SUBMICRON SYSTEMS ARCHITECTURE PROJECT
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

A Simple Simulator for Multicomputer Routing Networks

by

Michael J. Pertel

Caltech Computer Science Technical Report

Caltech-CS-TR-92-04

March 5, 1992

The research described in this report was sponsored by
the Defense Advanced Research Projects Agency.

A Simple Simulator for Multicomputer Routing Networks

Michael J. Pertel

March 5, 1992

Abstract

This report describes a concise program for simulating multicomputer routing networks [1]. The simulator is written in less than 200 lines of C code, and a complete listing is provided. Despite being terse, the simulator is exact, fast, and requires little memory.

1 Introduction

This author has had considerable experience designing, writing, and using simulators for experiments in routing network design. The simulator is an integral part of such experiments, but it is common to report only the simulation results. It is desirable to present a listing of the simulator along with simulation results for at least two reasons:

1. Unless the program is derived from a formal specification, it is nearly impossible to completely describe the model implicit in a simulator without reference to the actual code.
2. Given a complete program listing, someone interested in the simulation results is able to scrutinize the model and coding for errors, as well as to reproduce the results.

If a program is to be presented, it should be clear and concise. The design of many simulators makes it impractical to present their code along with their results. Some simulators are written quickly, with little thought given to their clarity. Some include a large amount of code that is irrelevant to the simulation model, such as code for parameter setting, convergence detection, and result reporting. It is also common for exact (as opposed to approximate) simulators to be overly detailed.

The Simple Network Simulator (SNS) program was written to be short and easy to understand. It was originally intended only to reproduce the results of a much larger and more sophisticated hierarchical discrete-event simulator. However, owing to its simplicity, SNS is faster and requires less memory than its larger predecessor.

SNS is a time-driven simulator. Time-driven simulators devote less code to supporting simulation mechanics than discrete-event simulators. Although the simulation-support code is a negligible fraction of a large discrete-event simulator, it would bloat this short program. Moreover, this simulator is easier to understand in a time-driven formulation.

The program is intended to be minimal, so it may seem crude in some respects. Although SNS may be interesting in itself, the primary motivation for this report is merely to make the simulator public. The program should be judged for its clarity and brevity, not for its cleverness or completeness.

Part of the simplicity of SNS stems from the simple router architecture it simulates. This report will describe a simple router design that is nevertheless very general. The generality of the model facilitates comparison of different routing algorithms. The architecture can support a wide range of routing schemes, and it is minimally complex.

The basic simulator uses unbounded buffers for the FIFOs in the router architecture. A trivial extension of the basic program allows simulation of networks with finite buffering. The program modification for finite buffers will be explained, and the FNS (Finite-buffering Network Simulator) code will be presented.

The routing architecture simulated by SNS is sufficiently general that it can support adaptive routing as well as dimension-order routing. Deadlock does not occur with infinite buffering. A modified program with a more liberal definition of an "allowed" routing decision will be presented; it will be referred to as ANS (Adaptive-routing Network Simulator).

The simulators presented in this report were not derived from formal specifications, so no formal proofs of their correctness are given. However, their theory of operation is explained to convince the reader that the simulators implement a realistic model. It is hoped that the simplicity of the programs will obviate formalities.

The routing-network research that inspired SNS is part of a large multicomputer design effort called the Mosaic project [2]. The Mosaic is a 128×128 mesh of single-chip nodes, each containing a 14 MIPS processor, 64 KB memory, and an asynchronous network router. Although SNS can simulate any mesh (arbitrary radix and dimension), the Mosaic topology is most interesting to this project. Some sample simulation results will be presented, but not a large suite. Simulation results will be presented in future reports.

2 Generic Routing Architecture

In order to study the performance of various routing algorithms, the author devised a generic router architecture [3]. The architecture consists of FIFOs on every input channel, a cross-bar between the input FIFOs and output channels, and central control logic (for channel assignment). There are some subtleties in the design of the channel-assignment logic, but the architecture is straightforward. This section will explain briefly why the simple FIFO-and-crossbar design is both necessary and sufficient for a wide class of routing algorithms to be deadlock-free. First, the channel-assignment problem will be explained and a solution described.

2.1 Channel Assignment

For a packet at the front of an input FIFO, some subset of the output channels will move it closer to its destination; these are called the packet's *profitable* channels. There will also be some subset of the output channels along which the packet is *allowed* to be forwarded by the routing algorithm. When a packet is allowed to be forwarded to an output that is not profitable, this is called *misrouting.*

The assignment logic must determine the allowed outputs for the head packet of each input FIFO, and it must assign an allowed output to each input when possible.

Note 2.1 (Channel Assignment) *The control logic must make exclusive assignments of outputs to waiting inputs. A packet at the head of an input FIFO should be forwarded along one of its allowed outputs. An output assignment lasts long enough to spool the entire packet to* *the selected output; no other input may be assigned to an output while it is spooling a packet. The channel assignment can be regarded as an injective mapping from a subset of the waiting input channels to the set of output channels such that every image is an allowed output for its preimage. Several desiderata complicate channel assignment:*

1. **Fairness:** *Each waiting input must eventually be assigned an output.*
2. **Practicality:** *As many inputs should be assigned as possible at all times.*
3. **Symmetry:** *All inputs should be treated identically, as should all outputs.*

The assignment logic should not only assure *progress* but *fairness.* Progress means forwarding arriving packets to their allowed outputs. Fairness requires progress for each input: any waiting input eventually gets assigned an allowed output. Lacking a reason for bias, the assignment logic might as well be *symmetric.* Symmetry means that all inputs are treated identically and all outputs are treated identically. Symmetry is different from fairness: an assignment can be symmetric but not fair, or fair but not symmetric.

A channel-assignment algorithm that is both fair and practical is non-trivial, and other authors have proposed incorrect solutions to the problem. For example, it has been suggested that *examining* the inputs round-robin would give a fair assignment [5], but this is not true. Several inputs may vie for the same output. If the inputs are examined round-robin, then a waiting input is guaranteed a chance to be assigned an output, but it is not guaranteed that an allowed output will ever be available while the input is being examined. An input may wait indefinitely because its allowed output has always been assigned to another input before the waiting input is examined. Round-robin *service* is fair but not practical. If an input has exclusive access to the outputs, then it will be serviced within one packet spooling time, since a busy allowed output will become free as soon as it completes forwarding a packet. Therefore, any input is guaranteed service within $N_{in} \times T_{spool}$ [1] cycles. However, round-robin service is not practical because it significantly and unnecessarily delays making assignments.

