

# Multithreading and Lightweight Communication Protocol Enhancements for SCI-based SCALE Systems

Alan George, William Phipps, Robert Todd, and Warren Rosen<sup>1</sup>

*High-performance Computing and Simulation (HCS) Research Laboratory*  
Department of Electrical and Computer Engineering  
University of Florida  
<http://www.hcs.ufl.edu>

## Abstract

*Future high-performance computing systems for both general-purpose and embedded applications must be “able”: scalable, portable, dependable, programmable, and affordable. The Scalable Cluster Architecture Latency-hiding Environment (SCALE) project underway in the HCS Research Laboratory hopes to contribute to this end by bridging the gap between the research worlds of parallel computing and high-performance interconnects. This paper introduces new methods and results for lightweight communications and low-latency multithreading as part of SCALE Suite, a new software system under development for SCALE machines.*

## 1. Introduction

The goal of the SCALE research project at the HCS Research Lab [GEOR96a, GEOR96b, GEOR95] is to investigate and develop a family of techniques by which cluster architectures can be constructed with latency-hiding, high-level parallel programming and coordination languages, and low-level lightweight communication protocols in a fashion that is open, portable, high-performance, distributed, fault-tolerant, and scalable. In particular, the goal is scalability across different interconnects supporting uniform memory access (UMA), non-uniform memory access (NUMA), or cache-coherent NUMA (CC-NUMA) shared memory as well as (and in tandem with) those with no remote memory access (NORMA) via message passing. These scalable cluster architectures leverage the constantly rising levels of performance and

dropping costs in commercial off-the-shelf (COTS) hardware including state-of-the-art workstations, high-performance interconnects, adapters, switches, single-board computers, etc.

The parallel architecture of a SCALE system is constructed by combining high-performance interconnects in a multilevel fashion. The scalable topology we propose is a cluster which starts with shared-bus, shared-memory symmetric multiprocessors (SMPs) with UMA. These SMPs are combined via the Scalable Coherent Interface (SCI) [SCI93] with NUMA and CC-NUMA across local-area distances, and then clusters of these SCI-based multiprocessors are combined via Asynchronous Transfer Mode (ATM) across metropolitan-area and wide-area distances. The software aspects of the SCALE system emphasize latency-hiding mechanisms including distributed-directory cache coherence, instruction and data prefetching, relaxed memory consistency, shared virtual memory, and multiple contexts via multithreading. High-level parallel programming and coordination language implementations must be developed to exploit latency-hiding mechanisms to achieve high-performance and scalability across multiple tiers in the SCALE system, such as multithreaded MPI and Linda. Low-level, lightweight communication protocols must be developed to exploit the low-latency and high-throughput potential of the underlying high-performance interconnects. This approach will achieve flexibility

and portability without the performance drain associated with TCP/IP. While SCI and ATM are highlighted as target interconnects for this project’s demonstration system, and in some sense represent opposite ends of the high-performance interconnect spectrum, systems

