

Routing without Flow Control Hot-Potato Routing Simulation Analysis

Lawrence Bush
Computer Science Department
Rensselaer Polytechnic Institute
Troy, New York
December 5, 2002

Abstract

This paper presents a simulation analysis of the algorithm presented in “Routing without Flow Control,” by Busch, Herlihy and Wattenhoffer, 2001, [1]. The hot-potato routing algorithm is simulated using Rensselaer’s Optimistic Simulation System. The simulation uses a novel reverse computation approach to efficiently and optimistically parallelize the system simulation. In addition to simulating and analyzing the routing algorithm, the performance of the simulation itself is also analyzed.

1. Problem Description

Busch (et al.) [1] presents the first dynamic hot-potato routing algorithm that does not require explicit flow control. Hot-potato routing is also known as deflection routing. In hot-potato routing, the nodes or routers in the network do not have buffers to store the packets in transit. They do, however, have a delay loop to contain the packet while the routing decision is taking place. Therefore, a packet that is being routed must be sent along its destination path or deflected in an alternative undesired direction. The hot-potato algorithms are useful for optical switching networks. This algorithm can be used to route packets through a buffer-less optical network.

1.1 Network Description

The performance of the algorithm presented in [1] is analyzed in a buffer-less, synchronous, N by N rectangular mesh network.

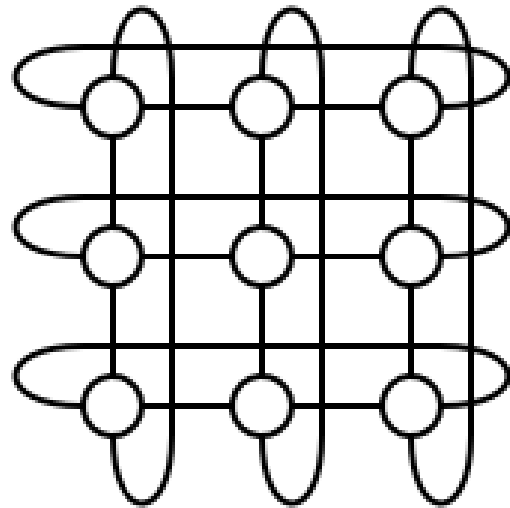


Figure 1: 3 by 3 Torus Network

1.1.1 Topology

Each node in an N by N rectangular mesh network is connected to its four neighbors via a bi-directional link. If the left edge of the mesh network is connected to the right edge of the mesh and the top edge of the mesh is connected to the bottom edge of the mesh, the result is a torus shaped mesh (See Figure 2).

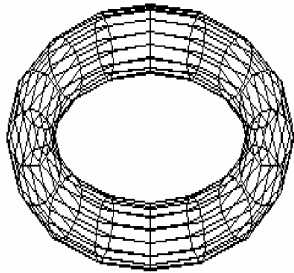


Figure 2: 3 - Spatial Representation of an N by N Torus Network

The network topology used in the theoretical algorithm analysis is the more straightforward mesh topology because it makes the problem more tractable. The theoretical analysis could be easily extended to the torus topology. The simulation uses the torus network because it is a more practical implementation of essentially the same topology. It is more practical because the maximum distance between any two nodes is $N - 1$ rather than $2N - 1$ for the mesh topology.

1.1.2 Characteristics

The network is synchronous. As such, time is measured in discrete time steps. A node traverses a link in one time step. The links are bidirectional.

The network is buffer-less. Buffering allows a network to store packets until they can be forwarded. A buffering network makes it difficult to establish a bound for the delay that a packet may encounter in its route. Also, buffering is not practical for certain types of networks such as optical networks. In an optical network, packets cannot be buffered without converting them to their electronic form. It is desirable to maintain packets in their optical form for speed.

1.1.3 Model of Optical Switching Network

In optical label switching, a packet's optical label contains routing and control information such as the source, destination and priority of the packet. The size of the packet is not considered in this particular model. In the hot-potato model, the packet label contains only the destination and priority.

The packet is delayed by an optical fiber loop to allow time for the processing of the packet label and the packet switching.

1.2 Algorithm

Busch (et al.) [1] details and presents proofs regarding a hot-potato routing algorithm under dynamic packet injection.

1.2.1 Algorithm Analysis

Dynamic versus static analysis of a routing algorithm differ by the definition of the workload. In a static analysis, all packets are assumed to be injected into the network simultaneously when the analysis is initialized. In a dynamic analysis, packets are injected continuously at rates that can vary.

Under dynamic analysis, the algorithm presented in [1] is shown to guarantee expected $O(n)$ delivery and injection times.

1.2.2 Algorithm Characteristics

Flow Control is a mechanism in which packet sources adjust their load so that they do not overload a network. They do this by notifying or monitoring the network. Either strategy requires explicit communication with the overall network.