[1]Without blocking, spooling time is equal to packet length: $T_{spool} = \mathtt{L}$.

An input must wait its turn before being serviced, even if it has an allowed output that is not needed by any other input.

The author devised and proved [2] a practical fair assignment algorithm based upon circulating a "priority" token round-robin. If an input has a packet waiting for service, it will not relinquish the token until that packet is serviced. The priority input will get serviced as soon as an allowed output becomes free; no other input can be assigned an output unless that output is not allowed for the priority input. The priority packet gets serviced within one packet-spooling time, since a busy output becomes free as soon as it completes spooling a packet. When the priority input is serviced, the token gets forwarded to the next input. If an input is not waiting, it forwards the token. Thus a waiting input eventually gets the token (within $N_{in} \times T_{spool}$ cycles), and the input with the token always gets serviced (within T_{spool} cycles). This algorithm is also practical, because assignments are not unnecessarily delayed — a waiting input gets assigned as soon as one of its allowed outputs is available and not used by another input.

2.2 Deadlock Avoidance

Deadlock is not an issue when the FIFOs are of infinite length, but it is of critical importance to any real router design. Detecting then breaking deadlock is not practical for multicomputer routing networks; the routing algorithm must be intrinsically deadlock-free. One way to avoid deadlock is to eliminate any cyclic dependencies in the packet routing, for example, by restricting the allowed output assignments to a subset of the profitable output assignments. If the packets are required to traverse the dimensions in a fixed order, then the *consumption assumption* is sufficient for deadlock freedom.

Note 2.2 *If a node always consumes a packet that arrives for it, then no packet can be blocked at its destination. In a unidirectional linear array of routers, a packet at the end of the array must be destined for that node, and hence it is consumed. A packet at distance 1 from the end of the array is destined either for that node or for the end of the array; in either case it is not blocked. By induction, the consumption assumption precludes deadlock in a unidirectional linear array. If the directions are independent, the result holds for bidirectional arrays. If*

[2] The formal statement and proof are omitted since the idea is simple.

the dimensions are traversed in a specific order, then deadlock-freedom follows by induction from the last dimension to the first.

There is a more general deadlock-avoidance technique that does not limit the allowed output assignments. If there is no blocking, then there is no deadlock. Blocking can be avoided by allowing the head packet of a FIFO to be misrouted when that FIFO becomes full; that is, all outputs become allowed when the packet's FIFO becomes full. If the number of inputs is equal to the number of outputs, then some output will always be available to absorb a packet from a full FIFO. This technique of avoiding deadlock by allowing misrouting in lieu of blocking requires that any input be routable to any output. Thus, a crossbar is not only sufficient, but necessary for a general router.

If misrouting is allowed, then buffering is needed to assure progress. Buffering allows a packet to wait for a profitable output to become available. Buffering increases performance by decreasing misrouting in a non-blocking network and by decreasing blocking in a blocking network. The buffers need not be FIFOs, nor do the FIFOs have to be on the inputs. For example, there could be a central buffer pool. However, having FIFOs on the inputs is a simple, practical design. The filling of a FIFO can be used to trigger misrouting of that input.

If the FIFO and crossbar architecture were actually implemented, the load applied to the network would have to be controlled. For example, congestion control could be implemented by requiring packet sources to await acknowledgement from their destinations before sending again; this would also preserve packet order between source and destination despite the presence of multiple paths between them. As long as misrouting is less likely than a profitable assignment, packets will make progress on average. By throttling the network load, congestion and misrouting are reduced.

The simulators described in this report do not need misrouting or congestion-control; since ANS uses unbounded buffers, deadlock is not an issue. ANS could easily be modified to use finite buffers and misrouting. The finite-buffer modification is shown in FNS, and `allowed(in,out)` could return true for any `out` when the `in` FIFO is full. The `deject` function would also have to re-`inject` packets dejected by misrouting.

3 Theory of Operation

SNS is a time-driven simulator. Although event-driven simulators may be more efficient for low-activity networks, SNS performs well for high-activity networks. To keep SNS concise, the parameter setting, convergence detection, and other code extraneous to the simulation model are kept minimal. A complete listing of the simulator in C will follow some explanation of how the simulator works. The explanation should convince the reader that the program simulates a routing network. The explanation should be read together with the code.

A `d`-dimensional mesh of radix `R` ($\mathtt{R}^{\mathtt{d}}$ nodes) is simulated. All channels are bidirectional. Each internal node of the mesh is connected to two neighbors (predecessor and successor) in each dimension. There is also a "local" or "internal" channel for the injection and dejection of packets at each node. Thus, every node in the mesh has `2d+1` bidirectional channels.

For each cycle of simulated time, every node in the mesh is simulated. Packets of fixed length `L` are modeled by the `packet` structure. A node contains `2d+1` input FIFOs fed by the two neighbors in each dimension and the local injector. All FIFOs have unbounded capacity.

Note 3.1 *Packets can be blocked only at the head. Since there is infinite buffering, if the head of a packet is enqueued at time T, then the tail will be enqueued at time $T+L-1$. If the head of a packet is transferred at time T, then the whole* `packet` *structure may be transferred at time T, but no other packet may be transferred on that channel before time $T+L$.*

As a packet traverses the network, it may be delayed by having to wait behind other packets in input FIFOs. A packet is also delayed by one cycle when it advances through a crossbar to the next node enroute to its destination. When a `packet` is enqueued in an input FIFO, the `tin` field records the time at which the packet arrives in the queue. If the `tin` of the head of a queue exceeds the current simulation time, then the queue is considered empty.

Note 3.2 *It is safe to enqueue a* `packet` *that arrives at time T during the simulation of time $T-1$, since the* `tin` *field will prevent any action on that packet until time T.*

Note 3.3 *The* `packet` *structures corresponding to all packets arriving at an input FIFO at or before time T have been enqueued prior to* *simulating the node for time T. A packet arriving at time T must have been sent at time $T-1$, and all nodes are simulated through time $T-1$ before any are simulated for time T.*

During each cycle of simulated time, each node will inject a packet with probability $\frac{4A}{RL}$. Since the destinations are chosen at random, this gives an applied load of $\frac{\frac{1}{4}NL(4A/RL)}{N/R} = A$. The `tls` (time-last-send) field of the `nodestate` structure prevents injection overlap.