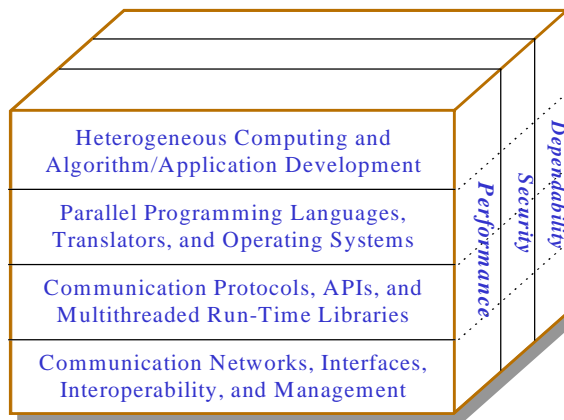


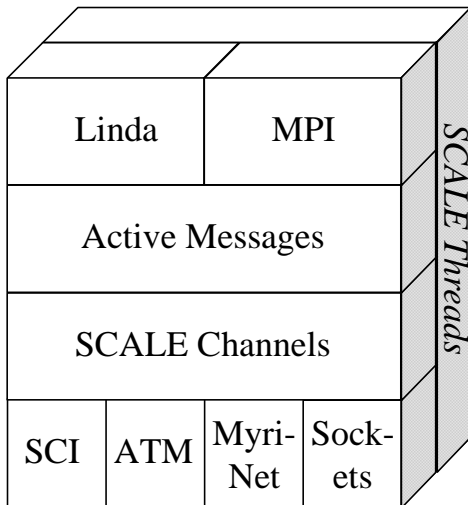
Figure 1. Core Research Areas of SCALE

and portability without the performance drain associated with TCP/IP. While SCI and ATM are highlighted as target interconnects for this project’s demonstration system, and in some sense represent opposite ends of the high-performance interconnect spectrum, systems

<sup>1</sup> Dr. Rosen is with the ECE Department at Drexel University in Philadelphia, PA.

constructed with many other high-performance interconnects can also be addressed with this technology including Fibre Channel, Gigabit Ethernet, HIPPI, SuperHIPPI, Myrinet, etc.

A key element of the SCALE project is a software collection known as the *SCALE Suite* which will provide an integrated communication and execution model introducing minimum latency, supporting distributed shared memory, and providing a suitable platform on which to develop efficient, portable, and scalable parallel applications. The mechanisms presented in this paper form a portion of the foundation for the SCALE Suite. Figure 2 illustrates the initial components in the SCALE Suite, ranging from high-level programming language translators (e.g. Linda [CARR92] and MPI [MPIF93]) to device drivers for high-speed interconnects (e.g. Dolphin's SCI, Myricom's Myrinet, FORE Systems' ATM, Berkeley's Sockets, System V streams, etc.) via a lightweight and interconnect-independent channeling layer known as *SCALE Channels*. Each of these layers takes full advantage of the low-latency thread management system known as *SCALE Threads*.



**Figure 2. Current Organization of the SCALE Suite**

This diagram shows the initial components in the SCALE Suite software library. High-level parallel programming languages and translators are augmented to efficiently interface to an optimized implementation of Active Messages which interfaces to network drivers through SCALE Channels. Execution and coordination of different threads throughout the library is handled by the SCALE Threads library.

This paper summarizes work to date in the development and enhancement of efficient multithreading mechanisms, including context and thread switching in UNIX, which reduce the switching latencies and thereby more effectively support high-speed, shared-memory interconnects. Furthermore, developments and enhancements for implementing the UCB Active

Messages specification are presented in concert with these multithreading enhancements. Finally, the results from a series of performance experiments are presented, in terms of both low-level metrics (e.g. latency, throughput, thread switching time) and high-level metrics (e.g. parallel speedup and execution time).

## 2. Multithreading Mechanisms

The multithreaded (MT) programming paradigm allows the programmer to indicate to the run-time system which portions of an application can occur concurrently. Synchronization variables control access to shared system resources and allow different threads to coordinate during execution [ROBB96]. Initially, the MT paradigm was designed as an implementation of a dataflow architecture, where parallelism in an algorithm can be maximized. However, due to performance limitations of actual systems, pure dataflow applications have been unsuccessful because of latency in the hardware and software. MT programming has since been used to introduce latency hiding in distributed systems or in a single system where components operate at different speeds.

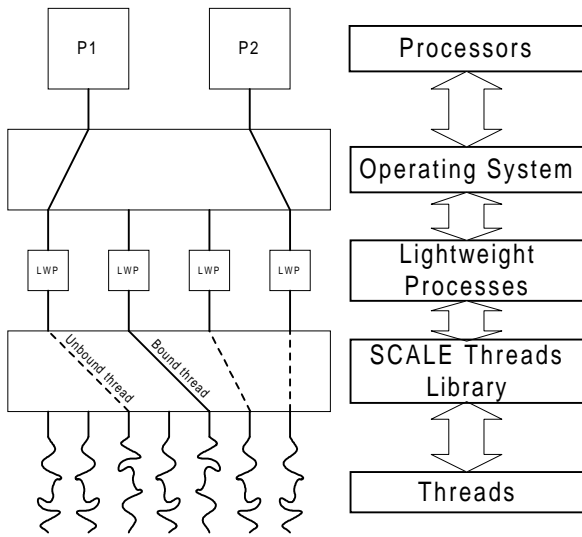
MT operating system kernels sometimes perform thread switching to keep the system reactive while waiting on slow I/O services, including tape drives, hard drives, and networks. In this way, the system continues to perform useful work while the network or other hardware is transmitting or receiving information at a relatively slow rate. Unfortunately, this approach has led to the development of heavy, inefficient thread run-time libraries that operate more like small processes than lightweight threads. With new low-latency interconnects like SCI, the thread library can no longer be considered a trivial cost.