Hot-potato routing avoids flow control by using a simple, locally optimal (greedy) routing strategy. The simple algorithm does not need to communicate with a central flow control mechanism. The routing algorithm can be implemented as a series of AND / NOT operations to minimize switching overhead thus allowing rapid switching implementation in an optical network. The injection intervals and delivery times are bounded. This allows the network to simultaneously accommodate high-speed injection rates and lower speed users. It also allows a much higher utilization of network links where flow controlled routing results in significant under-utilization of network links. Together these characteristics result in a more flexible and higher performance optical network.

The algorithm presented in [1] is greedy. A greedy algorithm is one in which a locally optimal solution is chosen. In the case of a routing algorithm, it chooses to route a packet to a link, which brings the packet closer to its

destination, whenever this choice is possible. As such, each packet attempts to take a greedy path to its respective destination.

A similar algorithm is presented in Das (et al.) [2]. However, that algorithm analyzed the performance in a static system.

1.2.3 Algorithm Rules

The algorithm rules presented in this section are defined in terms of good-links and bad-links. A good-link is any link that brings the packet closer to its destination. A bad-link is any link that does not bring the packet closer to its destination.

The basic logic behind hot-potato routing is that at each time step a packet attempts to follow a good-link. The result of this locally optimal decision is a greed-path. A variation on the greed-path is the home-run path which is also known as a one-bend path. A home-run path is defined as a path that only has one turn or bend in it and follows the row first followed by the column. For example, suppose a packet is following its home-run path. In the first part of its home-run path the packet remains in the row it is in, but moves in the direction of its destination column. The second part of the home-run path occurs after it reaches its destination column. Once it reaches its destination column, the packet follows the column links until it reaches its destination node.

There are four priority states: Sleeping, Active, Excited and Running. Sleeping is the lowest priority. Running is the highest priority.

The higher priority packets are given routing precedence over the lower priority packets. Priority ties are broken arbitrarily.

The actual routing decision is a bit more complex and the routing decision differs for packets of different priority states.

In the Sleeping state, the packet is routed to any good-link. When a packet in the Sleeping state is routed, it is given a chance with the probability of $1/24n$ (where N is the dimension of the N by N torus network) to upgrade to the Active state.

In the Active state, the packet is routed to any good-link. When an active packet is deflected, it is given a chance with the probability of $1/16n$

(where N is the dimension of the N by N torus network) to upgrade to the Excited state.

In the Excited state, the packet is routed via its home-run path. If the packet can be routed via its home-run path, the packet's priority is increased to the Running state. If the packet cannot be routed via its home-run path and is subsequently deflected, the packet returns to the Active state. Note that a packet remains in the Excited state for only, at most, one time step.

In the Running state, the packet is routed via its home-run path. Due to the dynamics of the routing algorithm, a running packet cannot be deflected from its path except while it is turning (from the first to the second part of its home-run path). If a running packet is deflected (by another running packet) while turning, it returns to the lower priority Active state.

2. Related Work

Experimental analysis of Hot-potato Routing Algorithms in a 2-Dimensional Torus Network is presented in [5]. This paper compares four different algorithms using tori of several sizes and 100 inputs. The implementation and testing strategy is significantly different than the approach taken in this paper, however, the objective is the same.

The implementation and testing strategy used in the experiments presented in this paper is similar to the approach taken in [4]. In [4] a parallel simulation approach is used to simulate a Personal Communication Service (PCS network using Rensselaer's Optimistic Simulation System (ROSS). This approach extends the work performed in [6] on the Georgia Tech Time Warp System to use the reverse computation method implemented by ROSS.

3. Solution Description

The hot-potato routing algorithm was simulated on ROSS. ROSS is a parallel discrete-event simulator, specifically, a C-based Time Warp system. The simulation was run on a quad-processor Personal Computer (PC) server. This optimistic simulation system uses Fujimoto's Global Virtual Time (GVT) algorithm for process synchronization, reverse computation to

reduce rollback overhead and Kernel Processes (KPs) to minimize total rollbacks.

3.1 Model Representation

This section explains how the hot-potato algorithm and the associated network is represented in ROSS.

3.1.1 Logical Processes

The primary component in a ROSS simulation application is the Logical Process (LP). A simulation is comprised of a collection of LPs, each simulating a separate component of the system. In the hot-potato routing simulation, each LP represents a router. The collection of LPs represent a network, specifically, a buffer-less optical network. In ROSS LPs are generated in the startup function when the simulation is initiated.

3.1.2 Messages

The LPs communicate with each other within the simulation via messages. Each message represents an event in the system. These messages are generated by the LPs when a new event is needed. The messages keep the system going, as such, ROSS is an event driven simulator. ROSS runs on a shared memory parallel processing PC server. Therefore, the messages are not “sent” in the way they would be on a distributed system. Sending a message from the source LP to the destination LP merely involves assigning ownership of the message’s memory location from the source LP to the destination LP. This shared memory architecture allows ROSS to use Fujimoto’s GVT algorithm rather than a less efficient distributed GVT algorithm such as Mattern’s [7].

The messages in this dynamic hot-potato routing simulation represent packets to be routed. A router will receive a packet, decide what to do with it and generate a new message (representing a packet) destined for another LP if the current router is not the packet’s destination.

