# Implementation and Performance of the GLOW Kiloprocessor Extensions to SCI on the Wisconsin Wind Tunnel

Stefanos Kaxiras and James R. Goodman
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St.
Madison, WI 53706
{kaxiras, goodman}@cs.wisc.edu

***Abstract—****The main obstacle for scalability to large numbers of processors for cache coherence multiprocessors is the high degree of sharing for some types of data. Programs that exhibit a high degree of sharing are expected to achieve modest speed-ups for the base Scalable Coherent Interface (SCI) protocol, particularly as the number of processors sharing the data grows large. The proposed GLOW Hierarchical Extensions to SCI are designed specifically to attack the bottleneck of widely shared data and enable scalability to thousands of processors. The GLOW extensions are based on building sharing trees that map well to the underlying topology of a system, thus taking advantage of physical locality. This is in contrast to the earlier proposal for Kiloprocessor Extensions (STEM). We have specified the GLOW extensions and simulated them in the Wisconsin Wind Tunnel (WWT) parallel discrete event simulator[17]. We used the k-ary n-cube topology as the basis for the simulations. The implementation decouples data from the directory information. Some of the directory information is maintained in agents (i.e. the GLOW-capable switches that route messages to different rings). The presence of the corresponding data in the agents is optional. We are currently studying several options that pertain to the distribution of data through the sharing tree, to the rollout algorithms and to the local storage of data. We give results from micro-benchmarking both GLOW and SCI. The read and write latencies we present demonstrate the scalability of the extensions.*

## 1 Introduction

One of the major obstacles to scalability to large numbers of nodes faced by most cache coherence protocols is the reading and writing of widely shared data. Although studies have suggested that this type of data does not appear in abundance in application programs, the operations involved are expensive and worse yet they get linearly more expensive as the size of the parallel computer increases. While a program running on a small multiprocessor spends only a fraction of its time accessing such data, access to this data may dominate execution time of the same program when run on a system with hundreds of nodes. The GLOW Hierarchical Extensions to SCI, a proposed component of the IEEE P1596.2 Kiloprocessor Extensions to SCI, are intended to avoid this problem and provide scalability for the Scalable Coherent Interface [1].

Current trends in research for distributed shared-memory parallel machines, favour the use of specialized protocols for different classes of shared data [10]. It is believed that, by classifying data according to its use, a compiler can exploit the sharing patterns to improve performance. This technique, which is central to the success of software-based schemes, can also be exploited to enhance the performance of hardware-guaranteed cache coherence methods. Several classes of shared data have been identified, including migratory, read-only, *etc.* [10][11]. This approach to shared memory assumes that the programmer and/or the compiler can identify the type of a datum, generating code that conveys this information to the hardware, which can exploit the information to minimize unneeded network operations. This general concept has already been endorsed as part of the SCI standard, particularly in the specification of the QOLB synchronizing prefetch operation. The previously proposed Kiloprocessor Extensions to SCI, which we will refer to as the STEM Extensions[2], also assume compiler or user assistance in that a node must ask for a non-modifiable cache copy in order to permit the building of tree-structured lists.

The STEM extensions provide a logarithmic-time algorithm to build, maintain and invalidate a binary sharing tree without regard to the topology of the interconnection network. The complexity of the algorithm, however, is high, requiring complex transactions that generate significant traffic. Most importantly, this traffic is non-local and consumes large portions of system bandwidth. The complexity of the algorithm adds many costs, including both design and implementation complexity,

This work is supported in part by NSF Grant CCR-9207971, generous funding from the Apple Computer Advanced Technology Group, and donations from Thinking Machines Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618, with matching funding from the University of Wisconsin Graduate School.

and difficulty in performance evaluation. The STEM algorithm was proposed under the assumption that the latency of any message is unit latency (*i.e.* all messages have the same latency, regardless of the distance they travel). This is not an unrealistic assumption, because the average distance between randomly chosen nodes in most common topologies is not significantly less than the diameter of the topology.[1]

Combining [14] has been proposed to reduce the bandwidth requirements of the STEM algorithm and possibly capture some geographical locality. As employed in STEM, combining operates on two requests that happen to be in a queue at the same time, generating a single new request along with a response to one of the original requests. Combining depends heavily on traffic patterns, and in fact will rarely occur except in the presence of substantial congestion[3]. While this is the critical time for combining to occur, it is of course not guaranteed to occur even then. On the other hand GLOW extensions force the combining of requests in the switches(agents), consistently reducing traffic to the memory directories.

Large, high-performance systems are likely to be built on topologies where requests going towards memory–as well as data distributed back to nodes–follow traffic patterns that naturally form trees. STEM captures *temporal locality* in the tree: requesting nodes, in close proximity in *time*, end up neighbours in the tree. It is not possible, however, for STEM to capture *geographical locality*, because its structure is dependent on the timing of the requests. By geographical *locality* we mean that nodes in physical proximity become neighbours in the logical sharing tree, regardless of the timing of their requests. Such locality in the tree leads to protocols where messages only travel a short distance with very low latency, consuming little bandwidth.

The GLOW extensions capture the geographical locality of all topologies so far explored by mapping the logical sharing trees on top of the trees formed from the natural traffic patterns.

In this paper we report the implementation of GLOW Extensions in the Wisconsin Wind Tunnel (WWT)[17]. WWT is a parallel discrete event simulator that runs on a TMC CM-5. It executes application programs on the simulated machine with a very small slow-down compared to other simulation methods. This is because it executes target parallel programs at hardware speeds (without involvement) for the common case when there is a hit in the simulated cache. In the case of a miss the simulator takes control and takes the appropriate actions defined by the simulated protocol. The WWT keeps track of virtual time in processor cycles.

[1] In fact, the STEM algorithm is actually O($\lg^2$ N) since the distance the average message must travel is O(lg N).

# 2 Hierarchical Extensions to SCI

GLOW sharing trees are made of small, linear, SCI-like sharing lists. Each such list is confined to one physical ring. Although the trees are not optimal in the theoretical sense (some parts are linear) the latency of the operations involved–creation, rollout, invalidation–is low because messages are generally confined to a single ring.

GLOW extensions are based on a form of hierarchical caching. Requests from nodes in a particular ring will always be routed to a remote memory, through the same switch node within the ring. If this switch node contains directory information or a copy of the cache line, the request can be satisfied locally, without leaving the ring. Such a switch node that can intercept and satisfy requests is called a *GLOW agent*. In this paper we use the term *node* in the SCI sense. A node may contain either an SCI cache, or a GLOW agent, or both.

The requesting nodes will form a small (linear) SCI list under this agent instead of going all the way to the remote memory and joining a global (linear) list. This small linear list is called a *child-list* and the GLOW agent is its *parent*. In effect the agent will impersonate the remote memory–however far away it is–on the local ring. The agent itself, along with other nodes in the higher level ring (the next ring towards the home node), will in turn be serviced by yet a higher level node impersonating the remote memory on that ring. This recursive building of the sharing tree out of small child-lists continues up to the ring containing the true remote memory.

For the extensions to be general enough to cover many topologies, an agent can be the parent of several child-lists. In SCI the memory directory has one pointer because it can hold only one list. GLOW agents on the other hand, have a number of pointers so they can hold a child-list in any of the rings they are connected to. The number of pointers per cache line required by an agent is not specified but it is considered an implementation parameter, reflecting both topology and cost considerations. The sharing trees that result depend on the topology used and they can be as rich as the topology or as poor as the available budget. In contrast, STEM defines three pointers per cache line, restricting the trees to binary trees. In our implementation we have a forward and a backward pointer for the agent to be able to join lists and a number of child pointers so the agent can hold its child-lists. The number of child pointers one less than the number of rings to which the agent is connected, since an agent never impersonates memory for all the rings to which it is connected.