Note 3.4 *Injecting at time* $\texttt{tls}_{new} = \max(\texttt{curtime}, \texttt{tls}_{old} + \texttt{L})$ *prevents injection overlap. The head of a new packet is not transmitted until after the tail of the previous packet has been transmitted. The* `tin` *field allows a new* `packet` *to be placed in the local injection input FIFO immediately, though* `tls = tin = tsent` *may exceed* `curtime`.

Before a packet can be forwarded from one node to the next, it must reach the head of the input FIFO and an `allowed` output must be free. Output overlap is avoided by recording `tfree` for each output. If a packet is forwarded along an output at time T, then $T \geq \texttt{tfree}_{old}$ and $\texttt{tfree}_{new} = T + L$. Input overlap is avoided by recording `tnh` (time-next-head) for each input. If a packet is removed from the head of a queue at time T, then the next packet cannot reach the head of the queue before $\texttt{tnh} = T + L$.

Note 3.5 *Forwarding packet* `p` *from input* `i` *to output* `o` *is* `allowed` *at* `curtime` *only if* `max(p->tin,tnh[i],tfree[o])<=curtime`. *If the packet is forwarded at* `curtime`, *then* `p->tin=curtime+1` *corresponds to the packet head arriving at the next node one cycle later, and assigning* `tnh[i]=tfree[o]=curtime+L` *represents the time required to spool the packet from the input FIFO through the output channel.*

With dimension-order routing (DOR), a packet can use an output channel only if that output reduces its offset in some dimension and its offsets in all previous (lower) dimensions are zero. The timing requirements above together with the DOR protocol determine whether forwarding the head of an input queue to a given output at the current simulation time is `allowed`.

```
/*      SNS.c --- Simple Network Simulator (168 lines)
 */
#include <stdio.h>
#include <malloc.h>
double  drand48();
#define CHECK(c,m) {if(!(c)){printf("ERROR: %s\n",m); exit(7);}}
#define PBY(p) (drand48()<(p))
#define MAX(a,b) (((a)>(b))?(a):(b))
#define ABS(x)  (((x)<0)?-(x):(x))
#define CHANGE(old,new) ABS(((new)-(old))/(old))
int pwr(x,y) int x,y; {int r=1; for(;y>0;y--) r*=x; return r;}

/* Parameters
 */
#define N       16384           /* number of nodes */
#define R       128             /* radix */
#define d       2               /* dimension */
#define A       .5              /* applied load */
#define L       32              /* packet length in flits */

#define TOL     .03             /* convergence tolerance */
#define INTERVAL 1000           /* initial simulation interval */

#define B       (N/R)           /* bisection BW in flits/cycle */
#define NIN     (2*d+1)         /* number of router inputs */
#define NOUT    (2*d+1)         /* number of router outputs */

/* Measurements
 */
double numrecd=0.;              /* number of packets received */
double totlat=0.;               /* sum of received-packet latencies */
double tothops=0.;              /* sum of received-packet distances */

#define T (totlat/numrecd)      /* average TOTAL latency */
#define TP ((numrecd*L)/curtime)/* throughput in flits/cycle */
#define U (.25*TP/B)            /* bisection utilization */
#define D (tothops/numrecd)     /* average distance */

/* Data Structures
 */
typedef struct packet packet;
struct packet{int dest,tsent,tin,nhops; packet *next;};
typedef struct nodestate nodestate;
struct nodestate
{packet *head[NIN],*tail[NIN]; int tnh[NIN],tfree[NOUT],tls,pin,pout;};

/* Numbering Conventions:
 * Dimensions numbered from 0: x=0, y=1, z=3, etc...
 * Channels numbered: local=0, xpred=1, xsucc=2, ypred=3, ...
 * Node (x,y,z,...) numbered: ... + zR^2 + yR + x
 */
#define DIMOF(in) (((in)-1)/2)
#define PRED(dim) (2*(dim)+1)
#define SUCC(dim) (2*(dim)+2)
```

```
#define END(out)  (((out)%2)?((out)+1):((out)-1))
#define COORD(n,dim) (((n)/pwr(R,dim))%R)
#define NGHBR(n,o) (((o)%2)?(n)-pwr(R,DIMOF(o)):(n)+pwr(R,DIMOF(o)))

/* Simulator
 */
int curtime=0;                  /* current simulation time */
nodestate node[N];              /* simulator state */
initnode(n) nodestate *n;
{
        int i; n->pin=n->pout=n->tls=0;
        for(i=0;i<NIN;i++)
                {n->head[i]=n->tail[i]=(packet*)0; n->tnh[i]=n->tfree[i]=0;}
}
init(){ int n; for(n=0; n<N; n++) initnode(&(node[n])); }

main()
{
        int n,etime=INTERVAL; double oldT,curT=1.;
        init(); printf("\n\n");
        do{     oldT=curT;
                for(  ; curtime<etime; curtime++)
                        for(n=0; n<N; n++)
                                simulate(n);
                printf("N=%d, R=%d, d=%d, L=%d, A=%g, curtime=%d\n",
                                N,R,d,L,A,curtime);
                printf("numrecd=%g\n",numrecd);
                printf("D=%g, d*(R-1/R)/3.=%g\n",D,d*(R-1./R)/3.);
                printf("T(%g)=%g\n",U,T);
                printf("CHANGE(oldT,T)=%g, TOL=%g\n",CHANGE(oldT,T),TOL);
                printf("\n");
                fflush(stdout);
                curT=T; etime*=2;
        } while(CHANGE(oldT,curT)>TOL); return 0;
}

/* Node Behavior
 */
#define PIN     node[n].pin
#define POUT    node[n].pout
simulate(n) int n;
{
        int in,out;
        if(PBY(4.*A/(R*L))) inject(n); findpin(n);
        for(in=0; in<NIN; in++) for(out=0; out<NOUT; out++)
                if(allowed(n,(PIN+in)%NIN,(POUT+out)%NOUT))
                {
                        forward(n,(PIN+in)%NIN,(POUT+out)%NOUT);
                        if(!in) {PIN=(PIN+1)%NIN; findpin(n);}
                        if(!out) POUT=(POUT+1)%NOUT;
                }
}

findpin(n) int n;
{
```

```
        int i; packet *p; nodestate *nd = &(node[n]);
        for(i=0; i<NIN; i++)
                if((p=nd->head[PIN])&&(MAX(p->tin,nd->tnh[PIN])<=curtime))
                        return;
                else PIN=(PIN+1)%NIN;
}

inject(n) int n;
{
        packet *p=(packet*)malloc((unsigned)sizeof(packet));
        node[n].tls = p->tin = p->tsent = MAX(curtime,node[n].tls+L);
        p->dest = drand48()*N; p->nhops=0; p->next=0;
        enqueue(p,n,0);
}

int allowed(n,in,out) int n,in,out;
{
        int dim,nc,pc; packet *p=node[n].head[in];
        if(!p || p->tin>curtime) return 0;      /* p arrived */
        if(node[n].tnh[in]>curtime) return 0;   /* p at head */
        if(node[n].tfree[out]>curtime) return 0;/* out free  */
        for(dim=0; dim<d; dim++)
        {
                nc=COORD(n,dim); pc=COORD(p->dest,dim);
                if(nc<pc) return out==SUCC(dim);
                        else if(nc>pc) return out==PRED(dim);
        }
        CHECK(p->dest==n,"profitable()"); return out==0;
}

forward(n,in,out) int n,in,out;
{
        packet *dequeue(); packet *p=dequeue(n,in); p->tin=curtime+1;
        node[n].tnh[in]=node[n].tfree[out]=curtime+L;
        if(out==0) deject(p);
                else p->nhops++, enqueue(p,NGHBR(n,out),END(out));
}

deject(p) packet *p;
{
        numrecd++; totlat+=p->tin-p->tsent; tothops+=p->nhops;
        free((char*)p);
}

/* Misc Functions
 */
packet *dequeue(n,in) int n,in;
{
        packet *p=node[n].head[in]; node[n].head[in]=p->next;
        if(p==node[n].tail[in]) node[n].tail[in]=(packet*)0;
        p->next=(packet*)0; return p;
}

enqueue(p,n,in) packet *p; int n,in;
{
```

```
        nodestate *nd=&(node[n]);
        if(nd->head[in]) nd->tail[in]->next=p; else nd->head[in]=p;
        nd->tail[in]=p; p->next=0;
}
```

