Global Virtual Time Algorithms

Steven Bellenot
The Florida State University and
The Jet Propulsion Laboratory

Abstract

A new *GVT* (Global Virtual Time) Algorithm is given and compared to known *GVT* algorithms. This algorithm is best possible in the sense it has an overall run time of $O(\log N)$, a run time of $O(1)$ on each node and also it sends less than $4N$ messages. Performance data for both the new and old *GVT* algorithms with TWOS (JPL's implementation of Time Warp) are included.

Introduction

Our setting is Time Warp (Jefferson 1985), an optimistic synchronization method for distributed discrete event simulations. Time Warp has all the different nodes (processors) each executing simulation objects at (possibly) different virtual times (or timestamps). These objects schedule events for each other by sending messages. Unlike conservative methods, the Time Warp does not block, on each node it executes the object with the earliest timestamped on that node. Hence many nodes could be doing incorrect work. Time Warp saves enough information so that incorrect work can be rollbacked and redone. In theory, the entire history of a simulation could be saved, but in practice the memory used to save this history needs to be reclaimed.

The term *Global Virtual Time (GVT)* is used for two different (but related) quantities in Time Warp. In the abstract, *GVT* is an instantaneous parameter of a distributed simulation. It is calculated by first (at least mentally) freezing all the nodes and messages in transit. The value of *GVT* at that instant is the virtual time of the object or message in transit which is furthest behind. This abstract *GVT* represents the progress of the simulation. Everything with timestamp before this abstract *GVT* is correct computation. Everything after this abstract *GVT* is subject to change. Since this abstract *GVT* is impossible to measure on the fly, *GVT* (estimated *GVT* or *GVT* with no adjective) also refers to lower bound estimates

of the abstract *GVT*. Knowledge of *GVT* is necessary for both garbage collection of the past (fossil collection or the reclaiming of memory), commitment of output and even for detecting when the simulation has ended (Jefferson 1985). (*GVT* is also used in the proof that simulations using Time Warp will eventually terminate, indeed this proof just shows *GVT* always increases.)

Three reasons motivated considering *GVT* algorithms again. First, the number of nodes (processing elements) available to run with TWOS (Jefferson et al.1987), the Jet Propulsion Laboratory's implementation of Time Warp have increased. Hence the cost of non-scaling algorithms becomes more important. Second, the main platform for the TWOS development group had switched from the JPL Mark III Hypercube, which supported broadcasts, to the BBN Butterfly Plus, which does not. Third, the old *GVT* algorithm was known to cause long message delays during the collection phases even on the Mark III (Bellenot and Di Loreto 1989).

In a sense, the speed of the *GVT* algorithm hasn't been critical to TWOS. The *GVT interval* in TWOS is a run time parameter which determines the time between successive *GVT* calculations, the time from the end of one *GVT* calculation to the start of the next one. The run time of well-behaved simulations have shown no noticeable changes as the *GVT interval* varied from two to eight seconds. (The run time could suffer with smaller or larger times.) However, the choice of the *GVT* algorithm does become important in the less well-behaved cases as we shall see.

GVT Estimating Algorithms in General

We assume there are N nodes, numbered from 0 to $N-1$. Estimated *GVT* uses real-time intervals instead of the single real-time instant used for finding abstract *GVT*. For each node i, let $[START_i, STOP_i]$ be an interval of real time. We further require that the collection of these intervals $[START_i, STOP_i]$ have at least one point, call it *RTM* (for real-time moment), in common. Each node i computes MVT_i to be the minimum of the all the timestamps of messages either in transit from node i at the real-time $START_i$ or send in the real-time interval $[START_i, STOP_i]$. On node i, PVT_i is the timestamp of the furthest behind object at real-time $STOP_i$ and LVT_i is the minimum of MVT_i and PVT_i. Finally, (estimated) *GVT* is

the minimum of all the LVT_i's. Now on node i, MVT_i is no bigger then the minimum timestamp of all messages send from node i, which are in transit at real-time RTM. The value of PVT_i is measured at real-time $STOP_i$. Note PVT_i can be strictly bigger than the timestamp of the farthest behind object at real-time RTM, however any side-effects of running objects during the real-time interval $[RTM, STOP_i]$ are included in the MVT_i measurement. (Thus the estimated GVT can actually be larger than the abstract GVT at the real-time RTM.)

It is the receive time timestamp of messages in transit which are minimized to obtain MVT_i. The side effects of the arrival of a message will happen at its receive time. However, in TWOS, reverse messages (messages returned to their senders) are used for flow control. Reverse messages rollback their senders. Thus in TWOS we minimize the send time timestamps of messages in transit.

The Old GVT Algorithm

The old GVT algorithm explicitly defined these real-time intervals $[START_i, STOP_i]$ by receipt of certain messages. The old GVT algorithm ran in five phases:

(1) Node *0* broadcasts (sends the same message to all the nodes) a GVT start message to all the nodes. On node i, the receipt of a GVT start message is the real-time $START_i$.

(2) Each node responses to this GVT start message by sending a GVT ack message to node *0*. Node *0* receives a GVT ack messages from all nodes. (We call this operation a *collection*, it is the inverse operation to broadcasting.) After node *0* has received a GVT ack message from all the nodes, we are at the real-time instant RTM.

(3) Node *0* broadcasts a GVT stop message to all the nodes. On node i, the receipt of a GVT stop message is the real-time $STOP_i$.

(4) Node i responses to this GVT stop message by sending a GVT lvt message to node *0* which contains LVT_i. Node *0* receives a GVT lvt messages from all nodes. (This a second message collection.) After node *0* has received a GVT lvt message from all the nodes, then it computes GVT by minimizing all the LVT_i's.

(5) Node *0* broadcasts a *GVT* update message to all the nodes. This message contains the new estimate of *GVT*.

Note that for both the collections (2) and (4), node *0* must sequentially receive a message from all *N* nodes, hence the run time of the old *GVT* algorithm is $O(N)$.