Each packet in the dynamic hot-potato routing algorithm contains a header or label indicating its destination and priority. The data structure in the ROSS application that represents the message is the message struct. The packet header is represented in the simulation by three variables in the message struct.

```
typedef struct {
    ...
    enum priorities priority;
    int destination_row;
    int destination_column;
    ...
} Msg_Data;
```

3.1.3 Network Mapping

The routers in the dynamic hot-potato routing algorithm are configured into an N by N torus network. This topology is emulated in the simulation by restricting where a router can route a packet. Specifically, the routers are allowed to route packets to four neighboring routers. This is implemented by a calculation within each LP. In ROSS each LP is given a number. For example, if the network consists of a 32*32 torus network, ROSS generates 1024 LPs numbered from 0 to 1023. Row 1 contains LP 0 – 31, Row 2 contains LP 32 – 63 etc. These LPs form an implicit wrap-around grid of 32 rows each with 32 LPs per row. Each LP can send a packet in 4 directions (North, South, East and West). If an LP chooses to send a packet East, the LP_x sends the packet to LP_{x+1}. The network wraps around. Therefore, if an LP resides on the East most side of the network, it must send the packet to the West most LP in the same row. To do this, the following calculation is performed:

```
NewLp =
    ((lp->id / NumLpsRT) * NumLpsRT)
    + ((lp->id + 1) % NumLpsRT);

/*
lp->id      : The sending LP
              number.

NumLpsRT   : The number of rows
              in the network.

NewLp      : The destination LP
              number. */
```

As you can see from the above description, the network topology is not explicitly laid out by the simulation setup. It is implicitly defined by the routing restrictions of the destination calculation.

3.1.4 Routing Algorithm

The dynamic hot-potato routing algorithm is implemented within each LP or router. Each router is identical. When a message (synonymous with event or packet) is executed in a given router, the router executes the given event type denoted in the message struct. There are four event types: ARRIVE, ROUTE, HEARTBEAT and PACKET_INJECTION_APPLICATION.

The ARRIVE event simulates the arrival of a packet to a router. The main function of an ARRIVE event is to generate an appropriate message to itself (destined for the same LP) to initiate a ROUTE event. The priority level of the arriving packet determines the order in which the packet's route will be considered by the router. To facilitate this, the time stamps of the generated ROUTE events are staggered based on priority. If the packet arrives at its destination router, no new event is created. Instead, statistics regarding the event, such as its delivery time, are recorded.

The ROUTE event determines which direction the packet will be routed. It also determines if the packet's priority will be changed, as described in the algorithm description above. It then creates a new ARRIVE event at the appropriate destination router.

The HEARTBEAT event simply generates events to perform administrative overhead. In some configurations, that overhead can be taken care of by other events. In those cases, the HEARTBEAT event is not used, in order to reduce the total number of simulated events.

The PACKET_INJECTION_APPLICATION event simulates the injection of new packets into the system. The startup program determines the number of LPs that are packet generators based on the application input parameters. The number of packet generators can vary anywhere from zero to N by N LPs. In our tests, N LPs are packet generators. This simulates a scenario where the network is kept relatively full, yet there are still specific sources.

```
void
Router_EventHandler(Router_State
*SV, tw_bf *CV, Msg_Data *M,
tw_lp *lp) {

enum directions NewDir;
enum bool deflected;
NewDir = NO_DIRECTION;

/* reset bit fields CV->* to 0
for this event */
*(int *)CV = (int)0;

deflected = false;

switch(M->event_type) {

case ARRIVE:
Router_ARRIVE_EventHandler( SV,
CV, M, lp );
break;

case ROUTE:
Router_ROUTE_EventHandler( SV,
CV, M, lp );
break;

case HEARTBEAT:
Router_HEARTBEAT_EventHandler(
SV, CV, M, lp );
break;

case
PACKET_INJECTION_APPLICATION:
Router_PACKET_INJECTION_APPLICATION_EventHandler( SV, CV, M, lp
);
break;

}

}
```

3.1.5 Statistics

This simulation collects several statistics. In particular, we want to know what the expected packet delivery time is with respect to the network size. Therefore, each router keeps track of the total number of packets that were delivered to it, how long the packets were in transit and how far they came.

We also want to know how long a packet waits to be injected into the network (expected and

worst case time). Therefore, each router keeps track of the amount of time that each injected packet waited to be injected, the total number of packets that were injected into the system and the longest time that any packet had to wait to be injected.

All of the above statistics are aggregated from each router to determine system wide totals. These statistics are aggregated by a statistics collection function. The statistics collection function is an adaptable ROSS construct that executes once for each LP (router) when the simulation finishes. The application programmer implements the statistics collection function content in much the same way that a C++ visitor functor is implemented.

3.2 ROSS Specific Issues

There are certain aspects of the simulation application that are specific to ROSS (or inherited from its predecessor, Georgia Tech Time Warp [2]). These are not simply syntactic issues but conceptual in nature.