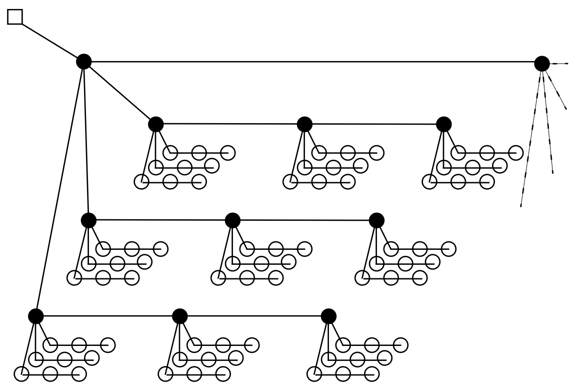

**FIGURE 1. Logical k-ary tree of lists**

An example of a general *k*-ary tree that can be formed this way is shown in Figure 1. The tree is drawn in such a way as to stress the lists as building blocks. In this figure memory is represented by the square in the top left corner, SCI caches by white circles and GLOW agents by black circles. The horizontal strings of SCI caches are the linear lists. Several of the GLOW agents in the above example have three child-lists. As we noted before a linear list and its agent parent are contained in one physical ring.

We say that all nodes in a list belong to a level of hierarchy. The nodes of the list that is connected to memory, are in the highest level of hierarchy. The nodes that cannot have children are in the lowest level. The nodes of the child-lists of an agent in level *L* are in level *L-1* or lower. In Figure , the tree has three levels of hierarchy and the lists in the lowest level are shown to have three members.

In summary the main points of the proposed GLOW Extensions are:

- Data that is widely shared is accessed with special requests that result in creation of sharing trees instead of the normal SCI lists.
- The logical *k*-ary sharing tree is mapped directly onto the underlying topology
- The *k*-ary sharing tree is made of small (linear) SCI-like lists. A (linear) list is confined inside a single ring.
- The nodes that are members of the same (linear) list have a common agent on the ring that by default routes their requests to other rings.
- Such agents have directory information and optionally a copy of the data. The directory information (the child pointers) is used to satisfy requests locally.

# 3 An Example Topology

GLOW Extensions can be implemented on top of a wide range of topologies, including hypercubes; meshes; trees, snowflakes, dense snowflakes and star topologies [4]; the lens [5]; and many others. We chose to implement and study the GLOW extensions on a popular topology we believe is a likely candidate for implementation. It is the *k*-ary *n*-cube topology made of rings, suggested in the SCI standard (p.12). We stress here that the GLOW extensions are not based on this topology and can be equally well applied to other topologies. As an example we have shown [12] how they are applied in a topology called Omega/Flip[7]. This topology is a refinement of the butterfly topologies originally proposed by Ross Johnson [2] and also suggested in the SCI standard (p.11). A disadvantage of the original butterfly topologies is that the rings have a varying number of nodes. The Omega/Flip is an improvement and its important characteristic is that it is built of rings of the *same size*.

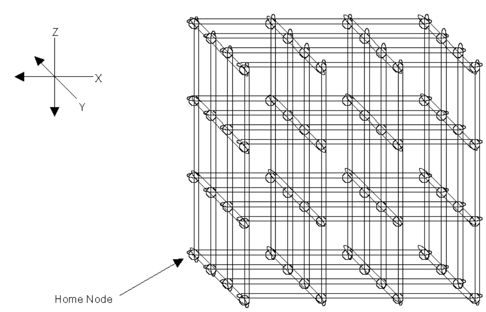


**FIGURE 2. 4-ary 3-cube**

## 3.1 *K*-ary *n*-cube

In this example the routing of requests is fixed in dimension order to have deadlock free routing [6]. For example, in a *k*-ary 3-cube the request messages are routed first along *x*, then *y* and finally *z* dimensions (x being the lowest dimension and z the highest). Response messages are routed in reverse order. At the place where

GLOW requests switch to the ring of the next dimension they are intercepted by the agent and processed. A new GLOW request is generated and continues in the next dimension. The response to an intercepted GLOW request travels back one dimension to the ring the request used. Notice that normal SCI requests are unaffected and proceed full speed through the agent.

In Figure 2 we show a 4-ary 3-cube which is equivalent to a 3-D Torus. This topology has four nodes per ring and three rings per node. The total number of nodes is 64. Without loss of generality we assume that all the nodes access a cache line in some node, labelled "Home node."

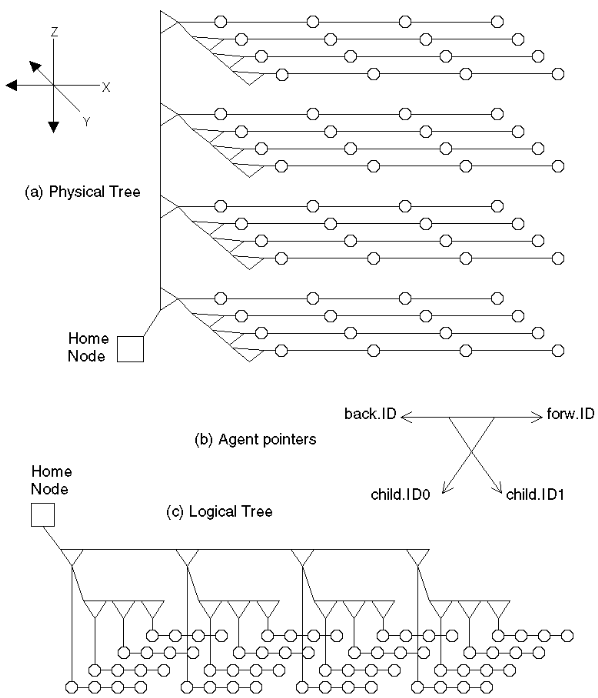


**FIGURE 3. Tree on 4-ary 3-cube**

In the complete sharing trees of this topology, not all nodes have the same number of child lists. The higher the hierarchical level of a node, the more child lists it has. For this example, nodes in the lowest level have none, in the next level they have one, and in the top level they have two. In general, in a $k$-ary $n$-cube, the logical tree will have $n$ hierarchical levels. At the highest level (the list that is directly connected to memory) the nodes will have $n$-$1$ child-lists. The child-lists do not necessarily belong to the same hierarchical levels. Moving one level down, the number of child-lists decreases by one, until the lowest level, where the nodes do not have any child-lists.

In Figure 3 we show how a $k$-ary tree is mapped on the 4-ary 3-cube. The GLOW agents are represented by triangles and the SCI caches by circles. Figure 3 (a) is the physical representation of the tree. In the second representation of the tree Figure 3 (c), we draw the logical representation as a tree of lists. In Figure 3 the lists are drawn in perfect order. This is not the usual case where the list order is the reverse order of the arrival of the requests to the agent. Figure 3 (b) shows the pointers supported by the GLOW agents (see Section 4).

# 4 GLOW Agents

The agents play a central role in our schemes. They intercept requests not addressed to them and cache tree tags and possibly data. The agents can be stand-alone on a ring or part of another SCI node. We have studied the second case as it is likely that agents will be implemented as part of the ring interface card on a workstation. Our view of this case is depicted in Figure 4 where we show a GLOW-capable workstation SCI node.

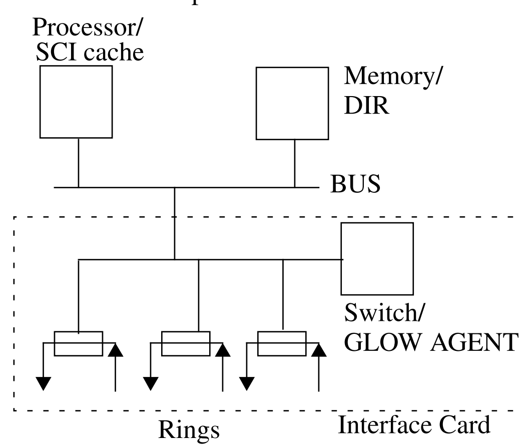


**FIGURE 4. GLOW Agent/SCI Node**

Additional functionality is required for an SCI switch to service a GLOW agent. Specifically the switch must be able to:

1. Intercept special GLOW requests used for the creation of the trees. The switch must recognize these requests by examining the command field (second symbol) of a request. Instead of routing these requests as with other SCI requests, switching them from one ring to another, the switch delivers them to the GLOW agent.
2. Annotate an intercepted special GLOW request with the ring ID whence it came. In this way the GLOW agent will be able to update the appropriate child-pointers.