For example, typical voluntary thread switches incur approximately thirty microseconds of latency on a 167-MHz UltraSPARC-1 workstation as compared with one-way SCI/Sbus latencies of less than four microseconds. Thus, although multithreading holds the promise to permit a processor to switch from a thread when it experiences a remote-memory access delay, the thread switching time is far too lengthy to exploit the potential of multithreading. Furthermore, results indicate that an increase in processor performance often provides at best only a marginal improvement in thread switching time due to memory access requirements, thereby making it critical to achieve new levels of efficiency in thread management systems.

Another problem with traditional thread management systems is the complexity of sending signals (the usual method to perform synchronization) on multiprocessor systems. Synchronization in multiprocessor systems using asynchronous signals degrades as more processors are added to the system because of the inherent bookkeeping

requirements. Also, signal passing for synchronization does not work over a distributed shared memory system without special hardware support.

True concurrency in a MT system can be achieved by scheduling different threads to different symmetric processors. Because the MT system must be compatible with the operating system, explicit processor scheduling may not be possible. Instead, the operating system offers multiple scheduling points into a single process by using lightweight processes. These lightweight processes are scheduled as full processes in that they are time-shared and can be executed on different processors, which allows separate domains of control for the operating system and the thread library. The operating system schedules processors with lightweight processes. The thread library schedules lightweight processes with available threads. Figure 3 shows this relationship and also shows how threads can be bound or unbound.



**Figure 3. SCALE Threads System-Level View**

This figure depicts the relationship between processors, lightweight processes (LWP's) and user-level threads in a system. The operating system multiplexes LWP's to processors and the thread library multiplexes user-level threads to LWP's.

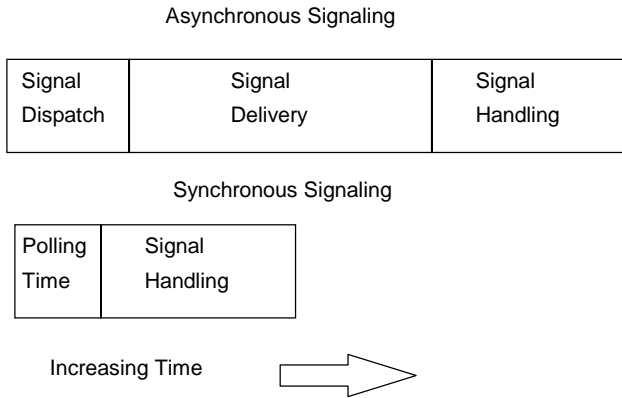
In general the MT concept offers latency hiding and concurrency, however COTS implementations have limited the performance and functionality by using heavyweight thread switching and signals for thread synchronization. Realizing the deficiencies in existing thread libraries, the HCS Research Lab has developed *SCALE Threads* which is a user-level thread management run-time system that implements opportunistic polling for synchronization and lightweight context switches to provide a thread library that can take advantage of high-performance interconnection networks and distributed shared memory.

### **SCALE Threads**

*SCALE Threads* is able to achieve a fundamentally higher level of performance by the elimination of preemption. Preemptive scheduling systems have gained much support for use in user-system interaction and error notification. They have been adapted to perform communication and coordination as well, but with limitations. *Asynchronous signaling* used in preemptive systems has a large fixed delay before handling the signal while the system stores the processor state and locking portions of the kernel to ensure atomic operation during signal delivery. The performance of preemptive systems is, therefore, dictated by the performance of the kernel in delivering signals. *Synchronous signaling* by definition uses a repetitive poll to check for incoming signals. This technique burdens the processor checking for signals and thus synchronous signaling is typically not used for general-purpose signaling in operating systems. However, because the poll is a much faster process than signal delivery, synchronous signaling can offer the best synchronization performance. The performance is determined by how much processor utilization the programmer can afford. A higher utilization yields minimal response times. Figure 4 shows the latency breakdown of logical asynchronous and synchronous signaling methods.