3.2.1 Reverse Computation

ROSS is an optimistic parallel simulator. Therefore, ROSS divides up the simulation tasks among processors (PEs), which then execute their assigned tasks optimistically. Basically, each processor operates semi-autonomously by assuming that the information that it currently has is correct and complete. ROSS performs inter-processor communication via messages. Therefore, each PE operates in this manner until an incoming message informs it differently. A PE can get ahead of the other processors. At some point, it may receive a message with a time stamp (ts) that is in the past relative to that PE's local simulation time. At that point, the simulation on that PE must rollback to the time-stamp of the incoming message. ROSS uses a technique called Reverse Computation to do this. This technique is different than the state-saving technique used in the Georgia Tech Time Warp system. It rolls back the simulation by computing the events in reverse, which re-instates the respective LP to its previous state.

For example, the following function reverse computes a ROUTE event:

```
void
RC_Router_ROUTE_EventHandler(
    Router_State *SV,
    tw_bf *CV,
    Msg_Data *M,
    tw_lp *lp)
{
    if( CV->c1 ) {
        tw_rand_reverse_unif(lp->id);
    }
    if( CV->c2 ) {
        tw_rand_reverse_unif(lp->id);
    }
    SV->link[M->Saved_NewDir]=
        M->Saved_NewDir_status;
}
}
```

3.2.2 Simultaneous Events and Randomization

Due to the nature of this simulation, simultaneous events are likely. The network is synchronous, as such, routing events occur at discrete time steps (one time step = 100). If two packets of the same priority level are routed from the same LP at the same time-step, the simulator executes them in an arbitrary order. The order is dependent on the pace of the simulation. The simulation is parallel; therefore, events simulated on one processor may get ahead of events simulated on a different processor. Consequently, the order that simultaneous events are simulated may differ from one simulation run to the next. As a result, the simulation is not deterministic. In other words, the results of the simulation may differ from one run to the next. The results typically will be approximately the same. However, it is desirable to show that a simulation is repeatable.

In order to make the simulation deterministic and therefore repeatable, paradoxically, I had to use a random number generator.

First I identified any case in the simulation where the execution order of simultaneous events would affect the outcome of the simulation. The only case where this happens is when a router routes two (or more) packets that have the same priority level, which would have been routed to the same link. In that case, the

packet that arrives at the router (in the simulation) first will be routed first and will get the desired link. Each such occurrence will usually affect the network statistics.

In order to avoid this situation, I introduced a randomized delay in the arrival of each packet. This delay is introduced when the packet is injected and is carried throughout the simulation. In the non-random simulation a packet would arrive at time 600 (any multiple of 100), but in the randomized simulation the packet now arrives at time 600 + random number from 0 to .5. The code works as follows:

```
/* inject PACKET */
ts = 1+ ( (double)
tw_rand_integer( lp->id, 1,
500000000)/1000000000);
CurEvent =
tw_event_new(lp, ts, lp);
```

The reasons that this randomization makes the simulation deterministic are:

1. the randomization eliminates simultaneous events,
2. the random number generator is reversible [3, Section 3.2],
3. and the random number generator is deterministic.

Each router uses the same random number generator with a different seed value. The random number generator is reversed by calling:

```
tw_rand_reverse_unif(lp->id);
```

3.2.6 LP/KP Mapping

ROSS uses KPs which are groupings of LPs within a KP. KPs help to minimize rollbacks to improve performance.

A key factor which impacts the effectiveness of KPs in improving performance is the LP/KP mapping. In the hot-potato simulation, this mapping was constructed so as to minimize inter-PE communication (IPC) and inter-KP (IKC) communication. If the LPs within a given

KP are adjacent to each other, when a packet is routed to an adjacent LP that LP is likely to be in the same KP. However, if the LPs within a given KP are randomly assigned, then when a packet is routed to an adjacent LP that LP is likely to be in another KP and quite possibly another PE. Therefore, it is beneficial to assign adjacent LPs to the same KP and adjacent KPs to the same PE in order to minimize IPC and IKC. Therefore, the hot-potato simulation uses an LP/KP/PE mapping which divides up the network into rectangular areas of LPs and rectangular areas of KPs. The LPs in a given area will be assigned to one KP and the LPs in a given area will be assigned to one PE. This configuration minimizes the size of the circumference of the KP – KP boundaries and PE – PE boundaries, which consequently minimizes IPC and IKP.

The number of KPs also affects the number of rollbacks. In general, the more KPs you have, the fewer rollbacks you have. This relationship was analyzed in the hot-potato simulation. A detailed discussion of this is presented in Section 4.2.3 of this paper.

3.3 Workload Characteristics

This section discusses the input parameters of the simulation and their affect on the workload and performance of the system.

3.3.1 Configurations/Parameterization

The simulation is parameterized so that it can be run in different configurations.

The first input parameter *N* indicates the size of the network to be simulated. *N* must be a multiple of 8 so that it comports with the number of KPs (discussed below) used in the LP mapping.