# 5 Protocols

## 5.1 Construction

In this section, we describe the construction of the *k*-ary tree. A node that wishes to read a remote cache line that is expected to be widely shared, sends a special *mread_tree* request towards the node that has the data in its local memory (home node). The request is intercepted at the first agent (where it would change rings), *at the agent's discretion*.

In the k-ary n-cube topology, conceptually all messages are injected in the lowest dimension ring first. If the message does not have to travel at all in the lowest dimension the first agent is encounters is the agent that resides in the same node.

The intercept causes a lookup to find information about the requested line. The lookup may result in a *hit* or a *miss*:

- If the lookup results in a hit, the requesting node is instructed to prepend to the appropriate child list. The node will either get the data from the previous head of the child-list or the agent itself if it keeps copies of the cache lines. If the node is first in a new child-list and the agent does not have the data it is the responsibility of the agent to *fetch* the data from a node on one of its other child lists.
- If the lookup results in a miss, the agent sends its own request for the line towards the home node. The agent does not forward intercepted requests but rather replaces the first of them by its own request. The requesting node is instructed to prepend to the appropriate child list. As soon as the agent gets a copy of the line it will pass it to the waiting node(s).

In the time it takes for an agent to get the data, new requests may arrive. Each of the new requesting nodes is instructed to prepend to the appropriate list and wait for the data. Three options have been identified for distributing the data to a list where all nodes are waiting for the agent to get the line:

1. *Tail to head*: As is currently defined in SCI. A node asks the agent (memory in SCI) for the data. The agent (or memory in SCI) instructs the node to prepend to the child-list and request the data from the old head. The very first node to request the data becomes the waiting tail of the child-list. It will eventually get the data from the agent. The agent may keep a temporary pointer to the first requesting node (the memory allocated for the optional data storage can be used for temporary pointers). Note this scheme requires the agent to maintain two pointers: one to the tail for providing the data, and one to the head for adding additional nodes.
2. *Head to Tail*: If a requesting node prepends to a child-list, where the old head is waiting for the data, then it assumes responsibility to forward the data. When the agent gets the data it will pass it to the waiting heads of its child-lists. The heads will then forward the data towards the waiting tails. This distribution happens in the opposite direction of the one described above (standard SCI). The benefit of this scheme is that the agent need only keep a single pointer for each child-list.
3. *Broadcast on the ring*: This is a broadcast confined in a single ring. All nodes that are waiting will receive the broadcast and consequently the data. To guarantee correct operation, it may be necessary to extend the extant SCI broadcast to provide a confirmation protocol to ascertain that all waiting nodes have actually received the data.

In the case of a miss in the agent, its own request will be treated the same way in the next agent. Eventually the last agent will get the data from memory and pass it to its child-lists.

When all nodes simultaneously read a cache line (for example, after a global barrier), construction of the tree is quite fast: All the linear lists can be constructed in parallel. Copies of the cache line are distributed down the tree concurrently with list construction.

As mentioned before the agent may choose to ignore a request completely. This does not affect the correctness of the protocols at all, but it may have a negative impact on performance. When an agent ignores a request the request is passed to the next hierarchical level where hopefully it will be serviced by the higher level agent. In this case the node that issued the original request joins a linear list in a higher hierarchical level than normally. The request may be ignored by multiple agents all the way to the home node where eventually it will be serviced by the memory directory. The agents may ignore requests to avoid deadlocks due to storage conflicts. Specifically, a request is always ignored when it requires storage for a tree tag that is currently taken by another tree tag in a waiting state (not yet part of a stable tree).

## 5.2 Rollout

Because of conflicts both SCI cache and GLOW agents need to be able to leave the sharing tree. Also, a node has to leave tree in order to be able to write the cache-line. This operation is called Rollout in the SCI terminology.

Rollout for SCI caches in a GLOW sharing tree is the same as SCI. The SCI cache first changes the *backward* pointer of its downstream neighbour to point to the upstream neighbour and then it changes the *forward* pointer of its upstream neighbour to point to the downstream neighbour. If the cache is the last (only) node in a child list it rolls out from its parent (which may be a GLOW agent or the memory). The cache also returns the data to its parent because it may be the last node in the tree to have a valid copy of the line. Notice that this is what happens in SCI when the last node of a list rolls out and writes the data back to memory.

Childless agents are not allowed in the tree. As soon as the last child rolls out the agent rolls out too. When an agent finds itself childless it is only connected to the tree with its *forward* and *backward* pointers. In this case the rollout is the same as for standard SCI caches. The agent too has to pass the data back to its parent if it is the only node in its list.

If the parent has allowed other nodes to prepend while a rollout is taking place it notifies the rolling-out node about the imminent prepend with a negative acknowledgement, just as memory notifies a node in SCI. The rollong-out node waits for the prepend, after which it informs the prepending node that it is no longer the head of the list. The prepending node repeats its request to the new head. While in SCI the memory is only responsible to notify rolling out nodes about prepends in GLOW the parent also notifies the node about imminent invalidation with a negative acknowledgement (see Section 5.3).

An agent rolls out because of conflicts in its directory (or data) storage. In this case one additional step precedes the rollout algorithm describe above: The agent first deals with its child-lists, then rolls out from its downstream neighbour and finally from its upstream neighbour (or parent).

Subsequently we describe the two approaches we are considering for dealing with the child-lists. When the first is used the rollout is called *Destructive* because it destroys parts of the tree. When the second is used the rollout is called *Linearizing* because it connects the child-lists into big linear lists. Additional details about the cases we are considering are elsewhere [12][13].

### 5.2.1 Destructive Rollout

The simple solution, borrowed from hierarchical caches, is to maintain multilevel inclusion [9]. When a node rolls out it invalidates all of its descendents using the invalidation algorithm described in Section 5.3. When the invalidation completes, the agent is childless and it can rollout from its downstream and upstream neighbours as usual. This simple scheme has great appeal because of its simplicity. However, it might lead to thrashing behaviour, constantly invalidating nodes which then immediately repeat their requests for the cache line. Part of our future work is to assess the behaviour of GLOW agents in this respect and in relation to the amount of storage allocated for directory information.

In the WWT implementation we have used this scheme first because of its simplicity. However the GLOW directory storage is sufficiently big so that thrashing behaviour does not occur for our simple benchmarks.

### 5.2.2 Linearizing Rollout

This scheme gives priority to preserving the sharing tree, permitting the structure of the tree to degrade gracefully.

The Linearizing rollout is based on chaining the child-lists and subsequently substituting the agent with the chained child-lists in the tree. There is more than one way to chain child-lists together[13]. Here we present only the primary candidate. The child-lists are concatenated, *i.e.*, they are chained tail-to-head into one big linear list. The head of the first child-list connects to the tree in the position of the agent's backid pointer. The tail of the last list connects to the tree in the position of the agent's backid pointer. The process is shown graphically in Figure 5.

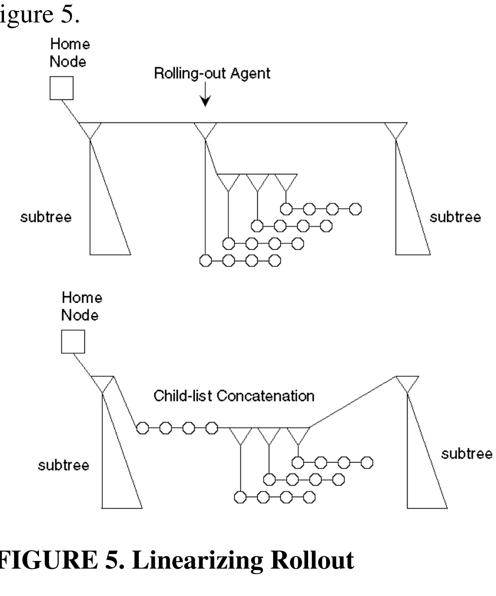


**FIGURE 5. Linearizing Rollout**

In the Linearizing rollout protocol, an agent rolling out and later rejoining the tree creates problems. In this situation the relation of the agent and its children is lost at the point of the rollout. If the agent rejoins the tree it will do so in another position in the linear list of the appropriate hierarchical level. New requesting nodes that would otherwise become heads in the old child-lists, will create new child-lists under the agent.