4 Finite Buffers

The effect of having finite buffers is to block the forwarding of a packet when the receiving FIFO is full. There are at least three possible models for finite buffers, differing in how the forwarding of a packet to a full FIFO is handled:

1. An output is "free" even if it is blocked by a full FIFO. If an input is assigned to a blocked output, the input merely waits for the output to become ready before forwarding the packet.
2. An output is "busy" whenever it is not ready to receive a packet. An input is never assigned to a blocked output, so an input remains unassigned if all its allowed outputs are blocked.
3. A blocked output may be assigned to an input, but the assignment is aborted once the blocking is detected.

In the first case, a packet might wait longer than necessary if it is assigned to a blocked output when another allowed output is not blocked. The second case is optimal in that a packet will not wait if it can be forwarded along one of its allowed outputs. The third case seems functionally equivalent to the second, but it may interfere with the algorithm used to ensure fairness. If the token is advanced when an assignment is made, but the assignment is later aborted, then holding the token no longer guarantees service.

SNS can be extended to handle finite buffers by keeping track of the number of packets in each FIFO and making an assignment that would overfill a FIFO not `allowed`. Every FIFO is made the same size, except the injection FIFOs remain unbounded. When the network throughput cannot support the applied load, the injection queues grow without bound. The `nodestate` structure is expanded to record FIFO occupancy. The `enqueue` and `dequeue` functions are modified to keep track of FIFO occupancy. The `allowed` function is modified to require that the destination FIFO have room for another packet. A listing of the modified program (FNS) is provided below.

If the head packet of a full FIFO is forwarded at time T, should another packet be allowed to enter the FIFO at time T? The impact of this choice on performance should be negligible. FNS handles the situation inconsistently, but this should have no effect. During the simulation of time T, FNS simulates each router in turn. If the destination router is simulated before the source router, then the full FIFO might be dequeued, thereby allowing the source to forward on that cycle. If the source is simulated first, the FIFO will still be full and the packet will not be forwarded until the next cycle. There seems little justification for complicating the program to eliminate this negligible inconsistency. Over-specified models lead to overly-complicated simulators. Note that no such inconsistency exists in SNS. The test for full FIFO in FNS is the only case where a state change made at time T can effect a decision made at time T.

It should be noted that the simulator makes no approximation in manipulating whole packets rather than individual flits. With infinite buffering this is clear: FIFOs never fill and hence never block, and crossbars block packets only at the head. Even with finite buffering, as long as FIFO lengths are integer multiples of the packet length, packets will only be blocked at the head.

Note 4.1 Packet-Blocking Property: *If all packets are length L and all FIFO lengths are an integer multiple of L, then packets are blocked only at their heads. If the head of a packet is transferred at time T, then the tail will be transferred at time $T + L$. Packet sources, sinks, and switches all treat packets as units. If the FIFOs do not divide packets, then packets will be indivisible throughout the network.*