The second input parameter *number_of_processors* indicates the number of processors in the parallel processing computer. ROSS needs this information so that it can properly and efficiently map LPs to processors.

The third input parameter *SIMULATION_DURATION* indicates how long, in simulation time, the simulation will run.

The fourth parameter `probability_i` indicates how many routers should produce packets. The user can specify that anywhere from 0 to 100% of the routers will be sources and inject packets into the network. The user input parameter is probabilistic such that if the user inputs $X\%$ then the probability that a given router will be an injector is $X/100$ (i.e. $100/100 = \text{all}$, $0/100 = \text{none}$).

Note that the network is initialized to full (four packets per router). Therefore, if the user inputs 0, then the system is run on a one-shot or static basis.

The fifth parameter `absorb_sleeping_packet` indicates if a router should absorb a packet that is in the sleeping state. Under normal operations, a network running a hot-potato routing algorithm would absorb any packet that has reached its destination. However, the model created and described in [1] uses certain assumptions and rule constructions that make the analysis more tractable. Therefore, the simulation may be run in either mode. One would indicate the algorithm's practical performance; the other only serves as verification of the algorithm proof.

3.3.2 Parallelism

The parameters discussed in section 3.3.1 (specifically N and `probability_i`) affect the workload of the simulation.

The size of the system defined by N significantly affects the speed of the simulation. Although a larger network with more routers generally does increase parallelization, the simulation presented here experienced lower event rates as N increased (See Figure 5). Additionally, the absolute time that the simulation takes to advance one time step is much larger (generally $O(n^2)$).

The input parameter `probability_i` determines how many routers have an associated injection application. Each injection application attempts to inject a packet at every time step. Therefore, if N routers are injecting packets, then $O(N/N)$ or $O(1)$ packets are injected at each time step.

As stated above, a larger network yields higher parallelism. This is because the work load on

each PE will usually be more balanced. Since the workload on each LP is somewhat random, the more LPs per PE that there are averages the randomness over more LPs creating a more balanced load.

More LPs (determined by N) per PE also creates less inter-process communication because adjacent LPs are simulated on the same PE. LPs only send messages to adjacent LPs (and to themselves). The LPs are grouped into blocks designed to minimize the connections between LPs on other PEs. Inter-processor communication only occurs when a packet is routed at the edge of one block (on one PE), to the edge of another block (on another PE). Rollbacks occur when messages from the past (in simulation time) are sent to a PE from another PE that is "behind" in simulation time. Therefore, if we minimize inter-processor communication, we also minimize rollbacks and increase parallelism.

I put "behind" in quotes because the simulation time is relative. The simulation time of the slowest PE is actually the real overall simulation time.

4. Solution Analysis

This section discusses the results of the hot-potato routing simulation as well as the performance of the simulation itself.

4.1 Algorithm Analysis

The hot-potato routing algorithm described in [1] guarantees an expected $O(N)$ delivery and injection time where N is the diameter of the network. The simulation was designed to test these guarantees over a variety of conditions.

Figure 3 and 4 display the results of the system simulated under four different loads and 32 different network diameters. The loads are represented as a percentage of the total number of LPs (routers) that will have an associated packet injection application. Each packet injection application injects packets at a rate of one packet per time step. The network diameters used in the simulated configurations range from 8 to 256.

Figure 3 shows the average packet delivery time with respect to the network size.

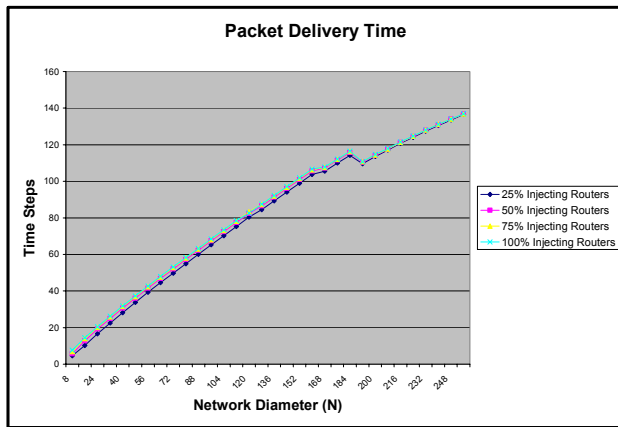


Figure 3: Packet Delivery Time

You can see from the graph that the average delivery time increases approximately linearly with respect to N . The packet injection rate has a very limited effect on the packet delivery rate. One notable feature of the results is the change in trajectory of the graph at approximately $N = 188$. This change is caused by the probabilistic packet state changing rules. In a larger network, a greater percentage of packets have changed to higher states. This change in state comes with a change in how the packet is routed and consequently makes the algorithm perform slightly better.

Another statistic of interest is how long a packet waits to be injected into the network. Figure 4 shows the average number of time-steps a packet waits to be injected.