*Opportunistic polling* offers a synchronous polling method that can keep the processor utilization on polling minimized while maximizing performance. In a synchronous signaling system, it is assumed that the programmer is able to perform other work while waiting for a signal, however a technique to instruct the system of pending work has not yet been formalized. Opportunistic polling combines synchronous polling with a multithreaded system where the programmer has expressed concurrency by threads.

Thread switches in *SCALE Threads* are all voluntary thereby offering two advantages: faster context switches and user-controlled thread switches. Opportunistic polling is implemented by notifying the thread system what condition must be satisfied before a thread can be rescheduled.



**Figure 4. Asynchronous and Synchronous Signaling Latency Breakdown**

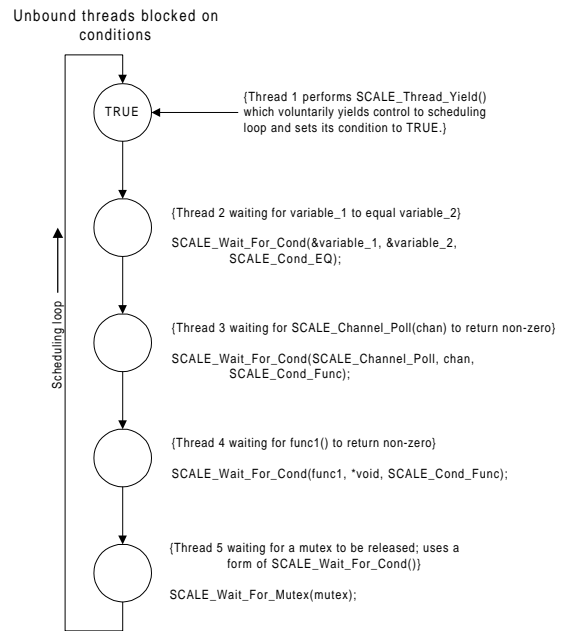
This figure shows the logical delays involved in handling an asynchronous and synchronous signal.

For example, a thread waiting for a communication to arrive posts that condition to the run-time system and yields to another thread. In this case, the condition is a “true” result from the communication polling function. The other thread will do some work and will eventually yield so that the condition may be tested again. The frequency of the other thread’s yielding will specify the performance of the system.

For instance, this yielding and condition setting process is shown in Figure 5 where five threads have been created and four of the threads are blocked on conditions. Thread 1 has entered the rescheduling loop voluntarily and has a “true” condition. This type of yield typically occurs when a thread has completed a section of work and has an opportunity to allow other threads to be executed. Thread 2 is waiting for two variables to be equal. It has notified the thread system that it is not to be executed until that condition is met. The thread system starts testing conditions that other threads are blocked on, by loading both variables and testing for equality in this case. If equal, the system performs a context switch from Thread 1 to Thread 2, otherwise the system continues in the loop, searching for a thread that is not already being executed by another processor that has a condition that evaluates “true”. Thread 3 is waiting on the condition that the function *SCALE\_Channel\_Poll(chan)* returns non-zero. By passing a function pointer into the conditional variable, a thread can block on an arbitrarily complex condition which is tested on each round of the thread scheduling process.

Threads can wait for communication events, shared variable relationships, or traditional synchronization variables such as mutexes and semaphores, which can reside in a shared-memory space allowing support for distributed shared memory. In the figure, Thread 4 shows a user function passed in as the condition to be satisfied,

whereas Thread 5 shows a traditional wait for a mutex to be posted.