This property simplifies the simulator by eliminating the need to decompose packets into flits. The simulator also runs faster by an amount proportional to the packet length. By accepting a small restriction on the FIFO sizes that can be simulated, the simulator can be made significantly simpler and faster while still being exact.

```
/*   FNS.c --- Finite-buffering Network Simulator (177 lines)
 */
#include <stdio.h>
#include <malloc.h>
double  drand48();
#define CHECK(c,m) {if(!(c)){printf("ERROR: %s\n",m); exit(7);}}
#define PBY(p) (drand48()<(p))
#define MAX(a,b) (((a)>(b))?(a):(b))
#define ABS(x)  (((x)<0)?-(x):(x))
#define CHANGE(old,new) ABS(((new)-(old))/(old))
int pwr(x,y) int x,y; {int r=1; for(;y>0;y--) r*=x; return r;}

/* Parameters
 */
#define N       16384           /* number of nodes */
#define R       128             /* radix */
#define d       2               /* dimension */
#define Q       1               /* length of non-injection input FIFOs */
#ifndef A
#define A       .5              /* applied load */
#endif
#define L       32              /* packet length in flits */

#define TOL     .03             /* convergence tolerance */
#define INTERVAL 1000          /* initial simulation interval */

#define B       (N/R)           /* bisection bandwidth in flits/cycle */
#define NIN     (2*d+1)         /* num. node inputs including local */
#define NOUT    (2*d+1)         /* num. node outputs including local */

/* Measurements
 */
double numrecd=0.;              /* number of packets received */
double totlat=0.;               /* sum of rcvd-pckt latencies */
double tothops=0.;              /* sum of rcvd-pckt distances */

#define T (totlat/numrecd)      /* average latency */
#define TP ((numrecd*L)/curtime)/* throughput in flits/cycle */
#define U (.25*TP/B)            /* bisection utilization */
#define D (tothops/numrecd)     /* average distance */

/* Data Structures
 */
typedef struct packet packet;
struct packet{int dest,tsent,tin,nhops; packet *next;};
typedef struct nodestate nodestate;
struct nodestate
{packet *head[NIN],*tail[NIN]; int len[NIN],tnh[NIN],tfree[NOUT],tls,pin,pout;};

/* Numbering Conventions:
 * Dimensions numbered from 0: x=0, y=1, z=3, etc...
 * Channels numbered: local=0, xpred=1, xsucc=2, ypred=3, ...
 * Node (x,y,z,...) numbered: ... + zR^2 + yR + x
 */
```

```
#define DIMOF(in) (((in)-1)/2)
#define PRED(dim) (2*(dim)+1)
#define SUCC(dim) (2*(dim)+2)
#define END(out)  (((out)%2)?((out)+1):((out)-1))
#define COORD(n,dim) (((n)/pwr(R,dim))%R)
#define NGHBR(n,o) (((o)%2)?(n)-pwr(R,DIMOF(o)):(n)+pwr(R,DIMOF(o)))

/* Simulator
 */
int curtime=0;                  /* current simulation time */
nodestate node[N];              /* simulator state */
initnode(n) nodestate *n;
{
        int i; n->pin=n->pout=n->tls=0;
        for(i=0;i<NIN;i++)
        {n->head[i]=n->tail[i]=(packet*)0; n->len[i]=n->tnh[i]=n->tfree[i]=0;}
}
init(){ int n; for(n=0; n<N; n++) initnode(&(node[n])); }

main()
{
        int n,etime=INTERVAL; double oldT,curT=1.;
        init(); printf("\n\n");
        do{     oldT=curT;
                for(  ; curtime<etime; curtime++)
                        for(n=0; n<N; n++)
                                simulate(n);
                printf("N=%d, R=%d, d=%d, Q=%d, L=%d, A=%g, curtime=%d\n",
                                N,R,d,Q,L,A,curtime);
                printf("numrecd=%g\n",numrecd);
                printf("D=%g, d*(R-1/R)/3.=%g\n",D,d*(R-1./R)/3.);
                printf("T(%g)=%g\n",U,T);
                printf("CHANGE(oldT,T)=%g, TOL=%g\n",CHANGE(oldT,T),TOL);
                printf("\n");
                fflush(stdout);
                curT=T; etime*=2;
        } while(CHANGE(oldT,curT)>TOL); return 0;
}

/* Node Behavior
 */
#define PIN     node[n].pin
#define POUT    node[n].pout
simulate(n) int n;
{
        int in,out;
        if(PBY(4.*A/(R*L))) inject(n); findpin(n);
        for(in=0; in<NIN; in++) for(out=0; out<NOUT; out++)
                if(allowed(n,(PIN+in)%NIN,(POUT+out)%NOUT))
                {
                        forward(n,(PIN+in)%NIN,(POUT+out)%NOUT);
                        if(!in) {PIN=(PIN+1)%NIN; findpin(n);}
                        if(!out) POUT=(POUT+1)%NOUT;
                }
}
```

```
findpin(n) int n;
{
        int i; packet *p; nodestate *nd = &(node[n]);
        for(i=0; i<NIN; i++)
                if((p=nd->head[PIN])&&(MAX(p->tin,nd->tnh[PIN])<=curtime))
                        return;
                else PIN=(PIN+1)%NIN;
}

inject(n) int n;
{
        packet *p=(packet*)malloc((unsigned)sizeof(packet));
        node[n].tls = p->tin = p->tsent = MAX(curtime,node[n].tls+L);
        p->dest = drand48()*N; p->nhops=0; p->next=0;
        enqueue(p,n,0);
}

#define RDY(o)  (node[NGHBR(n,o)].len[END(o)]<Q) /* FIFO not full */
int allowed(n,in,out) int n,in,out;
{
        int dim,nc,pc; packet *p=node[n].head[in];
        if(!p || p->tin>curtime) return 0;      /* p arrived? */
        if(node[n].tnh[in]>curtime) return 0;   /* p at head? */
        if(node[n].tfree[out]>curtime) return 0;/* out free?  */
        for(dim=0; dim<d; dim++)
        {
                nc=COORD(n,dim); pc=COORD(p->dest,dim);
                if(nc<pc) return (out==SUCC(dim) && RDY(out));
                else if(nc>pc) return (out==PRED(dim) && RDY(out));
        }
        CHECK(p->dest==n,"profitable()"); return out==0;
}

forward(n,in,out) int n,in,out;
{
        packet *dequeue(), *p;
        int dest=NGHBR(n,out), dchan=END(out);
        p=dequeue(n,in); p->tin=curtime+1;
        node[n].tnh[in]=node[n].tfree[out]=curtime+L;
        if(out==0) deject(p);
                else p->nhops++, enqueue(p,dest,dchan);
}

deject(p) packet *p;
{
        numrecd++; totlat+=p->tin-p->tsent; tothops+=p->nhops;
        free((char*)p);
}

/* Misc Functions
 */
packet *dequeue(n,in) int n,in;
{
        nodestate *nd=&(node[n]); packet *p=nd->head[in];
```

```
        nd->len[in]--; nd->head[in]=p->next;
        if(p==nd->tail[in]) nd->tail[in]=(packet*)0;
        p->next=(packet*)0; return p;
}

enqueue(p,n,in) packet *p; int n,in;
{
        nodestate *nd=&(node[n]); nd->len[in]++;
        if(nd->head[in]) nd->tail[in]->next=p; else nd->head[in]=p;
        nd->tail[in]=p; p->next=0;
        if(in && nd->len[in]>Q){printf("Q OVERFLOW!\n"); exit(66);}
}
```

5 Adaptive Routing

Multicomputer routing networks typically use single-path routing. In particular, dimension-order routing is the standard algorithm for mesh topologies, since it is intrinsically deadlock-free and easy to implement in VLSI [4]. It is interesting to explore the possibility of multipath routing to improve network performance or reliability. One type of multipath routing is adaptive routing [5]. In adaptive routing, the path followed by a packet through the network is effected by the local traffic conditions it encounters enroute. Most attention has been focused on minimal routing, in which the packet follows a shortest path from source to destination.