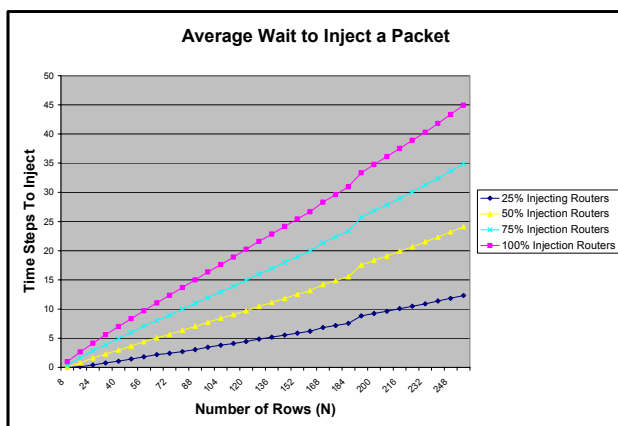


Figure 4: Average Wait to Inject a Packet

You can see from the graph that the average packet injection waiting time increases approximately linearly with N within each injection configuration. However, it is obvious that the injection rate (determined by the number of injection applications) has a significant impact on the injection wait.

The injection of packets is ultimately controlled by the network mechanics. The injection rate is limited because a packet can only be injected when there is a free link at that router. A link becomes free when a packet is delivered to a router. A router will have a free link if it is the final destination of a packet that is delivered to it. A router will also have a free link if it does not receive a packet from one of its adjacent routers at that time step. Therefore, it appears that the average injection rate is linear with respect to N and is bounded by the delivery rate.

4.2 Simulation Analysis

The purpose of using parallel simulation rather than sequential simulation is to speed up the simulation thus reducing simulation time. Simulation time is measured in seconds. ROSS is an event oriented discrete event simulator and therefore simulates the system event by event (in parallel). Accordingly, a simulator's speed is the average number of events that it simulates in a time period. A simulator's speed is also known as its Event Rate. Speed is unitized into events per second.

ROSS uses various mechanisms to increase the speed of the model. KPs are one such mechanism. The effect of KPs on rollbacks and event rate is examined in this section.

However, a fast simulation is not useful unless it faithfully simulates the system being modeled. Therefore, this section appropriately examines the correctness of the simulation before speed or its associated enhancement mechanisms.

4.2.1 Correctness

A significant concern in parallel simulation is the correctness of the results. A useful measure of correctness is repeatability of results.

An optimistic parallel simulation often simulates some of the events in a different order than they actually occur. Therefore, a synchronization

mechanism must ensure that the simulation faithfully emulates the system. However, the complexities of the system (i.e. simultaneous events) make this task difficult.

Therefore, it is important to validate the results of the parallel simulation with the results of the sequential simulation. Consequently, the only way for the results of the parallel simulation to match the sequential model is for the parallel model to be deterministic.

The sample output in Attachment 3 shows that the parallel and sequential models produce identical results (under the same model configuration). As such, the parallel model is deterministic and therefore repeatable.

4.2.2 Speed-up

As noted above, the primary objective of parallel (versus sequential) simulation is speed. This is very important for large problems which require the analysis of very large networks or very long model periods.

The graph in Figure 5 compares the speed of the simulation running in sequential mode (one processor) with the speed of the simulation running in parallel mode with two and four processors. The network diameters (N) used in the simulated configurations ranges from 16 to 256 which equates to 256 to 65,536 LPs.

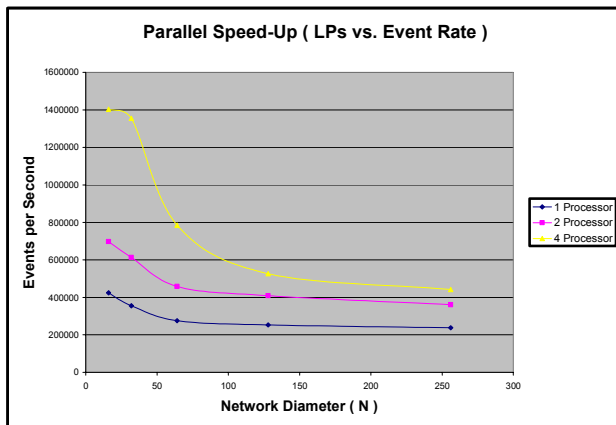


Figure 5: Parallel Speed-Up

The graph shows that for 1024 LPs (N = 32), the 4-Processor simulation is almost four times as

fast as the sequential (1-Processor) simulation. However, for larger networks, the 4-Processor simulation is approximately twice as fast.

Linear speed-up is considered to be optimal (although super-linear speed-up is sometimes experienced). Linear speed-up means that a simulator running with four processors is four times as fast as a simulator running with one processor. Parallel simulation requires synchronization overhead which reduces this speed-up.

The speed-up of a parallel simulation in relationship to linear speed-up is the simulation's efficiency. The graph in Figure 6 shows the efficiency of the graph in Figure 5.