The run time of broadcasts depends on the hardware support and the network topology. The Mark III hypercube supported a broadcast in which separate communication processors did all the forwarding. For this hypercube, broadcasts had run time $O(1)$ on each node and run time $O(\log N)$ overall. In a shared memory machine, like the BBN Butterfly Plus, broadcasts can be done in $O(1)$. However, TWOS remains true to message passing origins and does not take advantage of shared memory. Each broadcast on the butterfly requires the sending node to "send" an integer to all other nodes which has run time $O(N)$. Thus $5N$ *GVT* type messages are received in the old *GVT* algorithm, and between $3N$ and $5N$ messages are sent.

It is possible to replace the broadcasts and the collections with *binary-tree-like forwarding*. Instead of broadcasting to each node, node *0* would sent to both node *1* and *2*, and in general node i would forward the message to nodes $2i+1$ and $2i+2$ (if these numbers were less than N). A collection can be done in the reverse order and computation can be distributed at the same time. In phase (4), node i could take the estimates from nodes $2i+1$ and $2i+2$ and minimize it with LVT_i and pass this estimate on to node $(i-1)/2$ (if $i > 0$). Using these binary-tree-like forwarding instead of broadcasts and collections, the run time of the algorithm drops to $O(\log N)$. The new *GVT* algorithm uses a variation of these ideas.

Token Passing GVT Algorithms

Token passing *GVT* algorithms have also been used. Before describing the token algorithm, we look at a ring topology algorithm. The ring topology algorithm works in three phases:

(1) Node *0* starts the *GVT* calculation and sends a *GVT* start message to node *1*. On node i, the receipt of a *GVT* start message is the real-time $START_i$. Node i then forwards the *GVT* start message to node $(i+1) \bmod N$.

(2) The real-time that node *0* has received its *GVT* start messages from node *N-1* is both the real time *RTM* and the real-time $STOP_0$. Node *0* sends a *GVT* lvt message containing LVT_0 to node *1*. Each other node *i* receives an *GVT* lvt messages, the receipt is the real-time $STOP_i$. Node *i* minimizes LVT_i with the estimates it has received, and sends this new estimate to the node *(i+1) mod N*. When node *0* has received its last *GVT* lvt message, then node *0* has a new value for *GVT*.

(3) When node *0* has received its *GVT* lvt message, then node *0* has a new value for *GVT*.The new value of GVT is send around the ring. Node *i* sends a *GVT* update message to node *(i+1) mod N*.

This algorithm combines phases (1) and (2) of the old *GVT* algorithm into phase (1) above, and it combines phases (3) and (4) of the old *GVT* algorithm into phase (2) above. This algorithm has a run time of *O(1)* on each node and *O(N)* run time overall, it sends *3N* messages.

The token passing *GVT* algorithms are based on running all three phases of the ring topology algorithm together. That is the message (the token) being forwarded from *i* to *(i+1) mod N* contains the new *GVT* update from the first run, the GVT lvt estimate from the second run and is the GVT start message for a third run. (This algorithm is essentially the one described by (Preiss 1989).)

The New GVT Algorithm

The new *GVT* algorithm is similar to the old algorithm but it requires a "*Message Routing Graph*" (see below) and the algorithm works in just three phases:

(1) Node *0* starts the *GVT* calculation and sends a *GVT* start message to at most two other nodes. Each other node receives at most two *GVT* start messages, after having received all of its *GVT* start messages, all nodes but the last sends a *GVT* start message to at most two other nodes. Again, on each node the receipt of a *GVT* start message is the real-time $START_i$. These messages travel the arcs in the message routing graph in the forward direction.

(2) The real-time that the last node *N-1* has received all of its *GVT* start messages is the real-time $STOP_{N-1}$, and the real time *RTM*. Node *N-1* sends a *GVT* lvt

