# AN EFFICIENT GVT COMPUTATION USING SNAPSHOTS

Myongsu Choe and Carl Tropper
School of Computer Science
McGill University
Montréal QC,Canada H3A 2A7
**Email**: msc@cs.mcgill.ca, carl@magic.cs.mcgill.ca

## ABSTRACT

We describe, in this paper, a snapshot-based algorithm which makes use of distributed control. The algorithm is an extension of Mattern's snapshot algorithm and employs a leader election algorithm to pick a GVT initiator. The leader election algorithm executes concurrently with the GVT algorithm. We prove the correctness of the algorithm and compare its performance to several algorithms - Samadi's, Bellenot's, and Mattern's original algorithm. In the performance comparison, we make use of a Shuffle Exchange Network and a PCS network, evaluating the simulation time, peak memory consumption, and the (control) message complexity of the algorithms. The performance of our algorithm in all three categories is far superior to Bellenot and Samadi's algorithms. It also provides a (less dramatic) improvement over Mattern's algorithm. Key words: Time Warp, GVT, snapshot, cut.

## 1 INTRODUCTION

A good deal of research has been done on obtaining global snapshots in distributed systems (Chandy and Lamport 1985; Lai and Yang 1987; Mattern 1993). The information afforded by a global snapshot can be useful for a variety of purposes such as termination detection, resource management, and debugging a distributed program. In an optimistic simulation it can find use in computing the Global Virtual Time (GVT) for memory management and for controlling the progress of the simulation. The GVT is the minimum of all of the local virtual times (LVTs) of the logical processes (LPs) in the simulation and of the timestamps of the messages in transit. Knowledge of the GVT allows fossil collection to take place and provides a mechanism for determining the termination of the simulation. Unfortunately, it is impossible to know the exact GVT at real time $t$ in the simulation without stopping the simulation. Consequently, it is necessary to compute approximations to the GVT as the simulation progresses.

We present a snapshot-based algorithm for the computation of GVT in this paper which is essentially an outgrowth of Mattern's algorithm (Mattern 1993). As such it does not rely upon a centralized GVT manager, thereby obviating the need to dedicate a large part of a processor's resources to the task and decreasing the formation of communication bottlenecks. Any processor involved in the simulation can initiate the algorithm. However, one processor is chosen as a GVT manager by a distributed leader election algorithm to record the GVT and broadcast it to the other processors. The algorithm may be used in a non-FIFO environment.

We compare our algorithm to well-known GVT algorithms - Samadi's (Samadi 1985) and Bellenot's algorithm (Bellenot 1990) as well as Mattern's original algorithm. In making the comparisons, we use a large switching network — a Shuffle Exchange Network (SXN) — and a Personal Communication Systems (PCS) network as our test beds. The number of LPs in both of networks ranges from 31,000-146,600. As we shall see, our distributed algorithm outperforms the above GVT algorithms.

The remainder of this paper is organized as follows. Section 2 contains a description of some well-known GVT algorithms, Section 3 contains a description of Mattern's algorithm, Section 4 contains a description of our algorithm and discusses its relation to Mattern's algorithm, Section 5 contains our experimental results, and Section 6 contains the conclusion.

## 2 GVT ALGORITHMS

GVT algorithms can be categorized as being either centralized or distributed. Centralized GVT algorithms make use of a central GVT initiator (or manager). The initiator sends requests to all of the LPs to ascertain each LP's LVT and to determine the minimum timestamp of messages in transit.

As mentioned before, the GVT is the minimum of

the local virtual times of the LPs and of the timestamps of the messages in transit. An approach to computing the minimum timestamp of messages in transit is to make use of acknowledgements; hence it is only necessary to compute the minimum timestamp of unacknowledged messages.

Since it is not possible for LPs to compute LVTs at the same instant in real time, it is necessary for the LPs to report their LVTs within some time interval in order to compute a meaningful GVT. This is accomplished in a number of the algorithms (Samadi 1985; Preiss 1989; Bellenot 1990) by utilizing intervals of time $[start_i, stop_i]$, one for each $LP_i$, such that each $LP_i$ computes its LVT within the interval. The GVT initiator is responsible for determining these intervals.

We briefly summarize several well-known algorithms with which we compare the performance of our own algorithm in this paper. In Samadi's algorithm (Samadi 1985), a central GVT manager sends a $start_i$ message to each $LP_i$. Each $LP_i$, in turn, acknowledges the *start* message with an acknowledgement message containing the $min\{vt_i\}$ where $vt_i$ = {timestamps in the input queue, timestamps of unacknowledged messages in the output queue}. Upon computing the GVT, the GVT initiator sends new GVT value to all of the LPs.

Bellenot's algorithm (Bellenot 1990) makes use of a ring topology. In the algorithm, the GVT initiator passes around a token signalling the start of the computation at each $LP_i$, followed by a second round in which each LP performs a *min* reduction on the $vt_i$ and includes the result in the token. Upon completion of the second round, the GVT initiator launches a third round in which the token contains the new value for the GVT.

A fundamental difficulty with the last two algorithms is the fact that the GVT initiator must determine when to launch the algorithms without input from the LPs. Unnecessary invocations for which the algorithm results in a minimal advance in GVT waste processing time and bandwidth in the form of control messages. On the other hand, long intervals between GVT computations can result in poor memory utilization.