Note 5.1 *In a d-dimensional mesh, the number of shortest paths between two nodes separated by* $(\Delta x_1, \ldots, \Delta x_d)$ *is* $\frac{(\Delta x_1 + \ldots + \Delta x_d)!}{(\Delta x_1)! \ldots (\Delta x_d)!}$.

This author's study of adaptive routing has shown it to fulfill little of its promise. Earlier results showing that adaptive routing improved network throughput over dimension-order routing [5] were an artifact of the adaptive routers being given more buffering than the dimension-order routers. Given equal buffering, the performance gap disappears. The simplest architecture required to support adaptive routing [3] would require significantly more chip area (and run significantly slower) than a VLSI implementation of dimension-order routing [4]. A dimension-order router (DOR) is much smaller, and requires only a few percent of the area of a single-chip multicomputer node [2]. The simpler, asynchronous circuitry of a DOR is faster and leads to higher performance.

It is trivial to modify SNS to simulate adaptive routing instead of dimension-order routing. Minimal adaptive routing is obtained simply by making all profitable channels `allowed` in the channel assignment. Earlier studies that used various routing metrics to bias the choice between multiple profitable assignments showed that it makes little difference how the assignment is chosen [5]. The ANS (Adaptive-routing Network Simulator) chooses the first available profitable output. ANS differs from SNS only in its `allowed` function:

```
int allowed(n,in,out) int n,in,out;
{
        packet *p=node[n].head[in]; int pc,nc,dim=DIMOF(out);
        if(!p || p->tin>curtime) return 0;        /* p arrived? */
        if(node[n].tnh[in]>curtime) return 0;     /* p at head? */
```

```
        if(node[n].tfree[out]>curtime) return 0;  /* out free?  */
        if(p->dest == n) return out==0;           /* p at dest? */
        pc=COORD(p->dest,dim), nc=COORD(n,dim);
        if(nc<pc) return out==SUCC(dim);
        if(nc>pc) return out==PRED(dim);
        return 0;                               /* not profitable */
}
```

6 Convergence

This section will present some facts for consideration when using the simulator and estimating the accuracy of results. It will be assumed that the measurement of interest is the average network latency for a specified applied load, but most of the considerations presented are equally applicable to other measurements. For example, the simulators might be modified to measure injection and cut-through latency separately, instead of total latency [3] from generation to reception. This section will discuss the termination criterion used in SNS, its limitations, and recommendations for obtaining accurate results.

6.1 Ancillary Measurements

SNS contains a simple test for the convergence of the average latency measurement, and it terminates when convergence is detected. Other measurements, besides the measurement of interest, should be monitored. The simulator prints these measurements for user inspection, but they are not considered in the termination decision.

- The throughput should be monitored to verify that it converges to the applied load. If the throughput cannot support the applied load, then the latency will grow without bound as the injection-queue lengths increase.
- The average distance should converge to $\frac{d}{3}\left(R - \frac{1}{R}\right)$.

  Note 6.1 *In a bidirectional linear array of R nodes, there are R paths of length 0 and $2(R-l)$ paths of length $1 \leq l \leq R-1$. The total number of paths is $R + \sum_{l=1}^{R-1} 2(R-l) = R^2$. The average*

[3] The measured head-to-head latency does not include the `L`-cycle spooling time.

distance is $\overline{D} = \frac{1}{R^2}\sum_{l=1}^{R-1} 2(R-l)l = \frac{2}{R}\sum_{l=1}^{R-1} l - \frac{2}{R^2}\sum_{l=1}^{R-1} l^2 = \frac{1}{3}\left(R - \frac{1}{R}\right)$. [4]

The latency is proportional to the distance, and the latency cannot converge if the distance does not.

- The difference between the number of packets sent and the number received should also be monitored; it represents the number of packets in the network FIFOs, and it should be a small fraction of the total number of packets. SNS does not wait for the network to reach equilibrium (steady-state queue lengths) before taking measurements; this may slow convergence.

Note that throughput matches applied load if and only if the total number of packets in the network buffers is stable. Throughput is less than applied load if and only if the average queue lengths are increasing. Thus, it suffices to monitor the convergence of the throughput to the applied load. Ideally, the throughput should converge before recording latency measurements. The average distance of the set of packets whose latencies are recorded should be close to the true average distance (otherwise more packets should be simulated).

6.2 Effect of `TOL`

SNS contains only a simple check for convergence, but it serves at least three functions with very little code:

1. It provides sequential snapshots of parameter measurements so that the user can monitor convergence.
2. It provides a more flexible criterion for simulator termination than merely specifying a simulated-time interval.
3. It provides a framework for a more elaborate termination decision.

The user specifies an initial simulation `INTERVAL` and a convergence tolerance `TOL`. The simulator is run until the end of a time interval, then the interval is doubled. If the average-latency measurement `T` changed by less than `TOL` when the interval was doubled, the simulator halts. This method does not guarantee that the latency measurement

[4] $\frac{2}{R}\sum_{l=1}^{R-1} l = \frac{2}{R}\frac{(R-1)R}{2} = R-1$ and $\frac{2}{R^2}\sum_{l=1}^{R-1} l^2 = \frac{2}{R^2}\left(\frac{(R-1)^2 R}{3} + \frac{(R-1)R}{6}\right) = \frac{2}{3}R + \frac{1}{3}\frac{1}{R} - 1$

is within TOL of the true asymptotic value. For example, if the measured latency is consistently less than the asymptotic latency, but gets closer as the simulation interval is increased, then the simulator may stop prematurely if the convergence is slow.

Note 6.2 *If the error in the measured latency is proportional to the inverse square of the simulated interval, then the simulator stops when the error is approximately* $\frac{1}{\sqrt{2}-1} \cdot \texttt{TOL} \approx 2.4 \cdot \texttt{TOL}$. [5]

6.3 Effect of INTERVAL

The simulator is used to estimate the asymptotic average network latency $\overline{T} = \lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} T_i$. The simulator does this by computing the average of the latencies of n packets $\langle T\rangle_n = \frac{1}{n}\sum_{i=1}^{n} T_i$. It is assumed that the latencies of the individual packets T_i are independent, identically distributed, random variables. As n increases, $\langle T\rangle_n$ converges to $\overline{T}$ by definition, provided $\overline{T}$ exists (which we assume). We wish to know how close we can expect $\langle T\rangle_n$ to be to $\overline{T}$ for a given n as an aid to both using the simulator (choosing the simulation interval) and interpreting the results (estimating the accuracy of measurements).