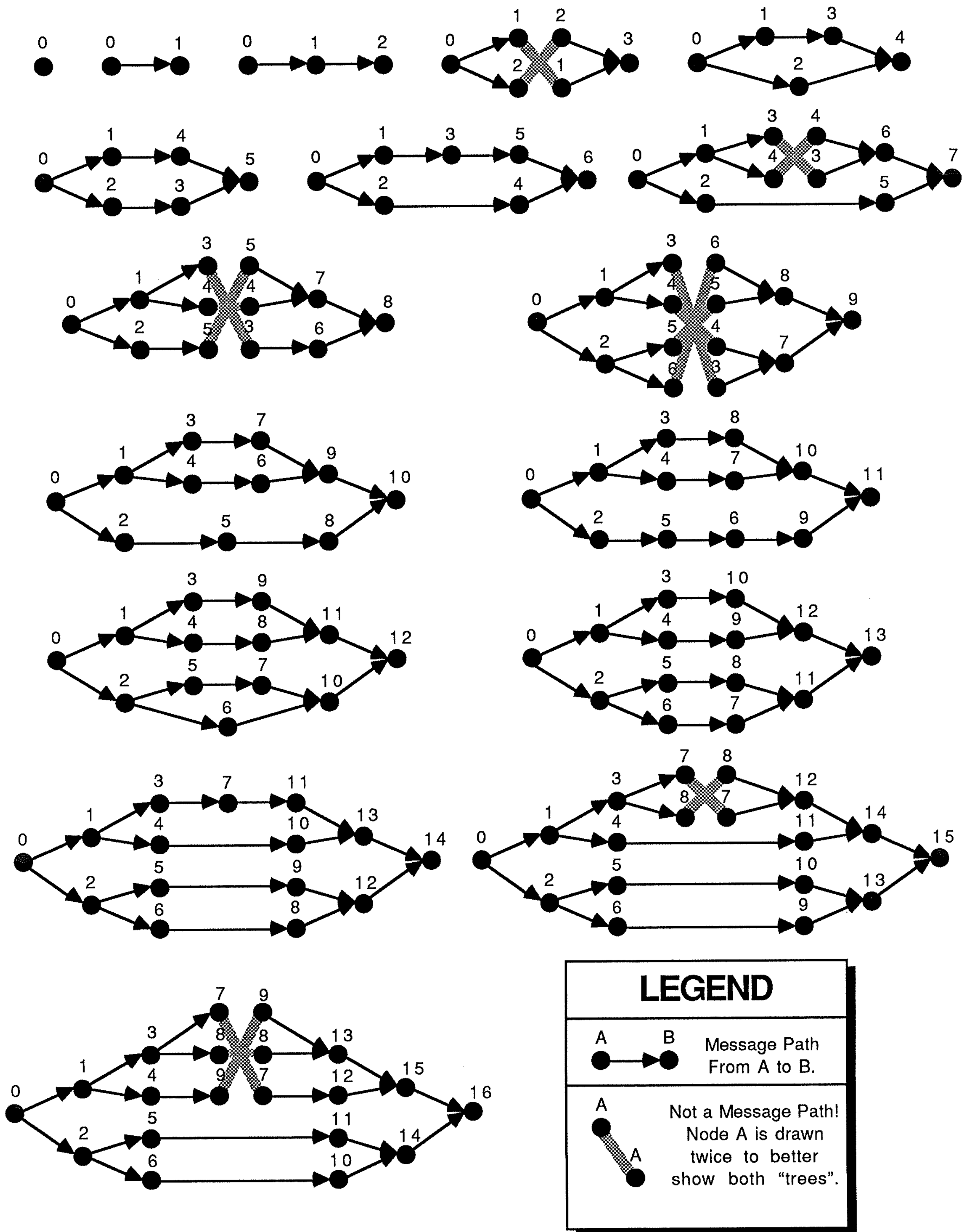


Figure 1. Message Routing Graphs for 17 and fewer nodes.

message containing LVT_{N-1} to at most two other nodes. Each other node i receives at most two GVT lvt messages, the receipt is the real-time $STOP_i$. Node i minimizes LVT_i with the estimates it has received, and sends this new estimate to at most two other nodes. When node 0 has received its last GVT lvt message, then node 0 has a new value for GVT. These messages travel the arcs in the message routing graph in the backward direction.

(3) The binary-tree-like forwarding (see above) is used (instead of a broadcast) to pass a GVT update message to all the nodes. This message contains the new estimate of GVT.

Like the ring topology algorithm, this algorithm combines phases (1) and (2) of the old GVT algorithm into phase (1) above, and it combines phases (3) and (4) of the old GVT algorithm into phase (2) above. This algorithm runs in $O(1)$ time on each node and run time of $O(\log N)$ overall. It sends less than $4N$ messages. There is $O(\log N)$ one time configuration cost on each node to construct the part of "Message Routing Graph" local to that node.

The Message Routing Graph

The message routing graph is different for different values of N. To aid the reader we have included both graphical representations of these graphs for values of N less than 18 (Figure 1) and the code each node executes to configure itself (Figure 2). The message routing graph is basically two binary trees and some connecting arcs. There are three cases to consider in constructing the message graph. We consider the easiest case first. Suppose $N = 2M$ so that N is even, then the basic construction goes as follows. Nodes 0 through $M-1$ form a binary tree with 0 at the root and the arcs pointed away from the root. Nodes M through $N-1$ form a binary tree with $N-1$ at the root and the arcs pointed toward the root. Finally there is an arc from node i on the first tree to node $N-1-i$ on the second tree if node i has no children. (See for example $N = 14, 12, 6$ in Figure 1.) If N is odd, then the middle node is considered part of both trees. (See for example $N = 15, 13, 11, 7, 5$ in Figure 1.) Finally if the number of nodes on the "bottom row" of each tree is less than half the number which could fit on the bottom row, then both trees share all the bottom row nodes. Note that in this case the "bottom row" appears twice in the network in Figure 1. This repeating of nodes makes this

```
int numFrom, numTo, from[2], to[2],
int Out0, Out1, numArrive;

/* my node number */
int myNum;

/* total number of nodes */
int numNodes;

FUNCTION gvtcfg()
{
    int Mid0, Mid1, myInv, OutFrom;
    int In0, In1, InTo, pen0, pen1;

    Mid0 = ( numNodes - 1 )/2;
    Mid1 = ( numNodes & 0x01 )?
        Mid0 : Mid0 + 1;

/* matching node on other tree */
    myInv = numNodes - myNum - 1;

/* for binary tree with root 0 */
    /* left child number */
    Out0 = 2 * myNum + 1;
    /* right child number */
    Out1 = Out0 + 1;
    /* parent number */
    OutFrom = ( myNum - 1 )/2;

/* for binary tree with root
    numNodes - 1 */
    /* left child number */
    In0 = 2 * myNum - numNodes;
    /* right child number */
    In1 = In0 - 1;
    /* parent number */
    InTo = (numNodes + myNum + 1)/2;

    for ( pen0 = 1; pen0 < Mid0 + 1;
        pen0 = 2 * pen0 + 1 )
    {
/* this is the only O(log N) part,
everything else is O(1) */
      ;
    }
    pen0 = pen0/2 - 1;
    pen1 = numNodes - pen0 -1;

    if ( pen1 <= 2 * pen0 + 3 )
/* the two trees can share the
bottom row */
    {
      Mid0 = pen1 - 1;
      Mid1 = pen0 + 1;
    }

    if ( myNum == 0 )
    {
      numFrom = 0;
    }
    else if ( myNum <= Mid0 )
    {
      numFrom = 1;
      from[0] = OutFrom;
    }
    else if ( Mid1 <= In1 )
    {
      numFrom = 2;
      from[0] = In0;
      from[1] = In1;
    }
    else if ( Mid1 == In0 )
    {
      numFrom = 1;
      from[0] = In0;
    }
    else
    {
      numFrom = 1;
      from[0] = myInv;
    }

    if ( myNum == numNodes - 1 )
    {
      numTo = 0;
    }
    else if ( myNum >= Mid1 )
    {
      numTo = 1;
      to[0] = InTo;
    }
    else if ( Mid0 >= Out1 )
    {
      numTo = 2;
      to[0] = Out0;
      to[1] = Out1;
    }
    else if ( Mid0 == Out0 )
    {
      numTo = 1;
      to[0] = Out0;
    }
    else
    {
      numTo = 1;
      to[0] = myInv;
    }

}
```

Figure 2. The configuration code to make the local part of the message routing graph.

case look more like the others. (See for example *N = 17, 16, 10, 9, 8* in Figure 1.)

