Abstract

In this paper, we describe implementation of the Parallel Traffic Simulator which applies parallel algorithms and computer graphics in the field of traffic simulation. It simulates traffic behaviour at the individual vehicle level. Parallel environment used consists of a set of workstations connected in a network. Networked workstations make up a loosely coupled parallel computer architecture with distributed memory. PVM network programming environment is used to implement message passing between workstations.

Simulation of traffic is an example of computation with changing load balance. Data parallel approach with initial static data decomposition distributes road network and initial positions of vehicles across participating workstations. However, as the simulation progresses, vehicles may concentrate only on some workstations which leads to system imbalance. Therefore we have developed a dynamic load-balancing algorithm.

Key words: parallel computing, parallel algorithm, dynamic load-balancing, PVM

1. Introduction

Much work has been done in the recent years in parallel computing. Parallel computing has brought substantial benefits to the areas where scientific problems are so complex that solving them requires extraordinary powerful computers (quantum chemistry, astrophysics, meteorology, computational fluid dynamics and turbulence). Of course, another application areas grab results from parallel computing as well. Simulation of traffic at the individual vehicle level is an interesting problem domain where complexity is measured in terms of large data sets that are often too large for a single processor to hold in memory at once. On a single processor it is impossible to control a movement of a very large number of vehicles along roads without making a sacrifice of simulation performance. This implies that by harnessing additional processors in parallel and decomposing the problem domain into sub-domains we can speed-up the simulation. Researchers at University of Edinburgh have proved the benefits of parallel computing. Their traffic simulation can move 200,000 vehicles spread over 7,000 roads at real time rates [1]. Data decomposition is the next important issue when designing parallel algorithms. Using data parallel approach, parallel algorithm initially distributes data across processors and makes them responsible for them. Thus, parallel algorithm for traffic simulation needs to parallelise data such as road network geometry, road network topology and initial positions of vehicles. In [1] each processor is statically assigned a queue, the parallel item of data, which is associated with all the lanes that comprise one direction of the road link. The relation processor-queue remains unchanged during the simulation run. As the number of vehicles on the particular processor vary between simulation frames this can result in situations where some processors have only a few or even none vehicles to control (all vehicles are outside the processor’s region). If such situation occurs load imbalance increases and usage ratio of the processor falls. Dynamic nature of traffic has motivated us to investigate dynamic load-balancing; dynamic redistribution of intersections, roads and vehicles during the simulation.

Our parallel computing environment is a multicomputer; set of UNIX based workstations on a local area network.

The remainder of this paper is structured as follows: We first give an overview of essential requirements in designing traffic simulation. Subsequently, we describe parallel algorithms and compare them with the parallel environment, parallel algorithm and load-balancing used...
in our project. Implementation details of the parallel algorithm and dynamic load-balancing are given in following sections. Finally, we draw some conclusions and give directions for further work.

2. Traffic simulation

Traffic simulation has been in existence for many years. They all had to use simplified models of traffic flow in order to produce results within practical timescales. A typical assumption is to represent traffic flow in a particular road as a single quantity. Such models are generally called macroscopic models, simulation being called macroscopic simulation. Unfortunately, such models do not properly represent real traffic behaviour in congested situations, and do not reproduce the fluctuating nature of real world situations [1]. Models that model traffic at the individual vehicle level being called microscopic models. Microscopic simulation enables more accurate study of congestion formation and dispersion and emphasizes the insight into the nature of road traffic flow. During each time step, the vehicles are moved towards its destinations, and from road to road if necessary, as in real-life.

3. Parallel traffic simulation

The parallel environment used for our project consisted of networked workstations connected by an Ethernet. Our parallel environment can be thought of as a MIMD system. The system consists of a set of computers connected in a network, each executing its own copy of the simulation program and having its own data subset on which to operate. One of the computers, called host, directs the simulation. It dispatches data and control to other workstations which are called nodes. Host process runs on a single workstation from which we can control the simulation. It provides user interface and displays the road network with current positions of vehicles. The application provides a separate load-balancing window. The window shows current distribution of roads on nodes by displaying the parts of the road network in different colors; all roads with the same color correspond to a single node.

3.1. Network data

In order to simulate the movement of vehicles we need road network data. We describe underlying road network with intersections (nodes) and roads (links). In terms of geometry we use two graphical primitives (point and line) to describe the topology of a graph. The simplest intersection has two roads (entry and exit road); two directed links having one node in common. Each intersection-to-intersection connection is either one- or two-way road. Additional data associated with roads (maximum speed allowed, road width, parkdistance, roadlength, etc.) intersections (traffic lights, give way roads, etc.) and polygons (shaded areas on the map which are not important for simulation - buildings, grass, etc.) complete the topology of the road network. Given the described data we can transform them into the parallel form.