# 3 GVT APPROXIMATION USING CUTS

The algorithm which we describe in this section is an extension of global snapshot algorithms based on the notion of obtaining a (consistent) cut. As such, we describe several tenets of those algorithms.

The objective of a global snapshot algorithm is to obtain the global state of a distributed system. The distributed system is presumed to adhere to causality, i.e. to preserve Lamport's "happened-before" relationship (Lamport 1978).

The notion of a cut underlies the construction of global snapshot algorithms (Mattern 1993). A cut (Figure 1) essentially divides the events of a system into those occurring before the cut and those occurring after the cut. Messages then travel between the "past" and the "future", as defined by the cut. A consistent cut is one in which no messages travel from the future into the past (Figure 1). Otherwise, we call the cut inconsistent. In order to obtain a global snapshot, local snapshots are gathered from individual processes "along the cut." In order for the global snapshot to be meaningful, it is necessary that the algorithm satisfies a consistent cut.

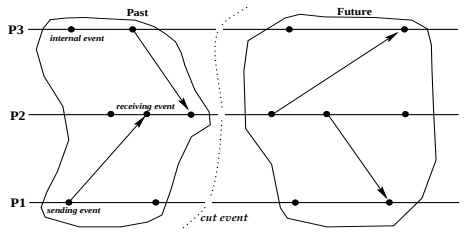



Figure 1: A time diagram with a cut

Lai and Yang (Lai and Yang 1987) developed an elegant algorithm for obtaining such a cut. Their algorithm applies to non-FIFO systems, and only invokes the piggy-backing of status information in one bit onto all messages. The algorithm is as follows: (1) Every process is initially white and turns black when taking a local snapshot. (2) Every message sent by a white (black) process is colored white (black). (3) Every process takes a local snapshot before a black message is received. Ensuring that a local snapshot is taken before a black message is received at a process is accomplished by examining messages for their color before processing them. In the event that a message is black, the local snapshot is taken prior to processing the message.

One way of implementing the algorithm is to circulate a control message which colors each of the visited processes black, i.e. upon receipt of a control message, a process colors itself black, as illustrated in Figure 2.

At the same time, the local state of the process can be appended to the control message (or sent directly to the process initiating the algorithm). However, it is possible that white messages are in transit while the local snapshots are being collected (Figure 2). Consequently, it is necessary to record the states of the channels. A way of doing this, suggested by Mattern, is for

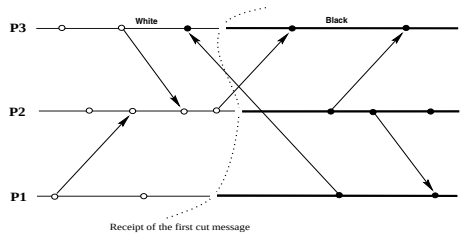



Figure 2: The receipt of a black message by a white process

black processes to send copies of these messages to the initiator and to use a termination detection algorithm to determine when they have all arrived.

Mattern describes a GVT algorithm which builds upon the notion of obtaining a consistent cut via the coloring technique described above. The basic principle of the algorithm is illustrated in Figure 3 from Mattern (Mattern 1993). Two cuts ($\mathcal{C}$ and $\mathcal{C}'$) are created by circulating a token.

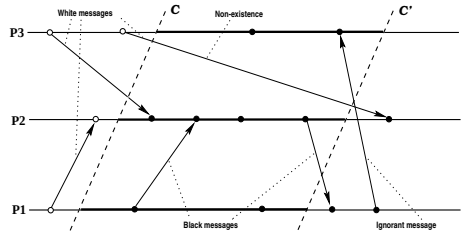



Figure 3: The GVT approximation principle

The reason for creating the two cuts is to avoid the possibility of a white message causing a rollback at a process after it reports its *lvt*. In order to avoid this problem, the second cut must be placed "far enough" to the right of the first cut. Mattern accomplishes this by appending a vector counter to the token which records the number of white messages sent to and received from other nodes already visited by the token. Upon receipt of the token by a GVT initiator, it can determine if all of the white messages which it sent were received and if it has received all of the messages sent to it. If so, then it can compute the GVT, otherwise a second round (i.e. cut) is necessary. The token also returns the minimum of the *lvts* of all of the processes and the minimum timestamp of all messages sent after $\mathcal{C}$.

# 4 THE ALGORITHM

The algorithm which we propose is based upon Mattern's algorithm (Mattern 1993) (discussed in the previous section). However, our algorithm differs from Mattern's algorithm in several important ways. In the first place, it makes use of a scalar counter as opposed to the vector counter used by Mattern. This is primarily because the use of a vector counter is time consuming, as the token must wait (during the second cut) at each process that the token visits until all of the messages sent to it have been received. This waiting can also result in a poor estimate for the GVT. Mattern's algorithm does not address the issue of selecting a process to launch the algorithm. Consequently, it is possible for all of the processes to launch the algorithm, a very expensive proposition. Our algorithm selects the GVT initiator via Chang and Roberts's distributed leader election algorithm (Chang and Roberts 1979, Tel 1994) from among the processes which wish to initiate the algorithm. Finally, it is important to point out that our algorithm collects information which can be useful for the implementation of other control functions such as load balancing or memory management (Choe 1998).