### 5.3 Invalidation

Invalidation–destruction of the tree–is defined similarly to STEM. When a node wants to write a line, it first becomes the head of the top level list. If the node was in a sharing tree it first has to rollout and then send a request directly to the home node to become the root of the tree (the node that is connected directly to the memory directory). Invalidation messages are then forwarded down the tree in parallel. When the invalidation messages reach the tails of the lists the tails invalidate themselves and send acknowledgment back. GLOW agents wait for the acknowledgment of all the invalidation messages they forwarded, invalidate themselves, and return their own acknowledgment. See references [13][2] and [13]for further details.

In a well-structured tree, invalidation is very fast because all the messages are exchanged locally–between nodes within the same ring. Even if the longest invalidation path through a GLOW tree is longer than that of the STEM tree, invalidation in many cases is faster because of the low message latency.

We note here that, for a scheme that can exploit broadcast effectively, confining lists to a single ring potentially offers very significant reductions both in message traffic and destruction latency.

## 6 Implementation of GLOW in the WWT

As mentioned before the GLOW extensions where implemented on the WWT and in particular as extensions to the WWT SCI [15]. So far we have only implemented the Destructive Rollout algorithm. The Linearizing Rollout is planned to be implemented next. There is support for the GLOW data storage option so the GLOW agents can keep copies of the cache-lines they handle. However in this study we have not yet exercised this option. To date we have implemented the first two data distribution schemes, *Tail-to-Head* and *Head-to-Tail*. The third scheme is more complex because it requires broadcast support.

### 6.1 The local cache optimization.

Furthermore we have implemented an optimization we call the *local cache optimization*. In this optimization the cache that resides in the same node as the agent does not have to join a child-list with other nodes in the lowest level ring as was described Section 5. Instead the agent has an extra one bit pointer that points to the local-cache. In effect it is as if the agent and the local cache are connected inside the node with a special 2-node ring.

With this optimization its not necessary for the local cache to communicate with GLOW agents and SCI caches outside the node to participate in GLOW operations. The changes in the protocols are minimal because the local cache is handled as an extra "node" on an extra ring connected to the agent. The complete sharing tree in the *k*-ary *n*-cube with the local cache optimization is shown in Figure 6. The GLOW agents are represented by triangles and the SCI caches by circles. In this figure we show again the physical (a) as well as the logical (c) representation of the tree as well pointers that the agent supports (b).

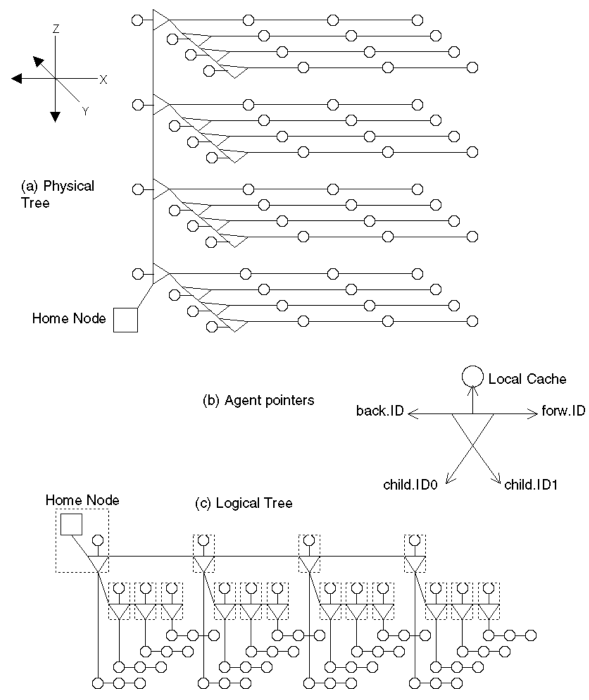


**FIGURE 6. Local Cache Optimization Trees**

## 6.2 WWT Node

In this section we describe in detail the parameters we have used for the nodes and agents.

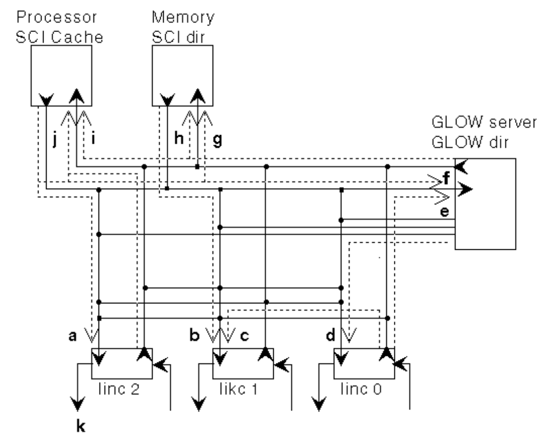


**FIGURE 7. Timing diagram of a WWT node**

Figure 7 shows a block diagram of a WWT node. The node comprises a processor, an SCI cache, memory, memory directory, a GLOW agent, and a number of ring interfaces (three shown for a *k*-ary *3*-cube). The connections between the interfaces represent the switch.

In our simulations we assume a 200MHz processor with a sustained IPC of one, and a 250MHz clock for the SCI network. These values where chosen to duplicate the simulation parameters used elsewhere [15] and [16].

The latencies in the node are defined with respect to network cycles (Net) and then scaled to processor cycles (P). All WWT results are reported in processor cycles(P). In the following tables we give the latencies for the data paths inside a WWT node, the access latencies of the caches, directories and memory and the latencies in the SCI *linc* interface. In Table 1 and Table 3 the leftmost column contains the labels of Figures 7 and 8 respectively.

### Table 1: Path latencies

| Fig. 7 | Path | Net | P |
|---|---|---|---|
| **a,b** | Processor/Cache/Memory/Dir to Queue | 18c | 14c |
| **g,j** | Queue to Processor/Cache/ Memory/Dir | 18c | 14c |
| **d** | GLOW to Queue | 5c | 4c |
| **e** | Queue to GLOW | 5c | 4c |
| **i,h,f** | Local (BUS) | (13c) | 10c |
| **c** | Switch Delay | 18 | 14 |

### Table 1: Path latencies

### Table 2: Access latencies

| Type | P |
|---|---|
| Cache Tag access | 3c |
| GLOW Tag access | 3c |
| Memory Dir access | 10c |
| Cache Read/Write Data | 16c |
| GLOW Read/Write internal Data buffers | 16c |
| Send Message | 5c |

### Table 3: SCI linc latencies

| Fig. 8 | Type | Net | P |
|---|---|---|---|
| **a** | Channel to Bypass to Channel | 9c | 7c |
| **c** | Queue to Channel | 3 | 2c |
| **b** | Channel to Queue | 6 | 4c |
| **d** | Channel to Channel | 1c | 1c |

The latency of a message is calculated using the above parameters and multiplying the appropriate parameters with the number of lincs and number of switches on the message's path.

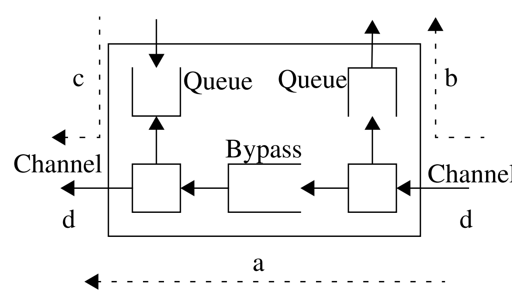


**FIGURE 8. SCI linc interface**

# 7 Simulations and Results

We have simulated very simple programs to micro-benchmark SCI and GLOW and get a feel of the behaviour of both when it comes down to individual reads and writes. We are currently debugging the simulator and preparing to run kernels and real programs.

The simulations are done for topologies indicated in the following table:

**Table 4: Topologies**

| Nodes | 16 | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|
| Dimensions. | 2 | 2,3 | 2,3 | 2,3,4 | 2,3,4 | 2,3,4 |

## 7.1 Reads

To measure read requests we have simulated a program where a number of nodes read a shared variable. All the nodes that participate in the read start together after initialization. The number of nodes that participate varies from all the nodes in the topology to 1/64 of the nodes. The various degrees of sharing we have simulated are 100%, 50%(1/2), 25%(1/4), 12.5%(1/8), 6.25%(1/16), 3.125%(1/32), 1.5625%(1/64).