3.2 Parallelising the data

Generally, we achieve the concurrency of a computation with:
- **functional parallelism** (pipelining) where a computation is divided into a number of steps that are executed concurrently;
- **data space parallelism** where a computation associated with a program load is initially distributed over the participating processors in either of two ways:
  - image space parallelism (often referred to as image space partitioning or screen subdivision) where areas of pixels are assigned to each processor.
  - object space parallelism (often referred to as object partitioning)

Algorithm for object space parallelism divides the input data, which can be 2D (set of polygons) or 3D (volumes, voxels) objects in some arbitrary manner into data subsets or subvolumes, respectively, and distributes them to multiple processors. Each processor processes these parallel items of data separately and locally, and finally redistributes and composites them into a final image. The total number of data subsets can be expressed...
as $P \times G$, where $P$ is the number of nodes and $G$ is the number of data subsets per node (granularity ratio). $G$ must be chosen properly so as to minimise overhead in data partitioning phase and maximize load balance.

The first task in the design of a parallel algorithm is to build a parallel data framework for simulation process. We have implemented object space partitioning technique where parallel items of data, regions, have to be computed first in order to partition road network geometry into that regions. Given a road network an algorithm divides the simulation area extent (rectangular area of an underlying road network that is used for simulation) into disjoint square regions. The size of these regions is adjusted according to the number of nodes we use (Figure 1). The regions form an equal lattice along each dimension. Each node controls one region. This reduces the amount of computation when partitioning the data but potentially increases the load imbalance. Next task is to classify each road and polygon (from underlying road network) according to the regions they overlap and send them to the nodes assigned to those regions. If road overlaps with two or more regions it will be split into two or more road segments, respectively. Each of these road segments belongs to a separate region and, therefore, is associated with a separate node.

Alternative approaches to partitioning of the data would be:

1. An algorithm with $G>1$. Increasing the number of regions (tasks) per node results in more work involved in partitioning the data and more communication between nodes, but load is better balanced [4].
2. Road network geometry partitioning where an algorithm assigns the first $N$ roads encountered as the first road subset. Using this approach implies each road subset to be a collection of roads which do not necessary lie within square region.

### 3.3 Parallel algorithm

The parallel algorithm for traffic simulation has two parts: host process and node process. Host process starts a simulation by loading road network geometry, data for simulation in microscopic form and parameters for parallel environment. It then initializes the parallel environment and subdivides a simulation area extent into 2D square regions. Regions are then mapped onto nodes: each node is assigned one region together with network data context associated with that region (roads, polygons, intersections, statecycles and initial positions of vehicles). Then all nodes allocate the data, for which they are responsible, in a local memory. When they finish, they notify the host, which sends them a message to start the simulation. During the simulation run host sends messages to nodes at regular intervals in order to provide the movement of vehicles and an update of vehicles positions in simulation window. It also triggers the execution of a load-balancing code in nodes, if necessary, and gathers statistic on current number of vehicles per node.

Node process does the actual simulation work. It acts as a slave to the host process. Node executes a non-blocking loop (asynchronous communication) while waiting for a new request from a sender which can be either host or other node. Request is a message associated with a specific task in node. When request arrives into node’s receive buffer, the node processes it and starts with an execution of the corresponding sequential code. When the requested task is completed, notification is sent back to the sender.

Node’s tasks can be classified, according to the flow of the parallel algorithm, into three groups: data partitioning tasks (initialize node, set node number, set mapview, set window size, set vehicle size, etc.), simulation tasks (simulation report, update data, update display, exchange vehicles, transmit intersection info, transmit vehicles info, report positions of vehicles, give vehicle, etc.), load-balancing tasks (report load-balancing, move data, receive intersection, receive road, receive vehicle, receive statecycle, direct intersection, merge entryroad, merge exitroad, etc.). The intention is to minimize the communication overhead as it is known that sending messages is expensive and slows the performance of the parallel algorithm.

Node associated with its respective simulation region controls and moves vehicles within that region towards their destinations. A single vehicle whether remains within the boundaries of region $A$ or leaves the region on the map associated with node $A$ and continues in the neighbouring region associated with node $B$. In the latter case nodes communicate by sending messages.

### 3.4 Load-balancing

In distributed system with set of nodes capable to work in parallel we want to optimize the nodes’ usage while ensuring even amount of load, and therefore computation, for all nodes. Load-balancing is not an intrinsic part of the parallel algorithm, but an extension.

In the parallel traffic simulation we can achieve load-balancing with static distribution or dynamic redistribution of simulation load. Vehicles are referred to as simulation load. Vehicles may concentrate the computation on few nodes only while consequently other nodes show low usage ratio or even latency intervals. To avoid this, we have implemented dynamic load-balancing approach which uses the heuristics based upon node’s local information to determine the part of the entire road network that will be moved to another node.