In the implementation of our algorithm, we identify the individual nodes of a multi-computer (upon which our Time Warp system is implemented) with the processes of Mattern's algorithm. We make use of one heap within each node to schedule events. The pseudo-code for our algorithm is contained in Figure 4.

Initially, all of the processors are *white*, and prior to a snapshot, the processor remains *white*. Each message which is sent by a white processor is colored white. Processors which are visited by control messages change their color to *black*. Messages which are sent by a black processor are colored black.

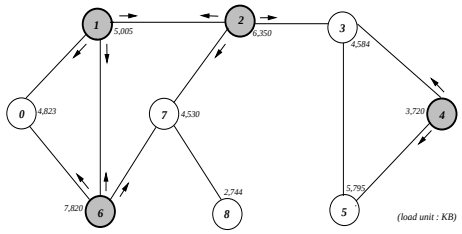



Figure 5: Election of GVT initiator (numbers next to the nodes represent the loads on a processor.)

Figure 5 depicts the coloring scheme of an 8 node network. Processors $p_1, p_6, p_2$, and $p_4$ initiate the coloring (or seeking a new GVT). We define the load of a processor as memory space occupied by the number of events and states since the previous GVT. Processors can initiate the coloring at a fixed frequency* (here, 1 sec.). In Figure 5, the shaded processors initiate the

*We initially allowed processors to initiate the coloring when their load exceeded a certain threshold. However, determining a proper threshold proved to be difficult. Too large a threshold resulted in too few GVT invocations, while too small a threshold resulted in too many control messages.

**var** $color_n$: boolean **init** white, $color$: boolean **init** white;
  $pid_n$: integer **init** unique, $pid_{init}, pid_n^{init}$: integer **init** undef;
  $lvt_n$: integer **init** 0, $lvt$: integer **init** $\infty$;
  $ts_n$: integer **init** $\infty$, $ts$: integer **init** $\infty$;
  $load_n$: integer **init** 0, $load$: integer;
  $count_n$: integer **init** 0, $count$: integer **init** 0;
  $gvt$: integer **init** $\infty$;
  $ctlmsg$: $\{pid_{init}, lvt, ts, count, color, load\}$ **init** $\phi$;
  $m \in Pred_n, l \in Succ_n$;

$S_n$: (* sending control message *)
**begin**
**(1) if** $color_n = white$ **and** $GVT\text{-}firing$ **then**
**begin**
$color_n$ := black, $pid_{init} = pid_n^{init} := pid_n$;
$load_n := sizeof(event) * |events| + sizeof(state) * |states|$;
record local states $(lvt_n, load_n)$;
**end**
**(5) forall** $l \in Succ_n$ **send** $\ll pid_{init}, min(lvt_n, lvt), min(ts_n, ts), count_n + count, load \gg$ **to** $l$;
**end**
$R_n$: (* receiving control message *)
{A message $\ll ctlmsg \gg$ has arrived.}
**begin**
**(1) receive** $\ll pid_{init}, lvt, ts, count, load \gg$ **from** $m$;
**(2)** compute min$(lvt_n)$ at processor $n$;
**(3) if** $color_n = white$ **then**
**begin**
$color_n$ := black, $pid_n^{init} := pid_{init}$;
**send** $\ll pid_{init}, min(lvt_n, lvt), min(ts_n, ts), count_n + count, load \gg$ **to** $l$;
**end**
(* $p_n$ is black, hence it is already participating in the election *)
**(6) if** $color_n = black$ **and** $pid_{init} \neq pid_n$ **then**
**if** $load_n \geq load$ **then** stop snapshot from $m$;
**else**
**begin**
$pid_n^{init} := pid_{init}$;
**send** $\ll pid_{init}, min(lvt_n, lvt), min(ts_n, ts), count_n + count, load \gg$ **to** $l$;
**end**
**endif**
**(10) if** $color_n = black$ **and** $pid_{init} = pid_n$ **then**
**if** $count_n + count = 0$ **then**
**begin**
$gvt := min((lvt_n, lvt), (ts_n, ts))$;
$color_n$ := white, $pid_n^{init} := undef$;
**send** $\ll pid_n^{init}, gvt, color_n \gg$ **to** $m, l$;
**end**
**else**
**begin**
**send** $\ll min(lvt_n, lvt), min(ts_n, ts), count_n \gg$ **to** $l$;
$count_n = count := 0$;
**end**
**endif**
**end**
$S'_n$: (* sending basic message *)
**begin**
**(1) if** $color_n = white$ **and** $sending\text{-}event$ **then** $count_n := count_n + 1$;
**(2) if** $color_n = black$ **then** compute min$(ts_n)$;
**(3) send** $< msg, color_n >$ **to** the receiving processor;
**end**
$R'_n$: (* receiving basic message *)
{A message $< msg, color >$ has arrived.}
**begin**
**(1) receive** $< msg, color, >$ **from** the sending processor;
**(2) if** $color = white$ **then** $count_n := count_n - 1$;
**(3)** process the message;
**end**

Figure 4: GVT Computation (at processor $n$)

coloring. Each processor sends a control message to all of its (graph-theoretic) descendants. We discuss the contents of the control message and their functions below.