Note 6.3 *The number of packets sent is a random function of the simulated time interval whose average is* `curtime*(4*U*B)/L`.[6] *The bisection bandwidth is* `N/R` *flits per cycle in each direction, and the bisection utilization* `U` *converges to the normalized applied load* `A`. *SNS will not terminate until* `curtime >= 2*INTERVAL`, *so the number of packets sent will be at least* `MINPKT = INTERVAL*8*A*N/(R*L)`. *The number of packets received converges to the number sent, provided the throughput matches the applied load. To assure that at least* `MINPKT` *packets are injected, choose* `INTERVAL = MINPKT*L/(8*A*N/R)`.

The latency measurement $\langle T\rangle_n$ is a random variable with the same average ($\overline{T}$) as the individual T_i, but a smaller variance. It is well known that the RMS error (standard deviation) of the average of n

---

[5] $\frac{T_I - T_{I/2}}{T_{I/2}} = \frac{\left(T_\infty - T_e/\sqrt{I}\right) - \left(T_\infty - T_e/\sqrt{I/2}\right)}{\left(T_\infty - T_e/\sqrt{I/2}\right)} = \frac{\left(\sqrt{2}-1\right) T_e/\sqrt{I}}{\left(T_\infty - T_e/\sqrt{I/2}\right)} \approx \left(\sqrt{2}-1\right) \frac{T_e/\sqrt{I}}{T_\infty} \le \mathrm{TOL}$

[6] One quarter of all packets cross the bisection in a given direction (for random traffic) and throughput=utilization×bandwidth. Flit throughput is `L` times packet throughput.

measurements decreases as $1/\sqrt{n}$. [7] The uncertainty in the measured latency is $\frac{1}{\sqrt{n}}\frac{\sqrt{\overline{(T-\overline{T})}^2}}{\overline{T}}$. We can estimate the accuracy as a function of n if we can estimate the variance of the latency.

Accurately estimating the variance of the latency is outside the scope of this report. However, a crude approximation provides some insight. A packet's latency is the product of the distance it travels and the average delay per hop. If we neglect the latency variation due to the variation in queue lengths, we can restrict attention to the variance of the packet distance. For the latency measurement to have converged to within `TOL` of $\overline{T}$, it is necessary (though not sufficient) that the average distance of the packet exchanges simulated has converged to within `TOL` of the true mean distance. By computing the number of packets that must be simulated to achieve a desired convergence of the distance measurement, we can choose `INTERVAL` large enough to guarantee this convergence. This technique gives an `INTERVAL` size that is large enough to give good statistics and avoid spurious termination while being small enough to minimize execution time.

6.3.1 Mesh Path Statistics

It was previously shown that the average distance in a radix-R d-dimensional mesh is $d(R-1/R)/3$, which gives the following identity:

$$\frac{1}{R^2}\sum_{i=0}^{R-1}\sum_{j=0}^{R-1}|i-j| = \frac{R-1/R}{3}$$

The mean-square distance in a one-dimensional mesh is:

$$\begin{aligned}
\frac{1}{R^2}\sum_{i=0}^{R-1}\sum_{j=0}^{R-1}|i-j|^2 &= \frac{1}{R^2}\left(\sum_{i=0}^{R-1}\sum_{j=0}^{R-1} i^2 + \sum_{i=0}^{R-1}\sum_{j=0}^{R-1} 2ij + \sum_{i=0}^{R-1}\sum_{j=0}^{R-1} j^2\right) \\
&= \frac{1}{R^2}\left(2R\sum_{i=0}^{R-1} i^2 - 2\left(\sum_{i=0}^{R-1} i\right)^2\right) \\
&= \frac{R^2-1}{6}
\end{aligned}$$

[7] $\sqrt{\overline{\left(\langle T\rangle_n - \overline{T}\right)^2}} = \sqrt{\overline{\langle T\rangle_n^2} - \overline{T}^2} = \sqrt{\left(\frac{n(n-1)}{n^2}\overline{T}^2 + \frac{n}{n^2}\overline{T^2}\right) - \overline{T}^2} = \sqrt{\frac{\overline{T^2}-\overline{T}^2}{n}} = \frac{1}{\sqrt{n}}\sqrt{\overline{\left(T-\overline{T}\right)}}$

These two results can be used to calculate the mean-square distance in a two-dimensional mesh:

$$\begin{aligned}\frac{1}{R^4}\sum_{i=0}^{R-1}\sum_{j=0}^{R-1}\sum_{k=0}^{R-1}\sum_{l=0}^{R-1}(|i-j|+|k-l|)^2 &= \frac{2}{R^2}\sum_{i=0}^{R-1}\sum_{j=0}^{R-1}|i-j|^2+2\left(\frac{1}{R^2}\sum_{i=0}^{R-1}\sum_{j=0}^{R-1}|i-j|\right)^2\\ &= \frac{1}{9}\left(5R^2-7+\frac{2}{R^2}\right)\end{aligned}$$

The variance of the distance in a two-dimensional mesh is:

$$\begin{aligned}\overline{D^2}-\overline{D}^2 &= \frac{1}{9}\left(5R^2-7+\frac{2}{R^2}\right)-\left(\frac{2(R-1/R)}{3}\right)^2\\ &= \frac{1}{9}\left(R^2+1-\frac{2}{R^2}\right)\end{aligned}$$

For large R, the standard deviation is $\approx \frac{R}{3}$ and the average distance is $\approx \frac{2R}{3}$; thus the uncertainty (standard deviation over mean) of the average of n measurements is approximately $\frac{1}{2\sqrt{n}}$ for a two-dimensional mesh. For a one-dimensional mesh, the variance is:

$$\frac{R^2-1}{6}-\left(\frac{R-1/R}{3}\right)^2=\frac{1}{18}\left(R^2+1-\frac{2}{R^2}\right)$$

For large R, the standard deviation is $\approx \frac{R}{3\sqrt{2}}$ and the average is $\approx \frac{R}{3}$ so the uncertainty of the average of n measurements is approximately $\frac{1}{\sqrt{2n}}$.