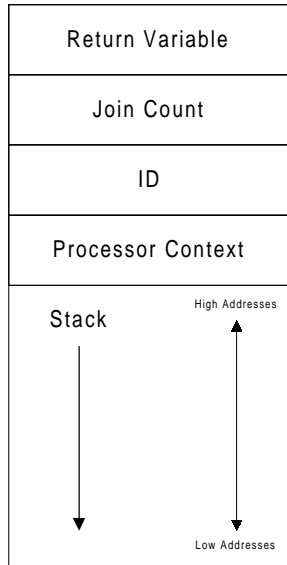
**Figure 5. Sample SCALE Threads Scheduling Loop**

This simple scheduling loop shows how threads post their yielding conditions so that another thread which needs to be rescheduled may test the posted conditions to find an unblocked thread.

SCALE Threads is able to perform arbitrary synchronization over a shared address space because it allows the user to specify a functional test to be tried when given a chance to reschedule. It is important to note that this added functionality and controllability comes at the price of the relaxation of response generation. In the asynchronous case, once the signal is sent a response will be generated as quickly as possible. In the synchronous case, the receiving thread constantly checks for the condition so any change in the condition will be noticed. Opportunistic polling relaxes this automatic response generation and allows the programmer to generate a response when it is convenient for the application. Only the programmer will know what “convenient” will mean for each application showing how opportunistic polling can be adapted to nearly any situation.

SCALE Threads scales with the number of local processors and the load on the system. As the number of processors is increased, the blocked conditions are tested more often and performance will be limited to the purely synchronous polling case which is optimal. As the load increases, the conditions are tested less frequently which leaves more computational power for the working threads. As the load decreases, the testing occurs more frequently

which means that response latency will be reduced, eventually being limited to the synchronous polling case if no threads are available to execute. Although this situation indicates that the processor will be totally utilized performing polling tasks, this is optimal since no other work was outstanding.



**Figure 6. Internal Thread Structure**

This shows the layout of a SCALE Thread. The thread information is stored directly above the thread stack so that memory can be allocated linearly when adding a thread. Likewise, thread migration can be simplified by the contiguous space used by a thread.

The structure of a SCALE Thread is shown in Figure 6. The thread information is stored directly above the thread stack so that a linear region of memory may be allocated for both the stack and the thread structure. In future work, this contiguous placement will ease thread migration for fault tolerance or load balancing. A thread stores a processor context, an integer identifier, the conditional variable used during rescheduling, a join count so that multiple threads may join with a single exiting thread, an integer (or void pointer) return variable, and two linked-list pointers to maintain the doubly-linked circular scheduling loop. The conditional variable is made up of a mutex, two pointer arguments and a test condition such as equality, less than, mutex, or function. The mutex condition lets the system know to return “true” if the mutex is obtained. Otherwise, the mutex is acquired to perform the test atomically and released at the end of the test. The function condition indicates that the first argument is a function pointer and the second argument is a void pointer argument to the function.

A SCALE Thread is actually a function call which can occur concurrently with other functions in the program. The creation function allows up to sixteen integer or void

pointer arguments to be passed to the function which differs from the single void pointer calling convention in both POSIX threads and Solaris threads [ROBB96]. By allowing a large number of arguments to be passed, existing functions can be used as thread functions instead of creating wrapper functions that accept a void pointer to the arguments.

Also different from POSIX and Solaris threads is a SCALE Thread’s ability to dynamically bind to or unbind from a lightweight process and to dynamically detach from or attach to the main execution. Dynamic binding may be useful if a thread has a portion of execution which must be executed in a time-shared fashion. A user interface thread is a good example of a time-shared thread where deterministic response time is necessary. A detached thread is a thread whose result is not needed by any other thread. In other words, no other thread will block waiting for this thread to complete and exit. Typically, threads which perform communication handling are detached threads and can be likened to daemon processes in UNIX. SCALE Threads allows threads to change binding or detaching after the thread has been created, unlike POSIX and Solaris threads which require a static status during execution.

The process of joining with a thread blocks the joining thread until the joined thread has completed, which is the fundamental type of synchronization in a dataflow application. Once completed, the joined thread produces its result for the joining thread. SCALE Threads allow an arbitrary number of threads to join with a single thread by incrementing the thread’s *join count* each time another thread attempts to join. This approach is in sharp contrast to POSIX and Solaris threads which allow only a single thread to join with another thread, which severely limits the applicability of dataflow programs to those thread libraries. By allowing multiple joining threads, a barrier synchronization routine can be easily implemented by creating a thread which blocks until its *join count* reaches a certain number and then exits to allow the joining threads to continue.