When a control message arrives for the first time at a processor $p_n$, the processor is colored black, set the $pid_{init}$ in the cut message to the $pid_n^{init}$ (a GVT initiator known at $p_n$), and then passed to the descendant processor. Otherwise, the load of the processor is compared to the load on the sending processor and in the event that it is larger, it does not forward the control message of the sending processor, but instead sends its own load to its descendants. In the event that the load on the processor is smaller than the load included in the control message, it forwards the control message and does not participate in election any more. Ultimately, the initiating processor with the largest load is selected as the "GVT initiator." The control message makes use of the following variables:

$pid_n$: the unique identifier of processor $n$.
$pid_n^{init}$: the identifier of the GVT initiator (known at $p_n$).
$lvt_n$: local virtual time = $min\{lvt$s of all of the LPs on processor $n\}$.
$ts_n$: minimum timestamp of black message(s) = $min\{ts$ of all of the black messages sent by processor $n\}$.
*count*: a scalar counter used to determine if white messages are in transit.
$color_n$: the color of processor $p_n$, initially set to white.
$load_n$: the memory occupied by events and states since the previous GVT.

A $count_n$ field maintained at each processor keeps track of the number of white messages which are sent and received by the processor. Each time a white message is sent, it is incremented by 1 and each time a white message is received, it is decremented by 1. The control message maintains the (partial) sum of the count field of the processors it visits in the variable *count*. Upon receiving its own control message, the GVT initiator checks the count variable in order to determine if another GVT round is necessary. If $count = 0$, it propagates a new GVT to the network. Otherwise, it launches another GVT round (recall that all of the other processors will not launch a GVT computation, as they are "eliminated"). When $count = 0$, all of the white messages sent by processors have been received.

On its way, the control message also computes the minimums of the *lvts* and the *ts* of the nodes it has visited, $min(lvt_n, lvt)$ and $min(ts_n, ts)$, and records them in the *lvt* and *ts* field. While it is entirely possible for multiple launchings of the GVT token before a GVT is actually computed, our experimental experience is that two rounds are generally sufficient.

## 4.1 Correctness Proofs

We prove the correctness of the GVT algorithm by establishing the safety and the liveness of the algorithm. Safety corresponds to the algorithm producing an estimate which is less than (or equal to) the exact GVT. Liveness corresponds to the algorithm producing monotonically increasing estimates.

We first establish the safety property. Let $GVT(t)$ be the exact GVT at time $t$ and $\widetilde{GVT(t)}$ be the approximate GVT as computed by our algorithm at time $t$.

THEOREM 4.1 (SAFETY) Let $t$ be the instant at which $\widetilde{GVT(t)}$ is computed. Then $\widetilde{GVT(t)} \leq GVT(t)$.

PROOF. $\widetilde{GVT}$ is computed by the initiator $\Longleftrightarrow$ $count = 0$. (line 11 in $R_n$ of the algorithm) $count = 0 \Longleftrightarrow$ there are no white messages in transit. Hence, we need only concern ourselves with the timestamps of black messages in transit when computing the GVT, i.e. $GVT(t) = min\{lvts$ of all LPs at time $t$, timestamps of black messages in transit at time $t\}$. From the algorithm,

$$\widetilde{GVT(t)} = min\{min(lvt), min(ts)\}$$

where $min(lvt)$ = minimum of the *lvt*s for all of the LPs and $min(ts)$ = minimum timestamps of all of the black messages since each LP became black. The $min(ts) \leq$ timestamps of all black messages in transit at $t$ since the black messages in transit at time $t$ form a subset of all the black messages sent since each LP became black. Furthermore, at time $t$ no *lvt* can be less than the minimum timestamp of the black messages in transit at time $t$. (These are the only messages which can roll back an LP since the token has visited all of the LPs except the initiator prior to time $t$.)

Hence, we conclude that $\widetilde{GVT(t)} \leq GVT(t)$. ■

We now establish the liveness of the algorithm.

THEOREM 4.2 (LIVENESS) If $t_1 < t_2$, then $\widetilde{GVT(t_1)} \leq \widetilde{GVT(t_2)}$.

PROOF. After the computation of $\widetilde{GVT(t_1)}$, it is possible for one of the LPs to be rolled back by a black message, but not by a white message (the white messages have all arrived). However, the minimum timestamp of the black messages in transit is included in the definition of $\widetilde{GVT}$ and by virtue of this definition, the $\widetilde{GVT(t_2)}$ cannot decrease subsequent to the computation of $\widetilde{GVT(t_1)}$. The theorem follows. ■

THEOREM 4.3 Processor coloring and choosing a GVT initiator in the course of processor election can be achieved within finite time.