The gvtcfg routine (Figure 2) has run time *O(log N)*, however only computing the variable *pen0* requires takes this long. Using *log* functions this routine can run in *O(1)* time. The local information needed from the message routing graph is stored in the variables *numFrom*, *numTo*, *from[2]* and *to[2]* and a variable *numArrive* is used to count the number of received messages. (The variables *Out0* and *Out1* are used in the update routine.) We note for later use that we always have *numTo* plus *numFrom* is bounded by three. The message routing graph uses all the arcs once in each direction and the degree of each node is no more than three and some are less than three, the number of *GVT* start messages plus the number of *GVT* lvt messages is stricly less than *3N*. Also there are *N-1 GVT* update messages. Thus the new *GVT* algorithm uses less than *4N* messages. Because of the binary trees in the message routing graph, the longest path from node *0* to node *N-1* is *O(log N)*.

GVT Calculation Time and Normal Performance

Our performance numbers are from running a preliminary version of TWOS 2.1 on a Butterfly Plus, running the Chrysalis operating system underneath the Time Warp system.

In both the old and new *GVT* algorithms the natural measure of speed is the time between when node *0* starts a *GVT* calculation by sending the first message, until the time that node *0* knows the new estimate of *GVT*. We will call this time period the *GVT* calculation time. In even well-behaved simulations this *GVT* calculation time can vary widely (with TWOS 2.1 on the Butterfly). This is due to the lack of interrupts, so that an executing objects are not pre-empted for message arrivals. Messages must wait for running objects to finish to be received. Thus most of are graphs are of average *GVT* calculation times. However, the first and last *GVT* calculations of a simulation are done when nothing else is executing. This minimum *GVT* calculation time is graphed in Figure 3a. This graph clearly shows the linear run time of the old algorithm and the logarithm run time of the new algorithm.

To represent normal behavior we used the twq simulation. (Twq was designed and wriiten by Mike Di Loreto.) The twq