Our algorithm uses simple criteria for evaluation of load imbalance, number of vehicles per node. The algorithm checks the number of vehicles on each participating node and finds out the nodes with the maximum, and the minimum number of vehicles.
maximum number of vehicles tells the load-balancing algorithm about the node (source) from which the reasonable amount of load has to be transferred to the node (destination) with the minimum number of vehicles.

Load-balancing forces nodes to more autonomous behaviour by having decision making capabilities that we have built in nodes, the agent [3]. Through these decision making capabilities nodes are able to share the load and co-ordinate their activities by means of communication. An agent is the function in node that responds to the message from host to node controlling the maximum number of vehicles. An agent locally and independently, without any prior knowledge, decides which intersection will be moved to the destination node. After the communication task that follows is completed the destination node takes over the control of the new intersection in its list. During the communication task the algorithm transfers all data that define the intersection including all entry and exit roads. We will describe both host-agent and node-agent in more details later in this report.

The essence of our load-balancing algorithm is disassociation of image space and nodes’ responsibility. The early distribution of data between nodes makes the correlation between the road network primitives (data space), their positions on the screen (image space) and the (static) distribution of responsibility for these primitives which remains unchanged until load-balancing starts.

Figure 2: Architecture and communication mechanism of the Parallel Traffic Simulator

4. Implementation of the Parallel Traffic Simulator

Parallel Computing Unit, at Griffith University, School of Computing and Information Technology, in collaboration with IBM, has built a simple parallel computer to demonstrate the concepts of parallel computing to the general public. Originally, the parallel system consisted of 10 personal computers; one is called the host and the other 9 are called nodes, because they do the actual work in parallel. The problem is one of modelling Brisbane’s traffic. The movement of vehicles is simulated, as well as the behaviour of traffic lights and uncontrolled intersections. In order to simulate all of Brisbane’s traffic at the individual vehicle level, it would be normally be necessary to use a very fast computer system [2].

Two versions of the Parallel Traffic Simulator have been implemented: one on groups of PCs and another on groups of networked workstations Silicon Graphics Indy. The programming model used is a host/node model. Both implementations make use of X-Windows and PVM. Figure 2 shows the three major software components of our implementation.

4.1 Networked workstations

A group of nine workstations Silicon Graphics Indy connected by an Ethernet has been used for this implementation. Each of these workstations is equipped with processor R4600SC/133MHz, rated at 73.7 SPECfp92, with 32 Mbytes of physical memory.

Networked workstations make up a loosely coupled parallel computer architecture which in our case uses PVM (Parallel Virtual Machine), a parallel program development environment, to implement message-passing for communication and synchronization between nodes. Real time simulation is generally not achievable in such an environment, but with minor changes the code can be ported to a massively parallel supercomputer.

The latest extension of the Parallel Traffic Simulator concerns study and implementation of the adaptive load-balancing algorithm.

4.2 PVM 3.0

PVM ([5]) is a software system that permits a network of heterogeneous UNIX computers to be used as a single parallel computer. Thus large computational problems can be solved by using the aggregate power of many computers. Under PVM collection of networked workstations appear as one large distributed-memory computer (virtual machine). Applications written in C, can be parallelized by using message-passing PVM constructs common to most distributed-memory computers.

To run a program under PVM the user first executes the daemon process on the local host computer which in
turn starts up daemon processes on all other node computers. The user’s application (the node program) which should reside on each node can then be invoked on each node by a local host program (the host program) via the daemon processes.

4.3 Data generator

Data generator is sequential process in the pre-processing phase where several programs are used to prepare data in a microscopic form suitable for use by Parallel Traffic Simulator.

![Figure 3: Structure of microscopic data for use by Parallel Traffic Simulator](image)

*Intersection* is a point (junction) in a road network graph consisting of pairs of entry- and exit-roads. *Istate cycle* has at least one *istate* and each of them has one or more *ipaths*. Istate cycle is essentially a traffic light. Each Istate opens its corresponding paths (ipath) for a specified time (green light interval - vehicles using this paths are allowed to move forward) and closes the previous ones (red light interval - vehicles on this path are requested to stop and wait for the next green light interval). Figure 3 shows the hierarchical structure of microscopic data and an example of an intersection with the active state istate3. Exploded situation shows intersection with three open paths (ipath1, ipath2, ipath3). This data, once in microscopic form, is then mapped into the parallel data structures on the host at program start-up. The data generation process needs typically to be performed once only to set up file with microscopic data for a given road network.

Figure 3 shows an extract of the road network traffic with the lanes outlined. Dashed lines with arrows and shaded show the lanes where the vehicles must stop and wait until green light goes on. Vehicles on each link (entry- and exit-road) compose a *queue of vehicles*. A *queue of vehicles* is the most variable data structure in nodes. When moving from one link to another (at junctions), vehicle is removed from one queue and inserted into another. When the node process starts it sets up these queues and fill them with data received from host.