PROOF. Suppose that the channels in the network have finite transmission time, that transmission is fault-free, and that a processor takes finite time $\delta$ to be colored. If all of the processors begin to color at the same instant, the time for coloring will be $\delta$. Otherwise, if the processors are colored sequentially, in the worst case, it takes $N\delta + \epsilon$, where $\epsilon$ is the time for the token to traverse the network and $N$ total number of processors participating in the simulation. Therefore, choosing a GVT initiator requires time $\leq N\delta + \epsilon$. ■

### 4.2 Complexity Analysis

In our algorithm, we assume that there are $m$ initiating processors out of $n$ processors. One of the $m$ initiating processors is elected as the GVT initiating processor. In the worst case, every processor launches the GVT computation at the same instant and the complexity of coloring is $O(n^2)$. Therefore, the coloring phase has the same complexity as the coloring phase in Mattern's algorithm, but the collecting phase behaves like the centralized approach because there exists only one initiating processor chosen among the processors. Thus, the collecting phase is $O(n)$, the same as the centralized approach. On the other hand, the complexity in the collecting phase of Mattern's algorithm is the same as the one in the coloring phase.

If we assume that the topology of the network is a ring and the unidirectional leader election algorithm proposed by Chang and Roberts (Chang and Roberts 1979; Tel 1994) is used, the complexity of the coloring corresponds $\Theta(n^2)$ order of messages in the worst case and $O(nlogn)$ in the average case. Mattern (Mattern 1989) also provided interested results showing that quadratic message complexity is an exceptional case and that the worst case estimate is rather conservative. Our simulation results support this point as our empirical results point to a complexity in the coloring phase of $O(nlogn)$. The centralized approach has a complexity of $O(n)$ in both phases. Table 1 summarizes the message complexities of each control type.

## 5 SIMULATION RESULTS

### 5.1 Test bed

We modified and tested our algorithm making use of a Time Warp simulator built by Avril (Avril and Tropper 1995). The Simulator was originally developed on a BBN Butterfly GP1000, a MIMD (Multiple Instruction stream-Multiple Data stream) machine with shared memory architecture consisting of 32 processors. Even though it was developed on a shared memory system, implementation was developed for supporting *send*() and *receive*() non-blocking primitives under the message passing system. We did not exploit the machine-dependent advantages of the shared memory, hence there are no declared global variables in each processor. Only channels connecting processors and an input/output buffer for each processor are declared and allocated in globally shared memory.

We ported the simulation suites onto the Silicon Graphics PowerChallenge super-computing server. The PowerChallenge system is a MIMD machine based on shared memory consisting of homogeneous processors (called Symmetric Multi-Processors (SMP)) interconnected by high-speed buses. The system has 64-bit MIPS super-scalar RISC processors providing 360 double precision MFLOPS and 360 MIPS/90 Mhz and 64 MB of memory for each processor.

### 5.2 Applications

#### 5.2.1 Shuffle Exchange Network

We simulated three shuffle exchange networks - a 100x100, a 165x165, and a 200x200 SXN. The networks had a total of 36,951, 100,102 and 146,672 LPs respectively.

The Shuffle Exchange Network (Maxemchuk 1989; Robertazzi 1993) is a cylindrical multi-stage network with an in- and out-degree of two. For a stress test, we modified the original shuffle exchange network by interconnecting nodes in the first and last column and substituting one of input buffers with a local source and one of output buffers with a local sink, as seen in Figure 6. Nodes in each row are connected in a ring, thus this topology is also called a shuffle ring network. The structure of each node is shown in Figure 7. Except for nodes with local source and sink in the first and last column stage, ranging from $1 - 2\%$ of the switching nodes, the remainder of the nodes do not have local source or sink, but have links with adjacent nodes.

#### 5.2.2 PCS Network

A PCS is a wireless communication network which provides communication services for mobile users (Li and Qiu 1995; Alleyne 1997). The service area is tessellated with a small service area called a cell. Each cell has a transmitter with a fixed number of channels. In general, a regular hexagon is used to represent a cell.

Table 1: Comparisons of Message complexity

| Control type | Algorithm | Coloring | Collecting |
|---|---|---|---|
| Centralized | - | $O(n)$ | $O(n)$ |
| Decentralized | Mattern | $O(n^2)$ | $O(n^2)$ [a] |
| Partly distributed | Ours [b] | $\Theta(n^2)$ | $O(n)$ |

[a] We assumed that a leader election algorithm is not used to pick an initiator. Otherwise, the collecting complexity is $O(n)$.

[b] Our algorithm may be considered to be partly distributed because a GVT initiator is chosen by a leader election algorithm, while Mattern's algorithm allows all processes to initiate at once. This is clearly inefficient.

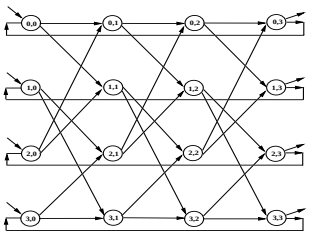


Figure 6: 4x4 Shuffle Exchange Network

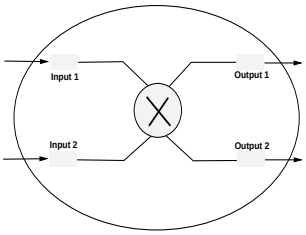



Figure 7: Structure of a switching node in SXN

When a user makes a call, a channel is assigned to the calling user in order to communicate with another. If all of the channels are allocated, then the call is blocked. If a user moves from one cell to another one during a call, a new channel from the cell is allocated to provide for the continuous connection of the call. This channel reassignment is said to be a hand-off. If all of the channels are still busy, the user's call has to be terminated.