### 7.1.1 FRESH Data

Figures 10 through 12 present the results for the read requests when memory is in state FRESH. In this case, where memory supplies the data immediately, the nodes get the data before they attach to the list and therefore the latencies are low. In GLOW the same holds for the nodes that are in the highest level, connected to memory.

Each figure represents a different topology. The x axis is the number of nodes in the topology that actually read the data. The y axis is the average latency that the readers see in every case. In the figures we show the SCI and the GLOW latencies. The curve labels give the number of nodes in the topology and its dimension.

For very small numbers of readers the average GLOW latency heavily depends on the nodes that where chosen to read the data. The first three figures (9 to 11) are averages of three runs, each run with different nodes sharing. The dip in the latency we see for two nodes is because for several cases one of the nodes is the other's agent. For larger number of nodes reading the average latencies do not depend on which nodes are reading.

The graphs show that, as expected, GLOW has considerable overhead for very small topologies and for very low degree of sharing. For 16 nodes and FRESH data GLOW is slower than SCI for all degrees of sharing. For 32 node topologies only the 3-Dimension topology has lower latencies GLOW and only in the case where all nodes are sharing the data. For 64 and 128 node topologies we have a threshold of around 25 nodes where GLOW starts to have lower latencies than SCI. For 256 and 512 node topologies the threshold is slightly higher.

In all the figures SCI latencies increase linearly with the number of nodes that are reading the data. This is because memory services the requests serially one after the other. In GLOW the burden of servicing read requests is distributed to multiple agents that operate in parallel. The memory directory has to service only the highest level agents.

### 7.1.2 GONE Data

Figures 13 through 16 present the results for the read requests when memory is in state GONE. In this case memory does not supply the data. In the SCI protocol nodes attach to the head of the list and get the data there. The latency that the nodes see is approximately the latency of all the previous nodes in the list to get the data. In GLOW this affects only the agents and nodes in the highest level list. As in the case of FRESH data, the nodes in lower level lists get the data as soon as their agents can supply them.

Again each figure represents a different topology. The x axis is the number of nodes in the topology that actually read the data. The y axis is the average latency that the readers see in every case. In the figures we show the SCI and the GLOW latencies. In general the SCI latencies of all the figures are approximately the latency of the data propagating through the list. GLOW latencies on the other hand are kept quite low because, as we mentioned above the delays are accounted for in the nodes of the highest level list. The threshold for GLOW having lower average latencies than SCI is around four for all topologies. These figures show the excellent scalability of the GLOW protocol to large numbers of processors.

## 7.2 Writes

To measure latencies for writes when the writer has to invalidate the list in SCI and the sharing tree in GLOW we used the following simple loop:

```
void node_code(){
  for(i=0;i<16;i++){
    Barrier();
    if (I am the current writer)
      write shared variable;
    Barrier();
    if (I belong to the readers)
      read shared variable;
}}
```

This loop allow us to control how big a tree is invalidated. In each iteration a different writer writes the data. The writers are spread evenly across the topology. After the second barrier a number of readers read the data.

In figures 17 through 20 we show the average latency of a write. This latency has three components: the latency for the node to rollout from the tree, the latency to go to memory and subsequently become the root of the tree, and the latency to invalidate the rest of the tree. Each of the figures 17 through 20 represents a different topology. The x axis is the number of nodes that read the shared data in each iterarttion of the loop. The y axis is the average latency of the 16 different writer. Each graph shows the SCI and GLOW write latencies for various degrees of sharing.

The threshold when GLOW begins to have lower latency than SCI is quite low, around four nodes, the same as in the case of reading data in memory state GONE. The graphs also show the excellent scalability of GLOW to large numbers of processors. Lastly in figures 21 through 24 we show the speed up of GLOW for each topology. The speed-ups are calculated by dividing the SCI execution time (in WWT virtual cycles) by the corresponding GLOW execution time. For very small topologies or for very low degree of sharing GLOW is slower than SCI, but for large topologies and medium to high degrees of sharing GLOW can be 40 times as fast as SCI for this simple loop.

# 8 Future work and directions

## 8.1 GLOWing STEM

We are examining ways to integrate STEM and GLOW. The *GLOWing STEM* protocol uses GLOW's forced combining to initially form trees that exhibit geographical locality. By using the Linearizing rollout, the GLOW tree degrades gracefully into linear lists. At this point the STEM protocol takes over and converts these linear lists to binary trees. Notice that the final STEM tree will also exhibit a high degree of geographical locality. In effect we borrow GLOW's guaranteed combining and its ability to capture geographical locality and we use them to produce the desirable linear lists for STEM. A last point here is that the STEM algorithm may start working on many parts of the GLOW tree that have degraded to linear lists in parallel, i.e. it is not required for STEM to wait until the GLOW tree has degraded completely to a global linear list.

## 8.2 Pruning cache approach

We are also investigating a more radical departure from the base SCI protocol that offers higher performance. The scheme depends on the observation that entire lists created under the proposed scheme are generally contained within a single ring. This fact, and the expectation that a modest number of nodes will be attached to a single ring, suggests the possibility of storing directory information related to a single list not in the form of doubly-linked list, but as a bitmap. Storing the information as a bitmap opens up a range of alternative algorithms, many using broadcast, and in particular, permits a pruning cache implementation in the style of that proposed by Scott and Goodman [8]. We note that the SCI transport layer provides a limited broadcast mechanism that is not used by the coherence layer.

### 8.2.1 Bitmaps and Rollout

The bitmap scheme mentioned earlier could be used to store in an agent all the siblings for each of its child-lists. The pruning cache concept is based on the observation that if the bitmap information is lost, the tree can be reconstructed conservatively, that is, by including all possible siblings in the child-list. While nodes not in the tree may therefore be inadvertently inserted into the tree, they need not contain data, so no harm is created by adding them (except that the tree is unnecessarily larger than it need be, and extra traffic may be generated in destroying it).

One of the benefits of the bitmap scheme is that a node can be rolled out for free. Note that this feature meets a basic requirement of the bitmapped scheme: all lists must be contained within a single ring. This scheme therefore always forms the perfect tree, and avoids the problems of patching ill-formed trees.

## 8.3 Sparse Trees

Sparse trees are trees constructed without invoking every agent on the way to memory thus minimizing the number of superfluous nodes in the tree. Restructuring of the trees is very complex and we do not have a solution to build the tree top-down. We do have, however, a proposal to create the tree slowly, bottom-up.

For sparse tree construction it is not necessary to invoke all the agent nodes on the way to memory. Instead a requesting node will invoke only the agent at the next hierarchical level. If this agent accepts a number of additional requests, it will invoke the next agent and so on. When an agent is first invoked by one of its children it will send a new *mread* request to memory. This request, in contrast to the *mread_tree* request, is not intercepted by other agents. When an agent decides to invoke its own parent agent it connects to it and forces it to join the tree if it has not already. After the new parent agent joins the tree the requesting agent disconnects from its current position (with a rollout) and rejoins in its correct position under its parent agent. By changing only the *forwid* and *backid* and never the child pointers of the nodes, the tree restructures and eventually it will assume the structure of a well formed GLOW tree.

## 8.4 StoreUpdate

### 8.4.1 StoreUpdate by the Head of Tree

StoreUpdate in the tree, is defined similarly to STEM. A node wishing to update all other entries in the sharing tree, leaves the tree (performing a rollout as described in the previous section). It then prepends to the top level list. Being the head, directly connected to memory, it can initiate a StoreUpdate. The StoreUpdate happens in two phases. The first is the initiation phase where the new data propagate throughout the tree in the same way as the invalidation messages. A node receiving a StoreUpdate from its parent or left sibling forwards it in parallel to all its children and right sibling. Alternatively a childless node can simply return its *forwid* to its parent, giving it the responsibility to forward the StoreUpdate to the next node. When the data reach the tails of the lowest level lists, the tails return confirmation of their update. Nodes in the tree wait for confirmation from all their children and right sibling, before they return their own confirmation messages to their parent.