A conservative estimate of the fractional uncertainty in the measured average distance for n packets is $\frac{1}{\sqrt{n}}$. [8]

6.4 Recommendations

To generate results of a desired accuracy, TOL should be chosen to be one third that value. Recall that for monotone convergence, the difference between two successive intervals is about .414 times the difference of the later value from the asymptote. It is not enough to choose a small TOL, since the simulator will halt if two successive measurements are coincidentally close. The initial simulation INTERVAL should be chosen large enough to ensure good statistics. A reasonable

[8] $\frac{\sqrt{\overline{(D-\overline{D})^2}}}{\overline{D}}=\frac{1}{\sqrt{2d}}+\frac{3}{2\sqrt{2d}R^2}+\mathrm{O}\left(\frac{1}{R^4}\right)$ for any mesh [6].

choice of INTERVAL is one that guarantees simulation of enough packets to make the error in the measured average distance less than TOL. These suggestions can be coded as follows.

```
#define ACCURACY   .03
double  TOL      = ACCURACY/3.;
#define MINPKT     (1./(TOL*TOL))
int     INTERVAL = (int)(MINPKT*L/(8*A*N/R));
```

6.4.1 Caveat

The above technique does not guarantee the specified ACCURACY upon termination. Simulator convergence is not merely the convergence of a sample average to a distribution mean. The simulator may converge more slowly than $1/\sqrt{n}$ or may not converge at all. This is because the network conditions (e.g., the average queue lengths) may change over time, so the measurement distribution may evolve over time. All of the considerations in this section have been premised upon random sampling of a static latency distribution, but the latency distribution is not static. As the applied load increases, the time required for the queues to reach their equilibrium length distribution increases. Beyond a maximum applied load (which depends upon the network parameters) the mesh cannot support the necessary throughput, queue lengths keep increasing, and there is no convergence. This section has considered how long the simulator must be run for measurements to converge after the network has reached steady state, but it has not considered how long the simulator must be run before the network reaches steady state. For small applied loads the network equilibrates quickly, but for heavy traffic the equilibration time may dominate the convergence.

The simulator should wait for the network to reach equilibrium before recording measurements. This can be done by monitoring the difference between the number of packets sent and the number received. The network can be regarded as having reached equilibrium once this difference has converged to within a specified tolerance. Equivalently, the simulator could wait until the measured throughput is within tolerance of the applied load. Latency measurements could be discarded until the network has equilibrated, and then SNS-style measurement and convergence detection could be applied to the equilibrated network. If the number of packets stored in the network FIFOs does not converge (if it grows at a non-diminishing rate) then the simulator should terminate. Monitoring the difference between the number of packets sent and received also provides a measure of the average queue length — divide the number of packets in queues by the number of queues.

SNS was designed to be simple, and code extraneous to the model (e.g., sophisticated convergence detection) was specifically avoided. Simplicity of the program requires sophistication of the user.

7 128×128 Mesh Results

Since the topic of this report is the simulator itself, rather than the results of the simulator, this section will contain only a few sample simulation results.

The results presented below were computed using `R=128`, `d=2`, `L=32`, `TOL=.03`, and `INTERVAL=1000`. For `A=.5`, the chosen `INTERVAL` should guarantee that at least 16000 packets are simulated; this should ensure that the distance measurement has converged to within 1%. For monotone convergence, this `TOL` value should ensure that the results are accurate to within 10%. The FNS results are for `Q=1`.

When SNS was run on a Sun SPARCstation 2, the execution times in hours were approximately (1.3, 1.3, 1.3, 1.3, 1.5, 2.8, 5.5, 11, 45) for applied loads of (0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8). The longer execution times for larger applied loads were due to simulating over greater intervals (e.g., 64000 for `A=.8`) to achieve `TOL=.03` convergence. The execution time is proportional to the product of the number of nodes in the network and the simulation interval. For the same number of nodes and the same simulation interval, the execution time increases slightly with applied load, since the simulation of a node takes longer proportional to the number of packets forwarded during a cycle.

The data is intended only as a sample of simulator output; this report is not concerned with simulator results, their accuracy, or their interpretation. If the reader wishes to use the programs presented in this report, the sample data may be used check for errors in copying the programs.

A	T_{SNS}	T_{FNS}	T_{ANS}
.01	85	85	85
.10	90	90	88
.20	97	97	97
.30	107	107	108
.40	117	117	121
.50	138	138	151
.60	166	166	191
.70	218	218	291
.80	327	331	> 1194
.90	675	—	—

The FNS results match the SNS results unless the applied load is very high. With dimension-order routing, the probability of blocking due to competition for an output is $O(\frac{1}{R})$, so large-radix meshes require little buffering. The `A=.8` ANS entry is listed as > 1194 because the simulator had not converged after 5 days of execution. After simulating an interval of 128000 time units, 1.6 million packets had been received, but the average throughput was only 78.8% of the bisection bandwidth. Only SNS was run for `A=.9`; at this high applied-load, the simulator took eight days to converge. Both the effect of buffering and the performance of adaptive routing [7] will be the topics of future reports.

8 Acknowledgements

The research described in this report was sponsored by the Defense Advanced Research Projects Agency. The author was supported by a National Defense Science and Engineering Graduate Fellowship. The author gratefully acknowledges the direction and assistance of his advisor, Dr. C.L. Seitz.

References

[1] Seitz CL: Chapter 5 "Multicomputers", pp. 131–200, in Hoare CAR (ed): *Developments in Concurrency and Communication.* Addison-Wesley, 1990.

[2] Seitz CL et al: §2.1 "The Mosaic Project", pp. 2–10, in Caltech Computer Science Technical Report: CS-TR-91-10 Semiannual Technical Report, 1991.

[3] Pertel MJ, Seitz CL: §4.9 "A Silicon Architecture for Adaptive Cut-Through Routing", pp. 17–19, in Caltech Computer Science Technical Report: CS-TR-90-14 Submicron Systems Architecture Project, 1990.

[4] Flaig CM: VLSI Mesh Routing Systems. Caltech Computer Science Technical Report: 5241:TR:87. 1987.

[5] Ngai JY: A Framework for Adaptive Routing in Multicomputer Networks. Caltech Computer Science Technical Report: CS-TR-89-09. 1989.

[6] Pertel MJ: Mesh Distance Formulae. Caltech Computer Science Technical Report: CS-TR-92-05. 1992.

[7] Pertel MJ: A Critique of Adaptive Routing. Caltech Computer Science Technical Report: (in preparation). 1992.