In simulating the PCS network, due to the use of finite size of cells, a "boundary effect" may exist. The cutting-off of the simulation at the edge of the simulated service area can affect the simulation results. If a mobile user crossed outside the simulated area, the user could disappear or appear at the boundary edge in the opposite direction.

In general, a simulation using less than 50 cells may be subject to the boundary effect. Lin and Mak (Lin and Mak 1994) suggested wrapping the hexagonal mesh (H-mesh) into a homogeneous graph with an in- and out-degree of six rather than square mesh (S-mesh) and showed that inaccuracy of the simulation result can be significantly reduced in this case. Let the dimension $n$ of a mesh be the number of nodes on each peripheral edge of the mesh. An hexagonal mesh with dimension three is illustrated in Figure 8 where each boundary cell is not shown to have six degrees.

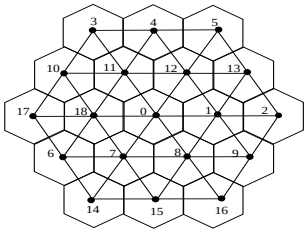



Figure 8: Wrapped Hexagonal Mesh with n=3

A phone call is modeled by an event and a cell is simulated by an LP. Each cell has fixed number of channels. For simplicity, we do not consider a dynamic channel assignment scheme. We suppose that each cell has 500 fixed channels. A call is further classified into a static call or a mobile one depending upon initial conditions such as initial velocity or direction when it is created. The velocity of the mobile call is assumed to be constant.

In our simulation, each call can have 0-30 Km/hr range of velocity and 7 directions: (1) East, (2) South East, (3) South West, (4) West, (5) North West, (6) North East, and (7) Still. The velocity and direction is determined by using a uniform distribution and the call completion time is determined by an exponential distribution with 300 seconds mean time. A cell diameter is 1 km. A mobile call is created at the center of the cell. Contrary to this, a static call can be created in a range of position in the cell. If the mobile call is generated at a boundary cell, the call will appear to another cell located in an opposite direction without vanishing. The call arrival to a cell including incoming and generating calls follows a Poisson distribution. We simulated a PCS network H(60) with 31,864 LPs.

## 5.3 Experimental Results

In order to evaluate the performance of our algorithm we compared it to Bellenot's algorithm (Bellenot 1990), Samadi's algorithm (Samadi 1985), and Mattern's algorithm (Mattern 1993).

Two variants of our algorithm called Snapshot(1) and Snapshot(2) were developed. In Snapshot(1) if, after the GVT computation, the GVT initiating processor has an above average load, the processor remains the initiator, while in Snapshot(2) the GVT initiator gives up its initiating rights and a new election for the GVT initiator ensues shortly thereafter. As a first step, we compared the performance of these two algorithms to each other and to that of Mattern's original algorithm.

In implementing Mattern's algorithm, we adopted a centralized approach instead of using a fully distributed one, thereby avoiding an election algorithm and providing a best case comparison for Mattern's algorithm. After determining that Snapshot(1) was the most efficient of these algorithms, we compared it to the algorithms mentioned above.

In making these comparisons, we made use of the following measures of their performance: (1) Message complexity, defined to be the number of control messages used by the algorithm. (2) Maximum memory used per process during the course of the simulation. This is an average over all of the processors employed in the simulation. (3) Simulation execution time.

In order to ensure that the comparisons were fair, we set the GVT interval to one second, i.e. after computing a new GVT value, another GVT computation is initiated at one second later.

### 5.3.1 Snapshot comparison

Table 2 contains a comparison of the number of control messages sent by both of the snapshot algorithms as well Mattern's algorithm for a 100x100 SXN. We used 6 processors in the experiment. As we can see in the table, Snapshot(1) uses some 75% fewer control messages than does Snapshot(2). Mattern's algorithm makes use of some 18% more control messages over the course of the simulation than does Snapshot(1).

Figure 9 portrays the simulation times of Snapshot(1), (2), and Mattern's algorithm for the 100x100 SXN as a function of the number of processors. As depicted by the figure, the performance of Mattern's algorithm is intermediate between that of Snapshot(1) and (2). The (percentage) difference in the simulation times between Snapshot(1) and Mattern's algorithm varies between 4-13% over the range of processors,

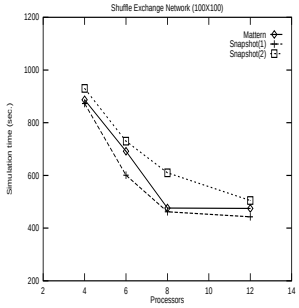



Figure 9: Simulation time vs. number of processors for SXN 100x100

while Snapshot(2) lagged behind Snapshot(1) by 8-24%.

### 5.3.2 Message Complexity

We compared the number of control messages used by Bellenot's algorithm, Samadi's algorithm, Snapshot(1) and (2) for a simulation of a 100x100 SXN, using 6 processors. The results are shown in Table 2, which contains both the number of control messages used and the percentage difference between Snapshot(1) and the other algorithms. As we can see, Snapshot(1) is clearly superior to the other algorithms by a large margin, using 62 and 65%, respectively, of the control messages of Bellenot's and Samadi's algorithms.