### 8.4.2 StoreUpdate with Bimodal Behavior

A method of doing StoreUpdate that is currently discussed in Kiloprocessor Extensions to SCI working group is for a node to have bimodal (or schizophrenic) behavior. This behavior allows a node to be in its proper position in the tree and also temporarily be head in the highest level list. A node in this position has exclusive access to the tree and can perform a StoreUpdate as described above.

# 9 Acknowledgements

We wish to thank David V. James, Ross E. Johnson and Stein Gjessing for their suggestions, contributions and constructive comments on this work. We also thank many other members of the working group for their suggestions and their encouragement. Many thanks to Douglas C. Burger and Alain Kagi who helped integrate the GLOW extensions to the WWT network simulator and WWT SCI, respectively.

# 10 References

[1] IEEE Standard for Scalable Coherent Interface (SCI) 1596-1992, IEEE 1993.

[2] Ross E. Johnson. "Extending the Scalable Coherent Interface for large-Scale Shard-Memory Multiprocessors," PhD Thesis, University of Wisconsin-Madison, 1993.

[3] Gregory F. Pfister and V. Alan Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks." *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 790-797, August 20-23 1985.

[4] Marvin H. Solomon and Raphael A. Finkel, "Processor interconnection strategies," *IEEE Transactions on Computers*, Vol. **C-29**, no **6**,pp. 360-371, May 1980.

[5] Raphael A. Finkel and Marvin H. Solomon, "The lens interconnection strategy," *IEEE Transactions on Computers*, Vol. **C-30**, no. **12**, December 1981.

[6] William J. Dally and Charles L. Seitz, "Deadlock-Free Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers*, Vol. **C-36**, no. **5**, pp. 547-553, May 1987.

[7] Duncan H. Lawrie, "Access and Alignment of Data in an Array Processor." *IEEE Transactions on Computers*, Vol. **C-24**, no. **12**, pp. 1145-1155, December 1975.

[8] Steve L. Scott and James R. Goodman, "Performance of Pruning-Cache Directories for Large-Scale Multiprocessors," *IEEE Transaction on Parallel and Distributed Systems*, Vol. **4**, no. **5**, May 1993.

[9] J. L. Baer and W. H. Wang, "Architectural choices for multilevel cache hierarchies," *Prod. 16th International Conference on Parallel Processing*, pp. 258-261, 1987.

[10] John Carter, John Bennet and Willy Zwaenopoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proceedings of the Conference on the Principles and Practices of Parallel Programming*, 1990.

[11] Wolf-Dietrich Weber and Anoop Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors," *ASPLOS III*, pp. 243-256, 1989.

[12] James R. Goodman and Stefanos Kaxiras, "Hierarchical Extensions to SCI," University of Wisconsin-Madison, C.S. Dept., Technical Report 1235, July 1994. (ftp.cs.wisc.edu)

[13] Stefanos Kaxiras and James R. Goodman, "Hierarchical Extensions to Cache Coherence in SCI," *Proceedings of the 1st International Workshop on SCI-Based High-Performance Low-Cost Computing,* August 1994

[14] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, M. Snir."The NYU Ultracomputer-Designing a MIMD Shared Memory Parallel Computer; *IEEE Trans. on Computers*, Vol. **C-32**, no **2**, pp. 175-189, Feb. 1983.

[15] Alain Kagi, D.C. Burger and J.R.Goodman., "Techniques for Reducing the Overhead of Shared-Memory Multiprocessing," University of Wisconsin-Madison, C.S. Dept., Technical Report 1266, February 1995.

[16] Douglas C. Burger and James R. Goodman. "Parallel Simulation of the Scalable Coherent Interface Transport Layer,", University of Wisconsin-Madison, C.S. Dept., Technical Report 1265, February 1995.

[17] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers." In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems*, pages 48–60, May 1993