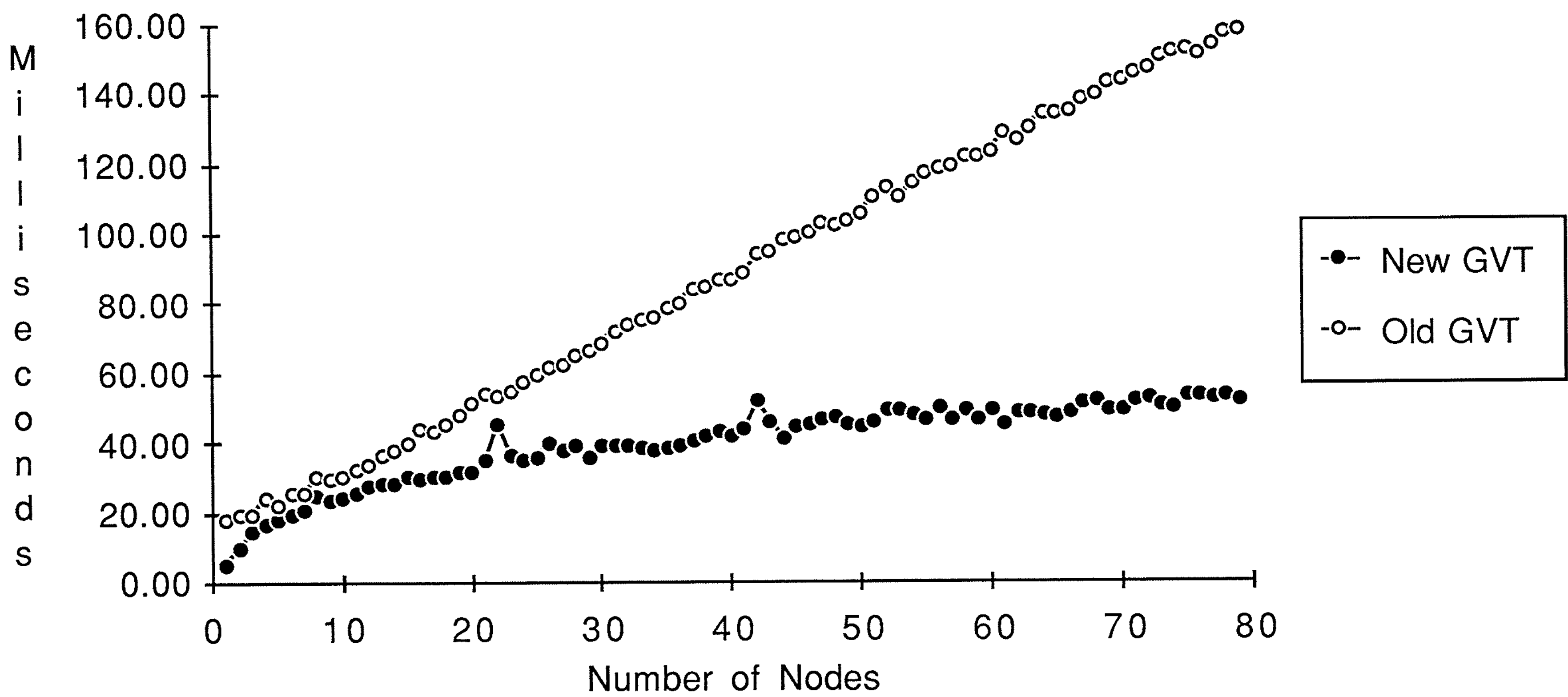


Figure 3a Minimum GVT Calculation Times vs Number of Nodes

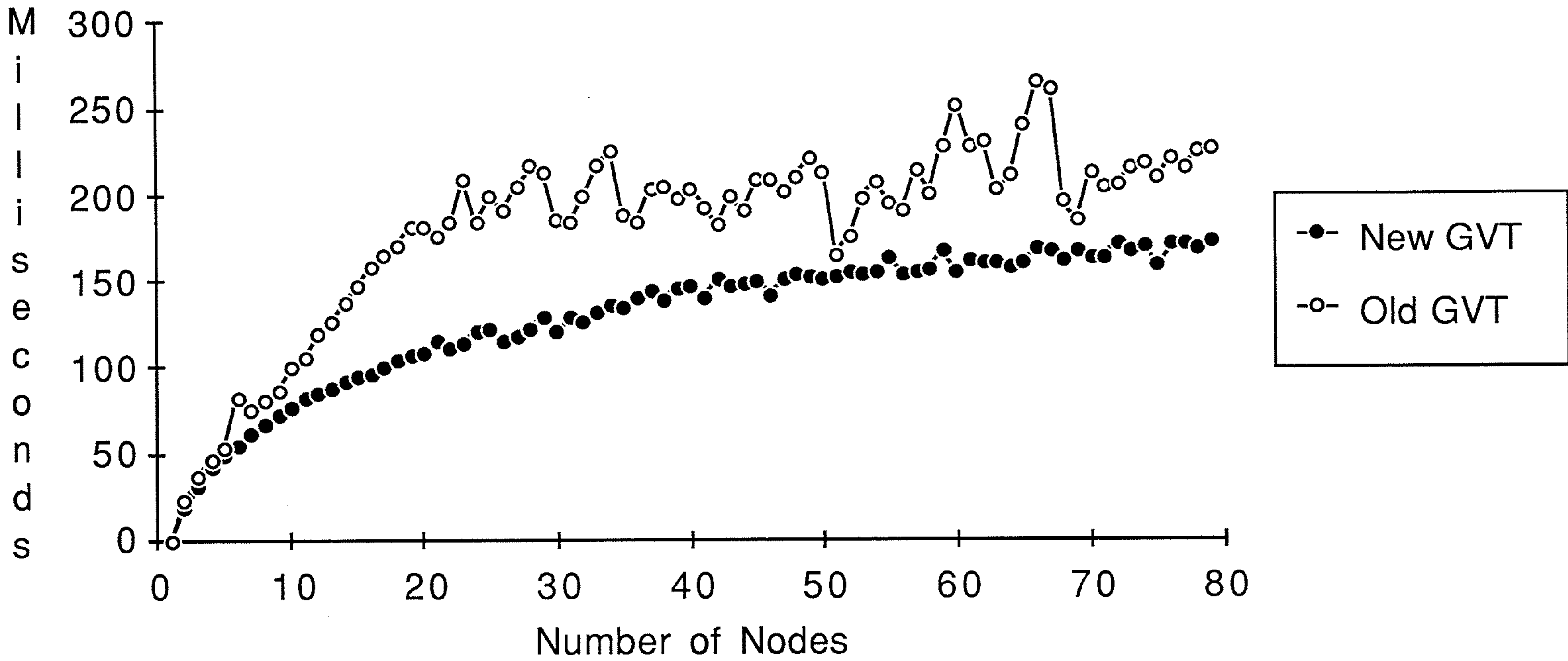


Figure 3b Average GVT Calculation times vs Number of Nodes

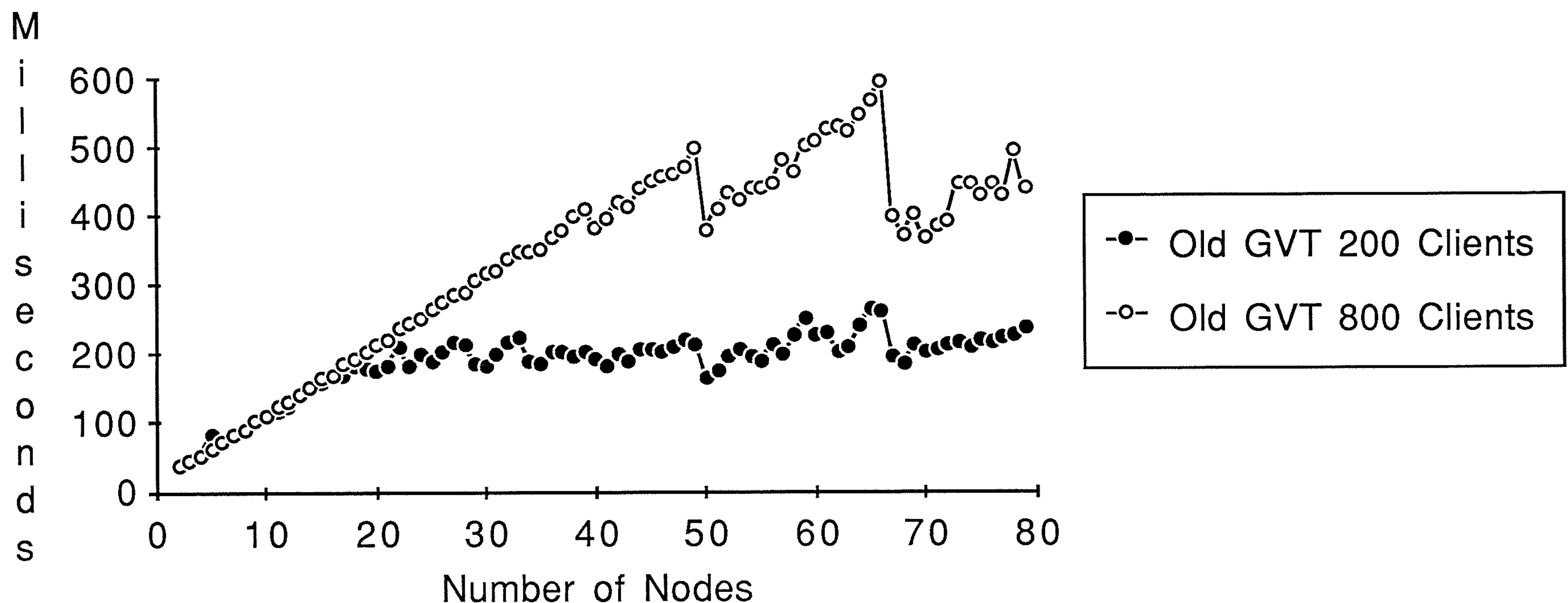


Figure 3c Increased Message Traffic and Old GVT Calculation Times

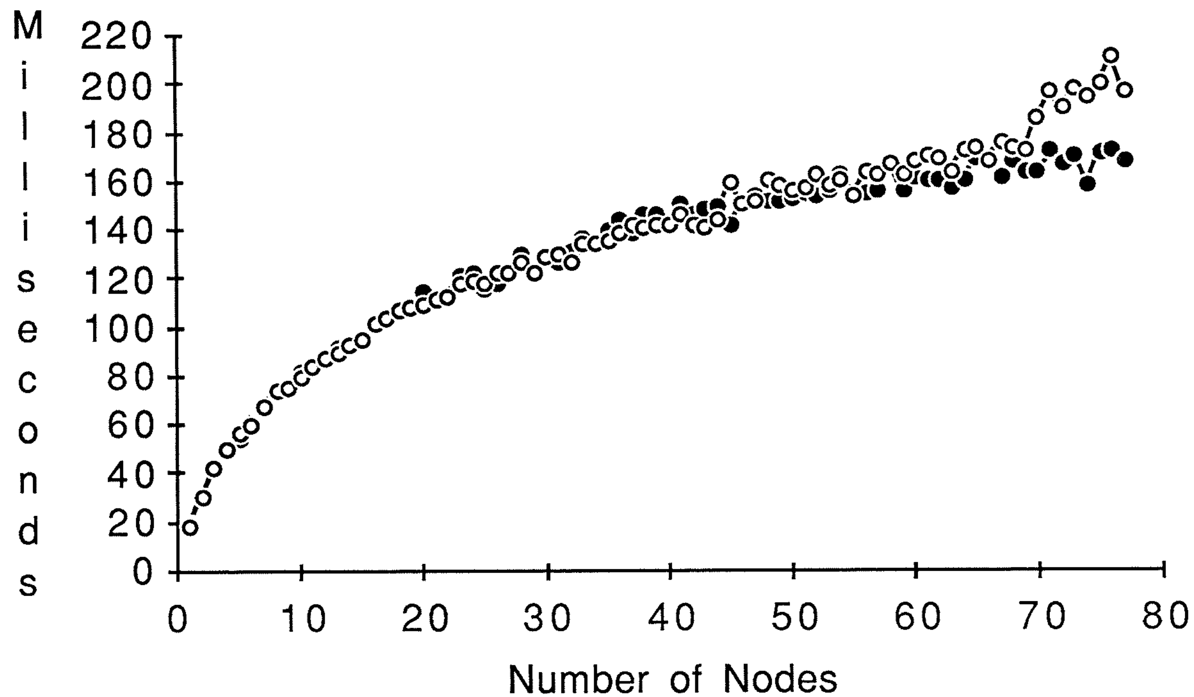


Figure 3d Increased Message Traffic and New GVT Calculation Times

simulation is a FIFO queueing model where the clients are messages and the servers are Time Warp objects. Figure 3 shows *GVT* calculation times for this simulation as *N* changes from *1* to *79*. Figure 3b graphs the old vs new *GVT* average calculation time when there are *200* clients. Several features of Figure 3b are worth observing. First, the initial slope of the linear growth of old *GVT* calculation time is much steeper than that of Figure 3a. Second, the average old *GVT* calculation time is near the minimum old *GVT* calculation time for $N = 79$. Third, the new *GVT* curve still has its logarithm shape but the average calculation time is about three times longer than the minimum calculation time.

The most interesting feature of Figure 3b occurs at about *20* nodes. The old GVT calculation times stops growing linearly and flattens out into a level graph with what seems to be a random variation. First we consider the random pattern. One of the advantages of twq is that it almost evenly spreads objects to nodes at run time. All our runs had *200* servers with server *i* placed on node *i mod N*. This means node *0* sometimes was overloaded with objects relative to most other nodes. When $N=66$, node *0* had *4* objects but almost all others had *3* objects, whereas when $N=67$ node *0* and almost all nodes had *3* objects. Since node *0* is also overused in the old *GVT* algorithm, this explains the drop in old average *GVT* calculation time as *N* changes from *66* to *67*. Similar reasons apply to the changes from *49* to *50*, *39* to *40* and *33* to *34*.