Figure 10 portrays the reason for these differences, in terms of the average number of rounds which a token message is required for each GVT computation. As we can see, in both Bellenot's and Samadi's algorithms, the token is constrained to make two rounds, while both Snapshot(1) and Mattern's algorithm can use fewer number than two rounds to compute the GVT.

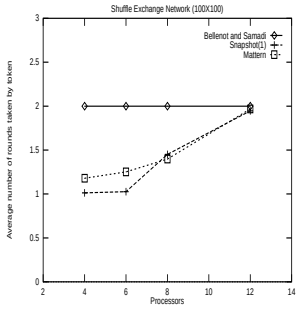



Figure 10: Average number of cut messages taken for each GVT computation

The difference in the message complexity of Mattern's algorithm and our own algorithm may be found in the condition **wait until** $V[i] + count[i] \leq 0$. This

Table 2: Number of Control Messages at SXN 100x100

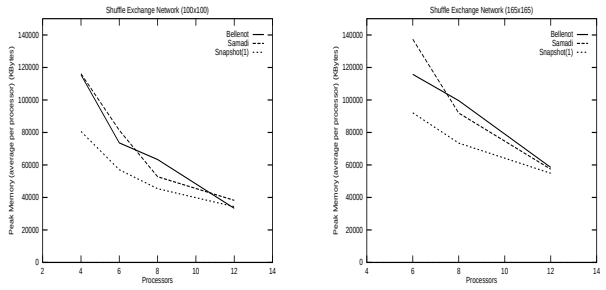
| processors | *Bellenot* | | *Samadi* | | *Snapshot(1)* | *Snapshot(2)* | | *Mattern* | |
|---|---|---|---|---|---|---|---|---|---|
| | # msgs. | % [a] | # msgs. | % | # msgs. | # msgs. | % | # msgs. | % |
| 6 | 17,600 | 62 | 17,899 | 65 | 10,858 | 19,008 | 75 | 12,832 | 18 |

[a]The figure derives from the comparison of Snapshot(1).

condition causes some delay at each process, thus resulting in more simulation time and the use of more control messages.

### 5.3.3 Memory Usage

Figures 11 (a) and (b) contain graphs of the average of the peak memory used by all of the processors involved in the simulation for a 100x100 and a 165x165 SXN respectively. Both of these figures very clearly indicate that Snapshot(1) results in substantial savings in memory compared to the other algorithms. For the 100x100 SXN, the difference between the memory consumption of Snapshot(1) and each of the other algorithms is approximately 30% when 4 or 6 processors are employed. At 8 processors, Samadi's algorithm exhibits a 14% difference, while the other two algorithms continue to have a 30% difference. Only when we reach 12 processors, we find a narrowing in the gap of memory utilization. Here Bellenot's algorithm uses approximately the same amount of memory, while Samadi's differs by 11%. A similar performance may be observed for the 165x165 SXN.




(a) SXN 100x100 (b) SXN 165x165

Figure 11: Peak memory usage for SXN 100x100 and SXN 165x165

### 5.3.4 Execution Time

We present the simulation times for Bellenot's algorithm, Samadi's algorithm, and Snapshot(1) in Figure 12 (a) for a 100x100 SXN. The same comparisons are made for a 165x165 SXN in Figure 12 (b). For the 100x100 SXN, we see that Snapshot(1) has the smallest execution time of the group. At 4 processors, it is approximately 10% faster than the other algorithms, while at 6 processors this difference rises to about 18%. The differences are about the same for 8 processors, while at 12 processors Bellenot's and Samadi's algorithms are only slightly slower than Snapshot(1). The differences in execution time for the 165x165 SXN are greater. At 6 and 8 nodes, all of the differences are in the neighborhood of 18%, while at 12 nodes, the range is 5-12%.

We simulated a 200x200 SXN using 6 processors. Bellenot's algorithm was approximately 60% slower than Snapshot(1), while Mattern's algorithm was 16% slower. The simulation using Samadi's algorithm did not complete.

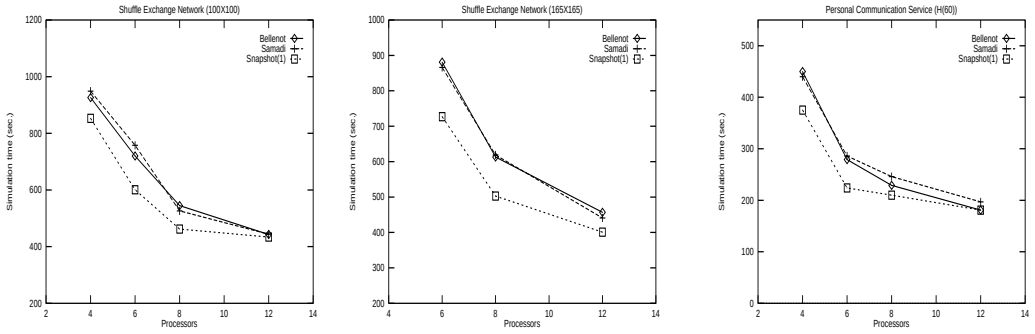
Figure 12 (c) contains a comparison of the above algorithms in the context of a simulation of a PCS network. The execution times for the same algorithms are plotted in the Figure 12 (c) for 4-12 processors. For 4 processor, the differences in execution time range between 15 and 24%, while at 6 processors the differences are between 20 and 23%. After this point, the difference between the execution time of Snapshot(1) and the other algorithms decreases.