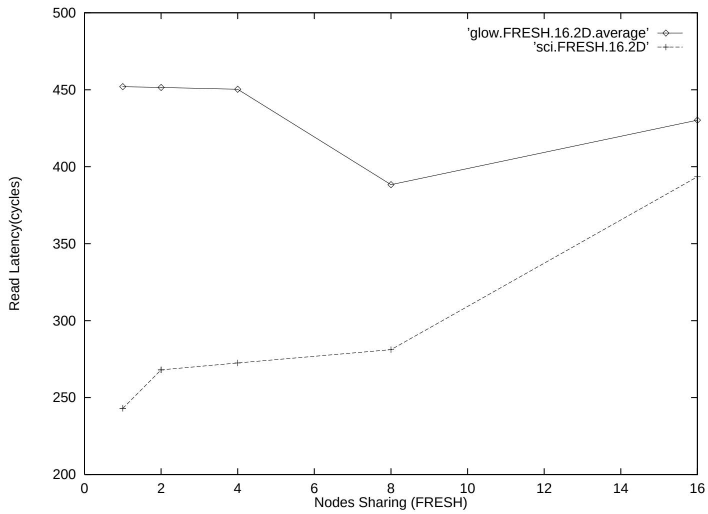


**FIGURE 9. 16 Nodes (2-D), memory FRESH**

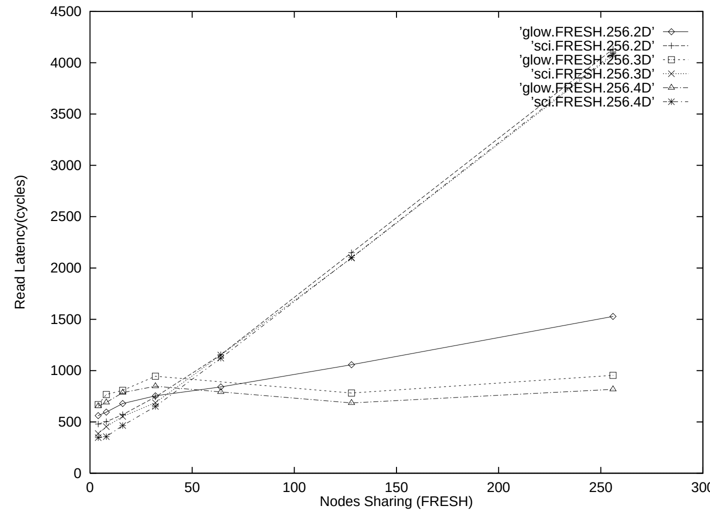


**FIGURE 12. 256 Nodes (2-,3-,4-D), memory FRESH**

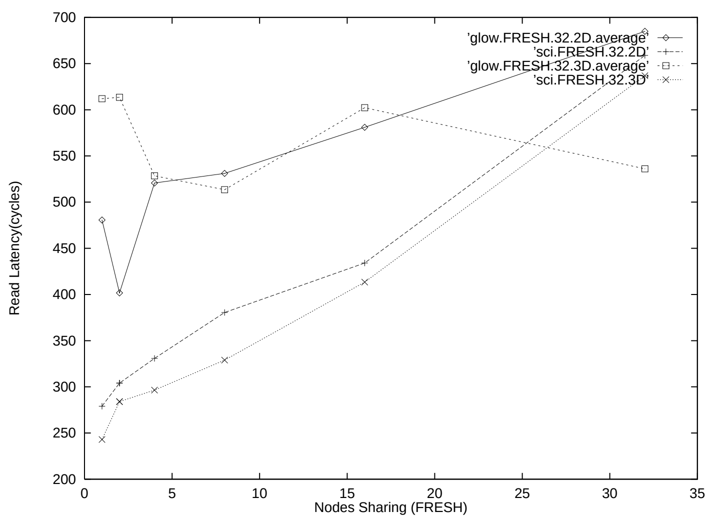


**FIGURE 10. 32 Nodes (2-,3-D), memory FRESH**

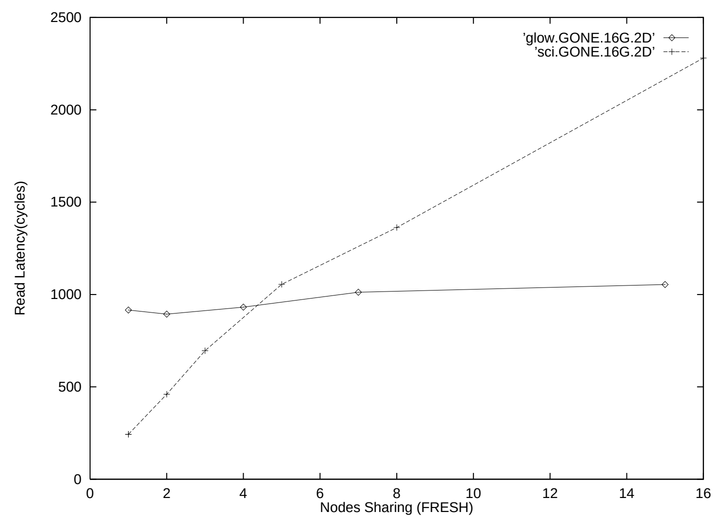


**FIGURE 13. 16 Nodes (2-D), memory GONE**

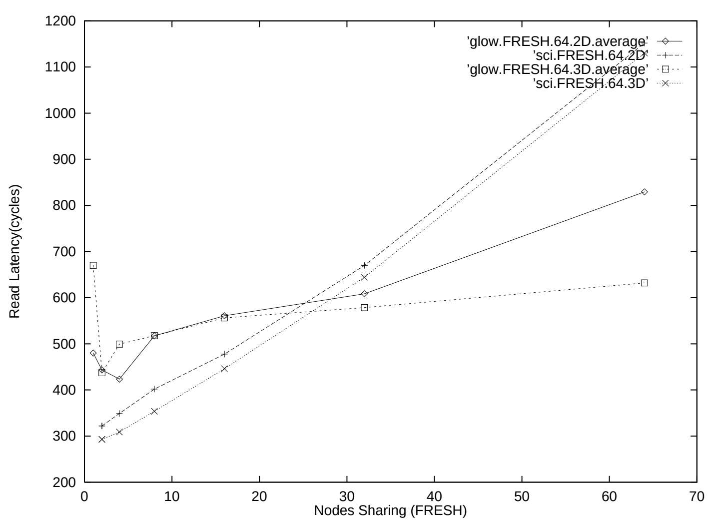


**FIGURE 11. 64 Nodes (2-,3-D), memory FRESH**

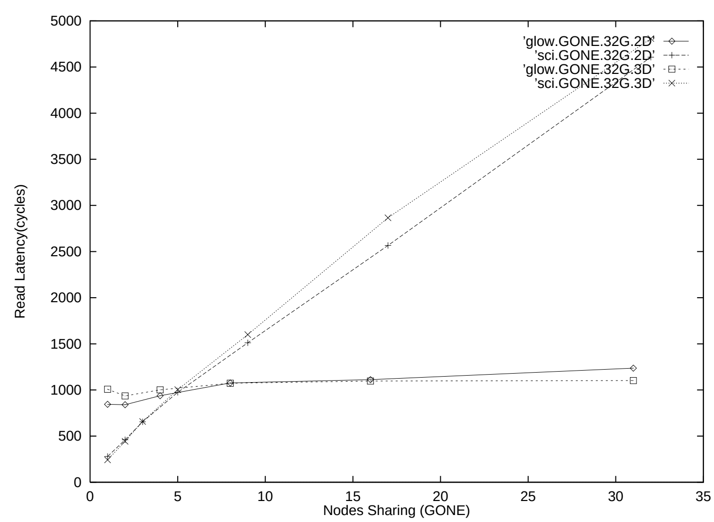


**FIGURE 14. 32 Nodes (2-.3-D), memory GONE**

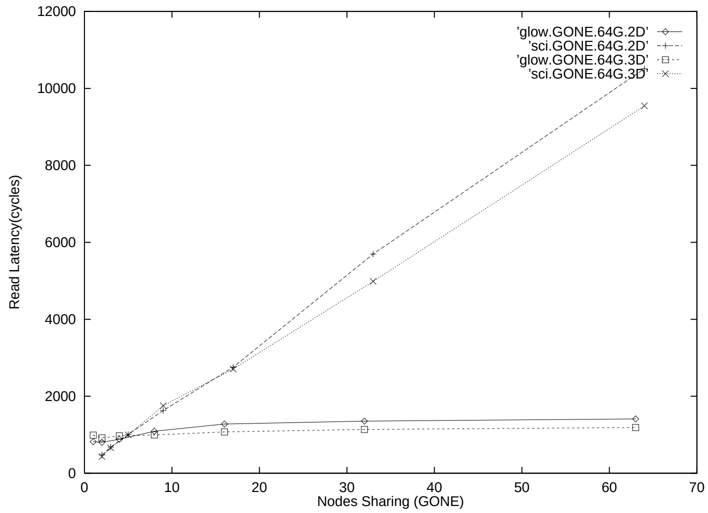


FIGURE 15. 64 Nodes (2-,3-D), memory GONE

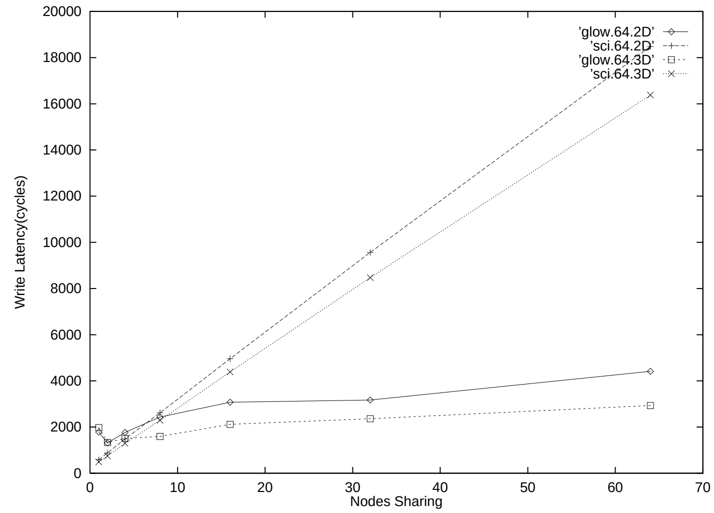


FIGURE 18. Writes in 64 Nodes (2-,3-D)

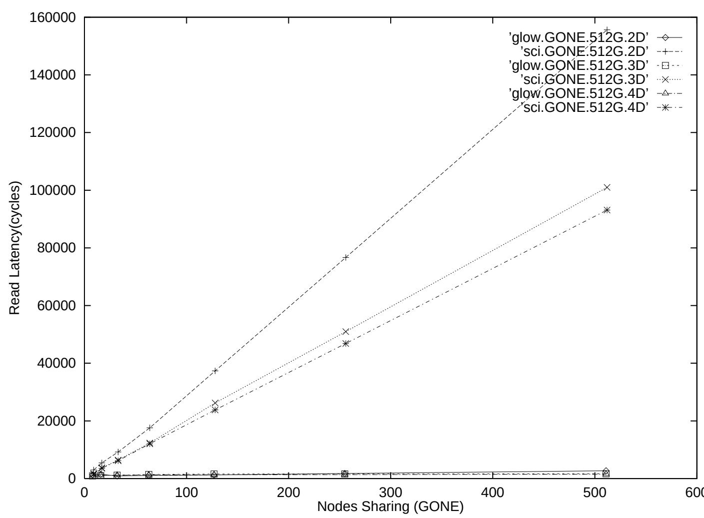


FIGURE 16. 512 Nodes (2-,3-,4-D), memory GONE

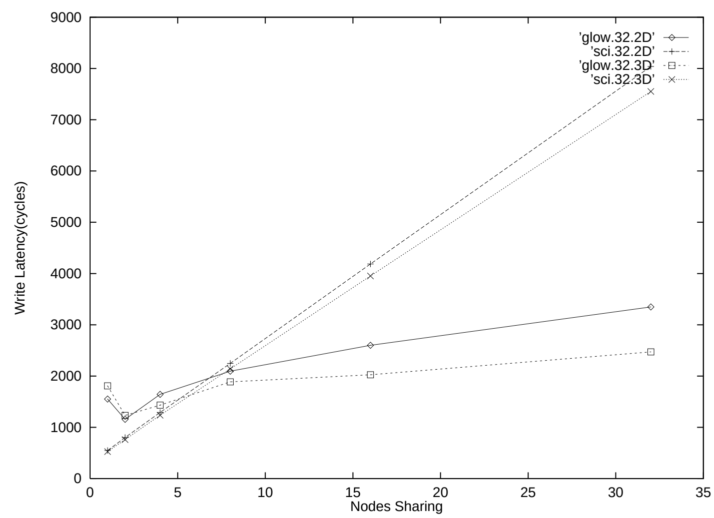


FIGURE 19. Writes in 32 Nodes (2-,3-D)