The reason Figure 3b flattens out is due to the increased idle time on each node. Twq was re-run with *800* clients and the average *GVT* calculation times for the old algorithm changed (Figure 3c). Again the initial steep linear slope appears, but this time it dominates the graph to about *40* or *50* nodes. The *66, 67* and *49, 50* drops are still clear, but so is a linear increase between *50* and *66*. Figure 3d shows the increase from *200* to *800* clients has little effect on the average new *GVT* calculation time.

Thus Figure 3 shows that the new *GVT* algorithm is more robust. However the twq simulation is well-behaved, the run time of executing twq with the new *GVT* algorithm vs the run time of executing twq with the old *GVT* algorithm are the same. In well-behaved simulations, *GVT* calculations don't need to be real fast. As a guess, *GVT* calculation times of a couple or three seconds shouldn't change the run time of twq for $N < 80$.

Performance Under Stress

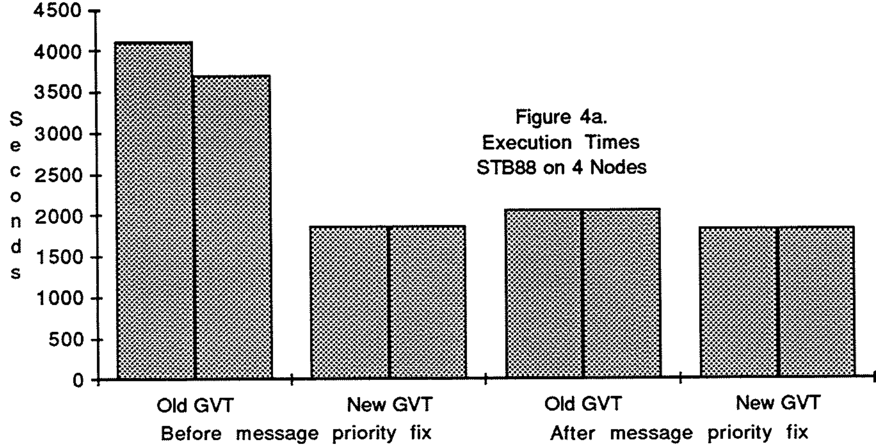
The STB88 benchmark (Wieland et al.1989) would barely run under TWOS 2.0 on *4* Butterfly nodes. Part of the reason for this poor performance would turn out to be a lack of tuning of TWOS to the Butterfly. It was during this period that the new *GVT* algorithm was added to TWOS. In the face of these tuning problems is where the new *GVT* algorithm really shined over the old algorithm. Figure 4 shows the story. The message priority fix has to do with the treatment of system messages. TWOS 2.0 and 2.1 use a Chrysalis FIFO queue to help with message sending. When a message is sent, an integer which gives the address of the message is placed in this queue. In 2.1 the FIFO order was fixed so that system messages like *GVT* messages got jammed in the front of the queue, instead of FIFO order. (TWOS 2.1 made a number of other improvements dealing with messages delays which eventually made the run times of two *GVT* methods about the same.)