4.4 Host process

According to the approach we settled on for our parallel algorithm the host process first reads data needed for the initialization of parallel environment. It then reads topology data for the given road network and builds its internal dynamic representation (frame). At the beginning majority of code here deals with intersection as it is the key structure and as such most complex of all. Data decomposition phase starts with the splitting of the previously created dynamic frame into the portions (subframes) used for the simulation according to the number of nodes in simulation. Then, clipping algorithms place intersections, roads and polygons within the boundaries of their corresponding nodes. Host then spawns node processes and sends initialization data to them. They remain idle until subframes to them. Now that host has successfully distributed data among nodes it can initialize X-Windows based application user interface. We use primitives from the X-Widgets library for user interface. Then host sends its first message (SCM_FULLDISPLAY) to all nodes. If we use standalone workstations as nodes then they are capable to display their parts of simulation on their local displays. Although we can optionally switch off displays rather then spend processing time for displaying simulation on nodes.

4.5 Node process

Host process launches node processes using PVM daemons in nodes. First, node process does initialization and receives basic information from the host that should be present before the simulation can be started. It then starts the simulation server which is an asynchronous message handler (non-blocking message polling loop) that allows node responding to both X and PVM events. PVM events start new tasks in node and may be generated either by host (i.e. loadbalancing report message) or by another nodes (i.e. receive car message). When a requested task has been completed, notification will be sent back to the sender.

4.6 Load-balancing algorithm

The algorithm, roughly described in previous sections, can be divided into four steps. The code for the first two steps reside on host, step 3 and 4 execute on node. In this context we introduced two individual agents: host-agent (step 1, step 2) and node-agent (step 3, step 4). Load-balancing is the interplay between these two agents. Host-agent receives request for balancing the simulation load at regular intervals. It then requests positions of vehicles from nodes by sending message SCM_REPORT_LOADBALANCING to them.
Immediately afterwards it calls the function where it gathers road co-ordinates from nodes and redisplays roads with node specific colours. The idea for host-agent is to find two nodes: the node with the maximum number of vehicles and the node with the minimum number of vehicles. If these two values diverge much from the calculated average value we can designate the simulation load as not well balanced. In this case the node with the maximum number of vehicles receives the message LB_LOADBALANCING_MOVE. Host-agent does not know what node-agent is actually going to do to contribute to good load-balancing.

Now, node-agent can start with step 3 of the algorithm. Node-agent expects from host-agent two integers which conform to the unique identifier of a node with the lowest workload during the current frame. In this step node-agent chooses single intersection among all intersections on the node. The simplest method for selection of intersection is the one based on random function. However, such approach reveals potential drawback of our algorithm. It does not guarantee that the selected intersection has the highest number of vehicles on its entry- and exitroads. It may even have none of them. If such situation occurs and intersection migrates to destination node, load distribution remains unchanged, but communication overhead increases. Finally, in step 4 node deploys PVM-routines to transfer the data structures associated with selected intersection to the destination node. Vehicles are transferred as well.

4.7 Visualizer

The Visualizer is a part of the application user interface designed in the X Window System programming environment that gives us the possibility to present a similar application user interface across all X-Windows based workstations. The Visualizer has three parts (Figure 4.): user menu based upon the X Toolkit library (Xt Intrinsics and Athena widgets), simulation window (parallel traffic simulator using road network in Brisbane) and load-balancing window.

User menu provides simulation control to the user and draws the graphs showing the current relative count of vehicles on each node. Simulation window simplifies the insight into simulation and load-balancing window enables better study of a load-balancing algorithm.

5. Future work

The main goal was to study and to implement the adaptive load-balancing for a traffic simulator. To illustrate the efficiency of such adaptive load-balancing approach we need to complete the tests that would show detailed performance and benefits of this method.

To evaluate the performance of the load-balancing algorithm we plan to compare the following three approaches:

- simulation without load-balancing,
- random load-balancing - node-agent selects intersection randomly, fragmentation of intersections could be high,
- local load-balancing - in order to decrease the fragmentation of intersections in node, node-agent tends to select the intersection which is close to the one previously selected

The efficiency of the load-balancing algorithm described above closely depends on the following components:

- host-agent algorithm (step 1: an algorithm for evaluation of load imbalance heuristics, step 2: an algorithm for determination of source and destination node),
- node-agent algorithm (step 3: an algorithm for selection of intersection, step 4: an algorithm for transfer of selected intersection)

We plan to refine these algorithms as they can significantly improve load-balancing policy.
References


[8] “PVM 3.0 USER’S GUIDE AND REFERENCE MANUAL”, Oak Ridge National Laboratory Oak Ridge, Tennessee 37831 (pvm@msr.epm.ornl.gov)