# 6 CONCLUSION

We presented, in this paper, a distributed algorithm for the computation of GVT in optimistic simulations. The algorithm is based on a global snapshot algorithm developed by Mattern. It employs a leader election algorithm to determine a GVT initiator in the event that more than one LP initiates the algorithm.

Two versions of the algorithm (Snapshot(1) and (2)) were developed and compared with Mattern's algorithm in the context of a Shuffle Exchange Network simulation. We noted that Snapshot (1) used some 75% fewer control messages than Snapshot(2) and some 18% fewer messages than Mattern's algorithm. Predictably, the execution time of Mattern's algorithm was intermediate between Snapshot(1) and (2).

We then compared Snapshot(1), Bellenot's




(a) SXN 100x100 (b) SXN 165x165 (c) PCS H(60)

Figure 12: Simulation time vs. processors

algorithm, and Samadi's algorithm. Our test beds were a 100x100 SXN, a 165x165 SXN, a 200x200 SXN, and a H(60) PCS network. We compared the algorithms on the basis of their message complexity, peak memory usage, and their execution time.

In terms of message complexity, Snapshot(1) is far superior to the other GVT algorithms using 62% fewer messages.

Our results pointed out a large difference in the peak memory consumption between Snapshot(1) and the other algorithms. For example, in both the 100x100 and 165x165 SXN's, the peak memory consumption of Snapshot(1) was approximately 30% less than that of the other algorithms for 4 processors. For obvious reasons, as more processors are employed on the same models, the gap in the memory usage decreased.

The differences between Snapshot(1) and the other algorithms in execution time for the 100x100 and 165x165 SXNs are not as pronounced as they are for memory usage and message complexity. For both of these models, Snapshot(1) is approximately 10-18% faster than the other algorithms. The difference in execution time for a 200x200 SXN were larger - Snapshot(1) was 60% faster than Bellenot's algorithm. For the PCS simulation, we see a difference of 15-24% between Snapshot(1) and the other algorithms.

Our results indicate that Snapshot(1) makes a smaller peak memory demand and uses fewer control messages than Bellenot's and Samadi's algorithm. An interesting extension of this research would be to compare the same algorithms on a network of workstations as the memory limitation on a network of workstations would be far more severe than on a PowerChallenge system. Under any circumstances, our results indicate that Snapshot(1) would appear to be a good choice as a GVT algorithm.

# REFERENCES


Alleyne, P. 1997. Distributed Channel Allocation Simulation. Master's thesis, School of Computer Science, McGill University.

Avril, H. and Tropper, C. 1995. "Clustered Time Warp and Logic Simulation." In *Proceedings of the 9th Workshop on PADS*, 112–119.

Bellenot, S. 1990. "Global Virtual Time Algorithms." In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, 122–127.

Chandy, K. and Lamport, L. 1985. "Distributed Snapshots: Determining Global States of Distributed Systems." *ACM Transactions on Computer Systems*, 3(1): 63–75.

Chang, E. and Roberts, R. 1979. "An improved algorithm for decentralized extrema finding in circular arrangements of processes." *Communications of the ACM*, 22: 281–283.

Choe, M. 1998. Distributed Process Cooperation in a Rollback-based Simulation. Ph.D. dissertation in preparation, School of Computer Science, McGill University.

Lai, T. and Yang, T. 1987. "On Distributed Snapshots." *Information Processing Letters*, 25(1): 153–158.

Lamport, L. 1978. "Time, Clocks, and the Ordering of Events in a Distributed System." *Communications of the ACM*, 21(7): 558–565.

Li, V. and Qiu, X. 1995. "Personal Communication Systems (PCS)." *Proceedings of the IEEE*, 83(9): 1210–1243.

Lin, Y-B. and Mak, V. 1994. "Eliminating the Boundary Effect of a Large-Scale Personal Communication Service Network Simulation." *ACM Transactions on Modelling and Computer Simulation*, 4(2): 165–190.

Mattern, F. 1989. "Message Complexity of Simple Ring-based Election Algorithms - An Empirical Analysis." In *Proceedings of the 9th International Conference on Distributed Computing Systems*, 94–100.

Mattern, F. 1993. "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation." *Journal of Parallel and Distributed Computing*, 18: 423–434.

Maxemchuk, N. 1989. "Comparison of Deflection and Store-and-Forward Techniques in the Manhattan Street and Shuffle-Exchange Networks." In *Proceedings of IEEE Infocom.*, 800–809.

Preiss, B. 1989. "The Yaddes distributed discrete events simulation specification language and execution environments." In *Proceedings of SCS Multiconference on Distributed Simulation*, 21:139–144.

Robertazzi, T (editor). 1993. "Performance Evaluation of High Speed Switching Fabrics and Networks." 301-360. IEEE Press.

Samadi, B. 1985. Distributed Simulation, Algorithms and Performance Analysis. Ph.D. dissertation, Computer Science Department, University of California, Los Angeles.

Tel, G. 1994. "Introduction to Distributed Algorithms." chapter 7, 229-232. Cambridge University.