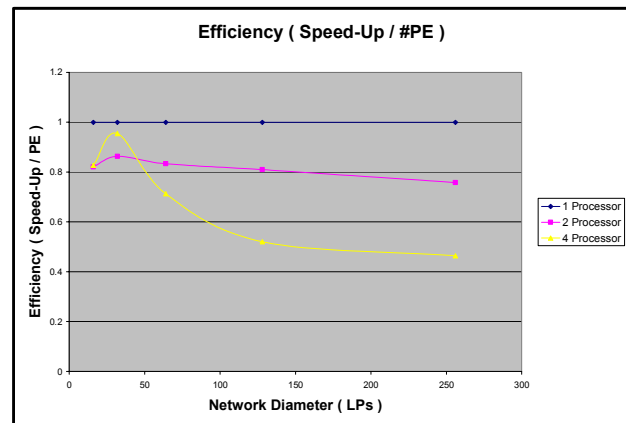


Figure 6: Efficiency

The simulation for smaller networks is close to linear (1), but the simulation of larger graphs drops to approximately .5. Overall, the efficiency of this simulation is very good.

4.2.3 Kernel Processes

ROSS uses KPs, which are groupings of LPs within a PE. One purpose of a KP is to contain rollbacks to a smaller sub-set of LPs within a PE. This is an improvement over rolling back all of the LPs simulated on a given PE. Rolling back an LP that was unaffected by the past message is called a false rollback. The more KPs you have, the fewer false rollbacks you have because each KP represents smaller sub-set of LPs. All else being equal, additional false rollbacks decrease the efficiency of the parallel simulation. However, there are competing overhead

considerations that compromise this assertion. [4, Page 6] shows that the optimum number of KPs (in terms of total events rolled back) is between 32 and 64. The dynamic hot-potato routing model uses 64 KPs.

The hot-potato model was tested to determine the effect of KPs on the simulation. For example, we measured the effect of the number of KPs on rollbacks and, consequently, event rate.

The graphs in Figures 7a, 7b, and 7c show that the number of rollbacks in the simulation of a small network is significantly affected by the number of KPs. However, as the simulation becomes larger, the effect is lessened. This effect is probably due to a trade-off between rollbacks and fossil collection [4]. As the number of KPs increases, the fossil collection overhead also increases. However, the fossil collection for small networks is not significant where the fossil collection for large networks is significant. This is due to the linear relationship between fossil collection overhead and the number of LPs.

Therefore, for a small network, the benefit of increasing the number of KPs far outweighs the costs. However, for larger networks, the benefits and costs approximately cancel each other out.

The graphs in Figures 7a, 7b, and 7c all show essentially the same information. However, due to the extreme variation in Total Events Rolled Back as the network diameter changes, Figure 7a totally obscures the Rollback – KP relationship for the larger networks. Therefore, the information is presented in Figure 7b and 7c using a different scale and only the larger configurations.

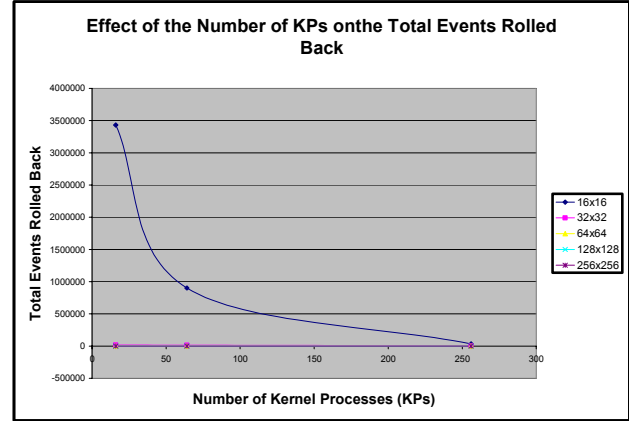


Figure 7a: Effect of the Number of KPs on Events Rolled Back (5 Network Diameters)

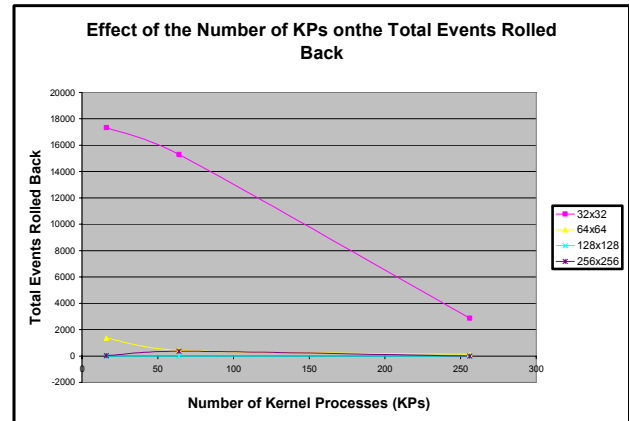


Figure 7b: Effect of the Number of KPs on Events Rolled Back (4 Network Diameters)