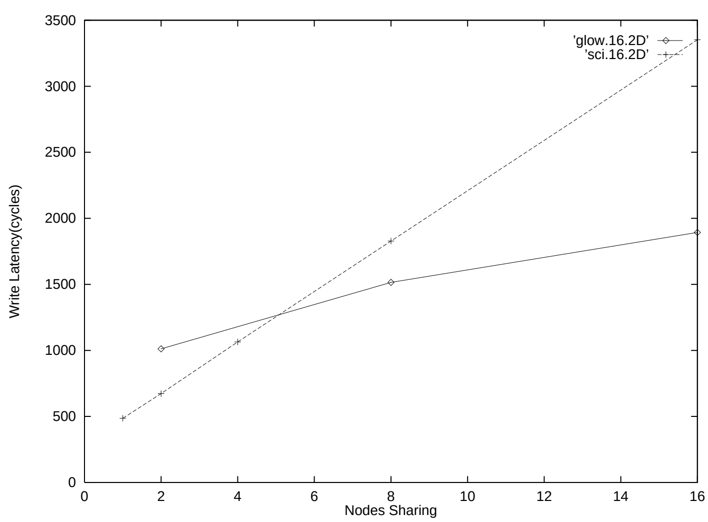


FIGURE 17. Writes in 16 Nodes (2-D)

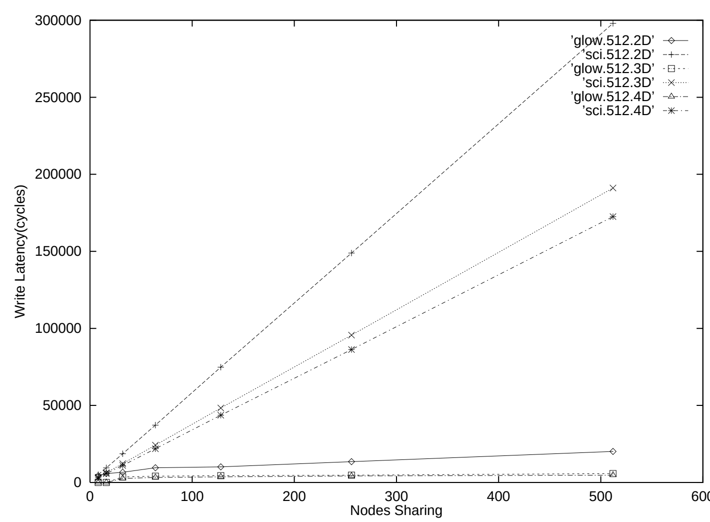


FIGURE 20. Writes in 512 Nodes (2-,3-,4-D)

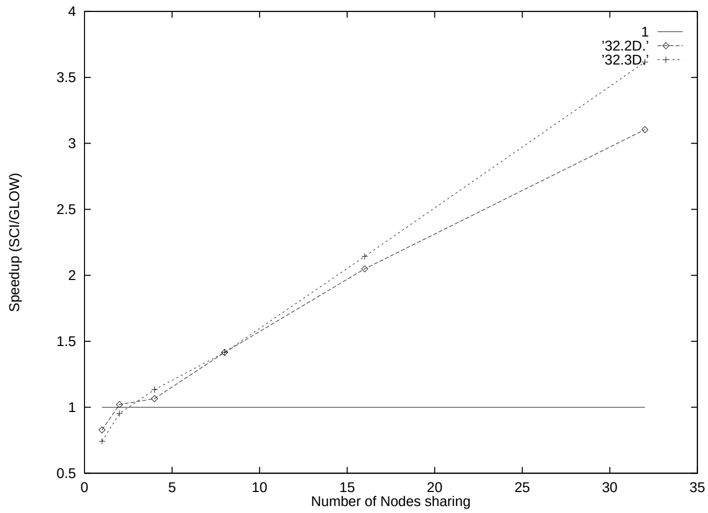


**FIGURE 21. Speed-up in 32 node topologies**

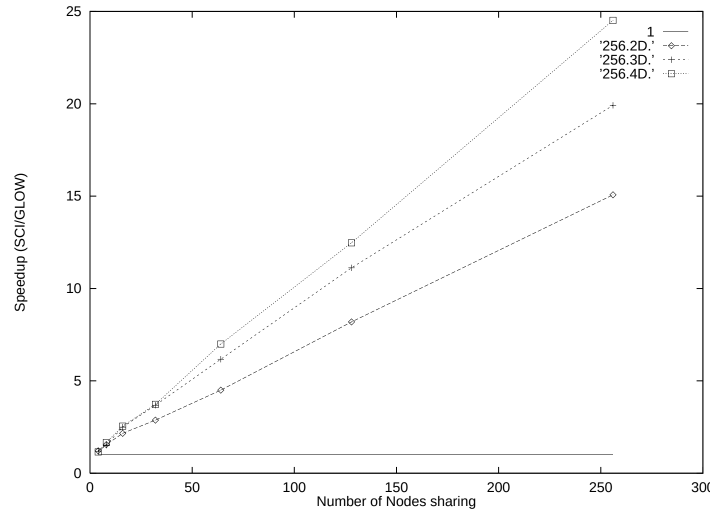


**FIGURE 23. Speed-up in 256 node topologies**

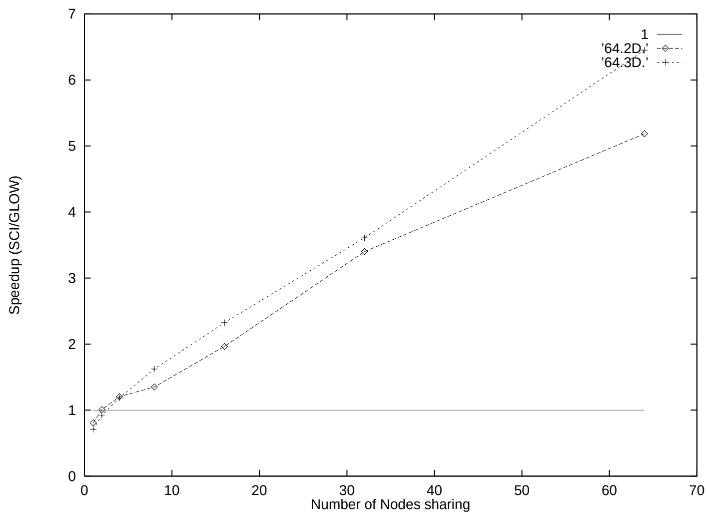


**FIGURE 22. Speed-up in 64 Node topologies**

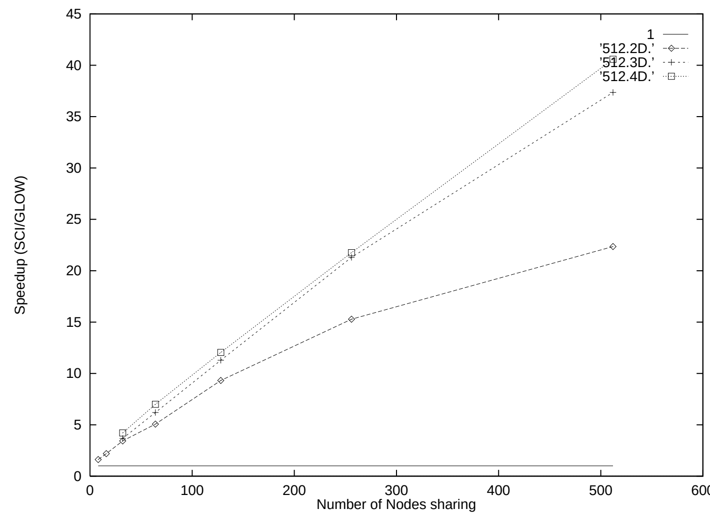


**FIGURE 24. Speed-up in 512 node topologies**