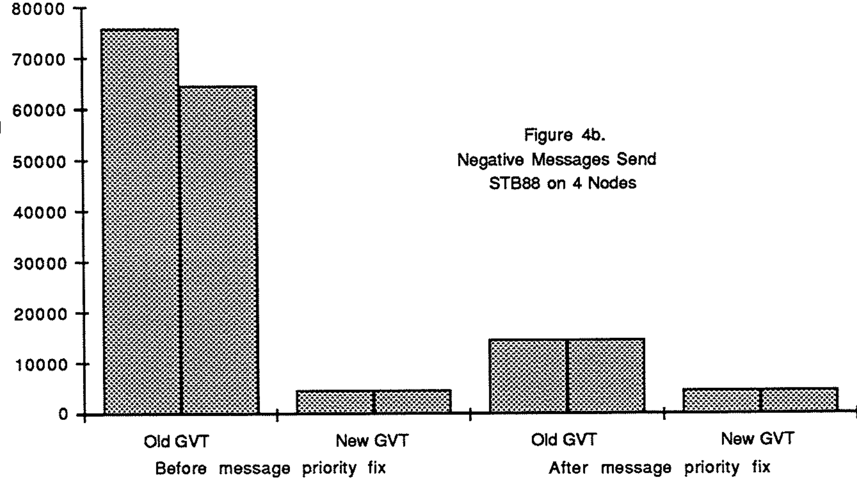
Figure 4a shows the difference in run time that the responsiveness of the new *GVT* algorithm can make. The run times under old *GVT* varied and were roughly twice that under the new *GVT*. Figure 4d shows the average *GVT* calculation times. Figure 4d seems to imply that the average *GVT* calculation time needs to be less than a couple of seconds. (Both the old and new algorithms had some GVT calculations which were over seven seconds.) Both Figure 4b and 4c show that the old *GVT* runs are doing lots of extra work. Even after the message priority was fixed, the old *GVT* algorithm send much more negative and reverse messages than the new *GVT* algorithm. However both the *GVT* calculation times and the run times are close after the message priority fix.

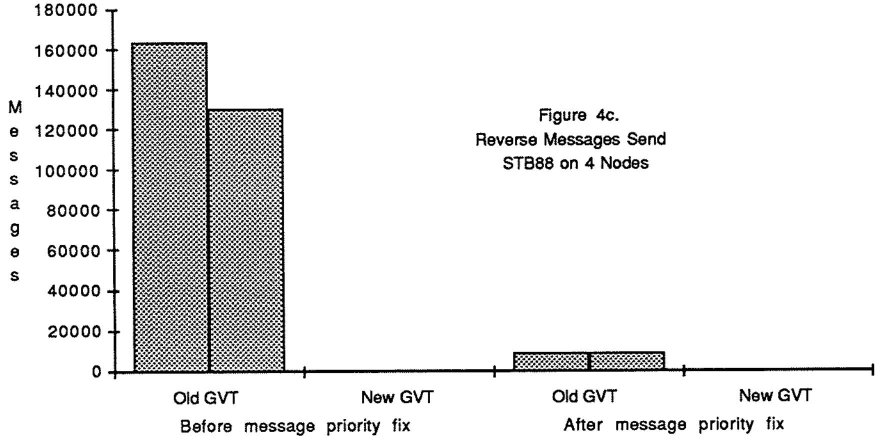
Not only did the new *GVT* really out shined the old *GVT* in this bad case. Careful inspection of Figure 3c, will show there is no data for *N=1* but there is data in Figure 3d for *N=1*. The old *GVT* algorithm could not run twq in the *800* client case on one node (it ran out of memory) but the new *GVT* algorithm could.

More Performance

Figure 5 contains a histogram of *GVT* calculation times, old vs new, for STB88 on *16* nodes. The run time of STB88 with the new







