Chronopoulos at atc@cs.utsa.edu.

## 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] T. Junchaya and G. Chang, "Exploring real-time traffic simulation with massively parallel computing architecture", *Transpn. Res. C*, Vol. 1, No. 1, pp. 57-76, 1993.

[8] V. Kumar et al., "Introduction to Parallel Computing Design and Analysis of Algorithms", *The Benjamin/Cummings Publishing Company, Inc.*, 1994

[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.

[13] S.K. Kim and A. T. Chronopoulos, "A Class of Lanczos-like Algorithms Implemented on Parallel Computers", *Parallel Computing*, 17, pp. 763-778, 1991.

[14] L. Mikhailov and R. Hanus, "hierarchical control of congested urban traffic – mathematical modeling and simulation", *(IMACS) Mathematics and Computers in Simulation*, 37, pp. 183-188, 1994.

[15] A. Bachem et al., "Microscopic Traffic Simulations of Road Networks using High-Performance Computers", *HPCN Europe*, pp. 306-311, 1996