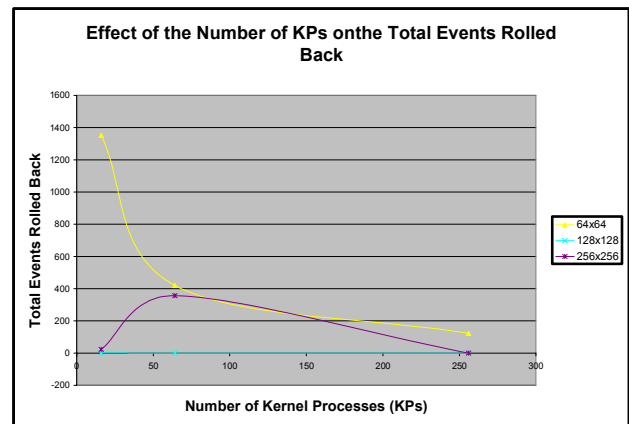


Figure 7c: Effect of the Number of KPs on Events Rolled Back (Larger Networks)

One fascinating feature of the data shown in Figure 7c is the extremely low rollback occurrence for the $N = 128$ configuration, while the larger ($N = 256$) and smaller ($N = 64$) configurations both have more rollbacks. This observation is incongruent with our assumptions regarding the Rollback – KP relationship. The only explanation that I have for this aberrant behavior is the ethereal concept known as serendipity.

The combined affect of the trade-off between the benefits and costs of increasing the number of KPs is captured by the event rate. The graph in Figure 8 shows the relationship between the number of KPs and the event rate. It is clear that the performance of the simulation of the smaller (16×16) network is improved by the use of more KPs. However, as the network size becomes larger, this benefit diminishes.

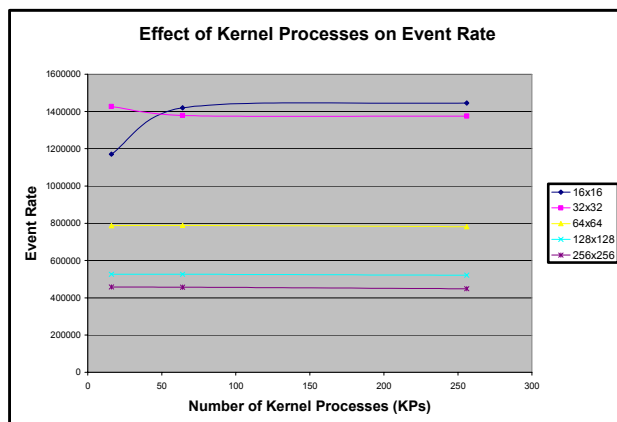


Figure 8: Effect of the Number of Kernel Processes on Event Rate

5. Conclusion

In this paper a practical version of the algorithm presented in [1] was analyzed using computer simulation. The performance of the computer simulation was analyzed as well.

This analysis:

- ✚ confirmed the theoretical analysis presented in [1] and
- ✚ demonstrated efficient parallel speed-up for the network simulation using ROSS.

The simulation analysis provided in this paper combined with the theoretical analysis presented in [1] provides sound evidence of the usefulness of the routing algorithm for optical network routing.

This paper also demonstrated the usefulness of computer simulation for the analysis of network routing algorithms. In particular, computer simulation proved to be well-suited for analyzing greedy routing algorithms, which are difficult to analyze theoretically.

ROSS provide very good speed-up for the parallel processing of the above simulations. This is very important for problems which require the analysis of very large networks. Consequently, ROSS proved to be a very good tool for this type of network simulation.

References

- [1] C. Busch, M. Herlihy, and R. Wattenhoffer. "Routing without flow control." In *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures*, (July 2001).
- [2] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. "GTW: A Time Warp system for shared memory multiprocessors." In *1994 Winter Simulation Conference Proceedings*, pages 1332-1339, (December 1994).
- [3] C.D. Carothers, K. Perumalla and R.M. Fujimoto. "Efficient Parallel Simulation Using Reverse Computation" In *ACM Transactions on Modeling and computer Simulation*, volume 9, number 3, (July 1999).
- [4] C. D Carothers, D. Bauer, S. Pearce, "ROSS: A High-Performance, Low Memory, Modular Time Warp System," In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pp. 53-60, (May 2000).
- [5] Bartzis, Caragiannis, Kaklamanis, and Vergados. "Experimental Evaluation of Hot-Potato Routing Algorithms on 2-Dimensional Processor Arrays". In *EUROPAR: Parallel Processing*, 6th International EURO-PAR Conference. LNCS, (2000).
- [6] C.D. Carothers, R.M. Fujimoto and Y.-B. Lin, "A case study in simulating PCS networks using Time Warp," In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS '95)*, 87-94, IEEE Computer Society Press, (1995).
- [7] R. M. Fujimoto, "Parallel and Distributed Simulation Systems," *A Wiley-Interscience publication*, (2000).
- [8] S. Keshav, "An Engineering approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network," *Addison-Wesley*, (1997).

Attachments

1. Instructions
2. Test Sequence
3. Sample Output
4. Code Printouts
 - Router.c
 - Makefile
5. Main Reference Paper:
 - C. Busch, M. Herlihy, and R. Wattenhoffer. "Routing without flow control." In *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures*, July 2001.
6. Presentation Slides