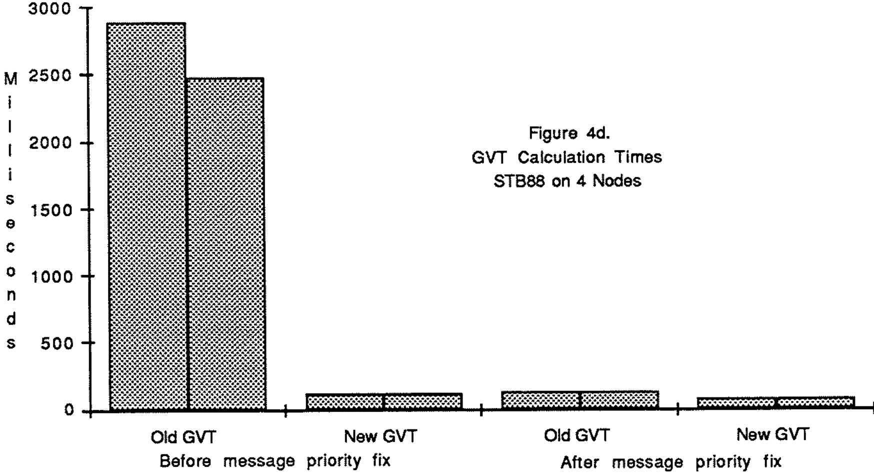


Figure 4. Performance Under Stress

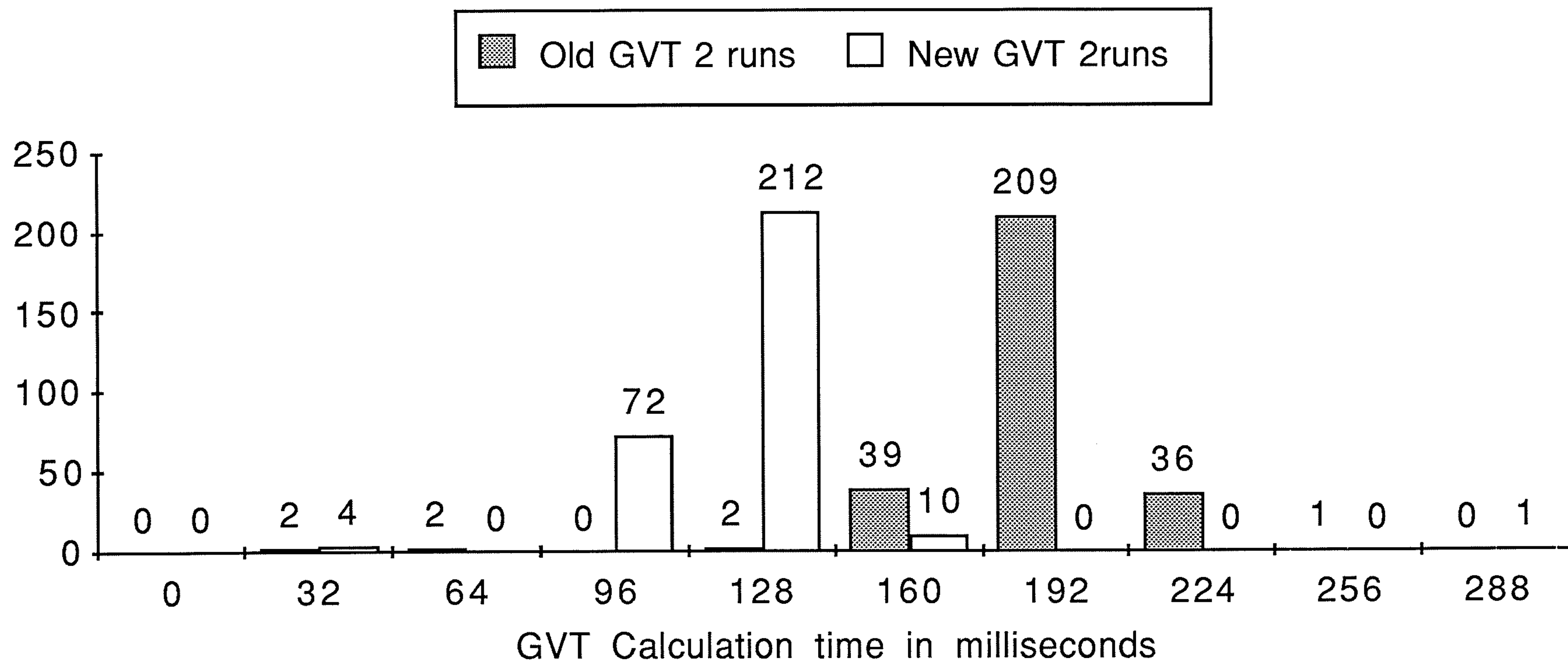


Figure 5 Normal GVT calculation times

GVT algorithm ran slightly faster than STB88 with the old *GVT* algorithm for these four runs, but not even *1%* faster. Perhaps this is a rough estimate of normal difference of GVT calculations times between the new and old GVT algorithms for modest-sized *N*. Removing sequential features from a parallel program does give one a measure of satisfaction. However, clearly the *GVT* algorithm wasn't limiting the speed of TWOS.

We close this paper with an interesting story. When the new *GVT* algorithm was added for the first time, TWOS ran a *16* node STB88 about *60* seconds faster, more than *10%* faster than with using the old *GVT* algorithm. However, the next day, when TWOS was re-linked STB88 with the old *GVT* algorithm ran *10%* faster than with using the new *GVT* algorithm. Eventually, the difference was traced to a change in an SCCS identifier string. An existing change was SCCS-edited into the official code. The time of day field of the SCCS string became one character longer. The C compiler, gave that string two extra bytes and moved all variables after this string by two bytes. These two bytes changed the Time Warp heap memory (where all messages and states are allocated) from a number divisible by four to one which wasn't. The 68020 is much faster for memory references which line up on four byte boundaries. Sometimes initial data is too good to be true.

REFERENCES

Bellenot, S. and Di Loreto, M., 1989, "Tools for measuring the performance and diagnosing the behavior of distributed simulations using time warp." In *Distributed Simulation 1989-- Proc SCS Winter Multiconference*, Vol. 21 No. 2, 145-150.

Jefferson, D., "Virtual Time." 1985, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, 404-425.

Jefferson, D., Beckman, B., Wieland, F., Blume, L., Di Loreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S.,1987, "Distributed Simulation and The Time Warp Operating System." In *Proc 12th SIGOPS--Symposium on Operating Systems Principles*, 77-93.

Preiss, B., 1989, "The Yaddes distributed discrete event simulation specification language and execution environments." In

Distributed Simulation 1989-- Proc SCS Winter Multiconference, Vol. 21 No. 2, 139-144.

Wieland, F., Hawley, L., Feinberg, A., Di Loreto, M., Blume, L., Reiher, P., Beckman, B., Hontales, P., Bellenot, S., Jefferson, D., 1989, "Distributed combat simulation and time warp: the model and its performance." In *Distributed Simulation 1989-- Proc SCS Winter Multiconference*, Vol. 21 No. 2, 14-20.

Author's Addresses:

(Til 25 August 1989)
Mail Stop 510-202, JPL 4800 Oak Grove Dr., Pasadena, CA 91109
steve@sapphire.jpl.nasa.gov

Math Dept, Florida State Univ, Tallahassee, FL 32306
bellenot@gauss.math.fsu.edu