### SCALE Threads Library Performance

Five different benchmark programs were designed to quantify the performance improvement of SCALE Threads over Solaris threads. The *mutex* program measures how long it takes to try to acquire and release a mutex. The *semaphore* program measures the time taken to post and wait for a semaphore. The *context switch* program measures the latency involved in a user context switch where the processor’s registers are saved and the stacks are switched. The *thread switch* program measures the latency of switching between two threads, which consists of a context switch and other thread management duties. The Solaris result for this benchmark is an estimate since voluntary thread yields in Solaris threads

do not always force a thread switch. The *thread synchronization* program creates ten threads, each blocked on a mutex and each able to unblock the next thread's mutex. The first mutex is released which sets up a chain reaction of locking and unlocking mutexes. This process is repeated and the time to lock, unlock, and switch threads is measured. Table 1 summarizes the benchmark results when executed on a 167-MHz UltraSPARC-1 processor, using SPARC V8 source code for object compatibility with other SPARC-based platforms. All programs in this paper were compiled using the SunSoft C++ 4.0 compiler with the optimizations `x05` and `xtarget=ultra1/170`.

**Table 1. Thread Benchmark Results**

| Benchmark              | SCALE Threads | Solaris Threads      |
|------------------------|---------------|----------------------|
| Mutex                  | 0.126 $\mu$ s | 0.706 $\mu$ s        |
| Semaphore              | 0.354 $\mu$ s | 5.82 $\mu$ s         |
| Context Switch         | 0.81 $\mu$ s  | 21.7 $\mu$ s         |
| Thread Switch          | 2.33 $\mu$ s  | $\approx$ 22 $\mu$ s |
| Thread Synchronization | 2.54 $\mu$ s  | 44.79 $\mu$ s        |

The SCALE Threads implementations of synchronization variables and context handling routines greatly reduce the minimum latency to perform synchronization. A full order of magnitude is saved when performing full thread switches and the performance benefits of opportunistic polling can be realized simultaneously. The reduced latency will make finer-grain parallelism and dataflow applications possible. The thread switching latency is on the order of a direct function call which encourages programmers to introduce as much concurrency as possible into an application. As evidenced, SCALE Threads offer more functionality, support for distributed shared memory, and lower-latency synchronization.

### 3. Lightweight Communication Protocols

Even with a fast thread library that can take advantage of low-latency networks, the software communication protocol used on the network needs to be as lightweight as possible to reduce the amount of end processing. With currently available software, a slow thread library is matched with a large communication overhead so that neither seems to be relatively slow. Since the average latency of a system determines how well the performance will scale as more workstations are added, higher latency software reduces the efficiency of a distributed system and limits its scalability.

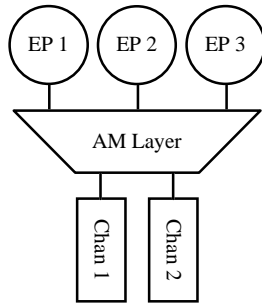
The communication protocol used for transporting data between workstations can severely affect performance. Traditional communication protocols such

as the TCP/IP stack were built to supply a generic, reliable service over relatively slow and error-prone networks. Modern high-speed networks and dedicated workstation interconnects do not require the same amount of error protection and generic service.

#### *Active Messages*

With the desire to provide a parallel, message-driven environment using a lightweight transport, the HCS Research Lab chose to use UCB's Generic Active Messages (AM) specification [MAIN95] to maintain compatibility with other software. The HCS implementation provides the AM function calls operating over generalized high-performance transport channels called SCALE Channels. Initially, SCALE Channels will be provided for Dolphin's SCI/Sbus-1 and -2B adapters, FORE System's SBA-200 ATM/Sbus adapters, and Myricom's M2F-SBus32 Myrinet/Sbus adapters, on SPARC-based workstations operating under Solaris 2.5.1. Channels have been developed also for sockets and System V streams and for both local and remote shared-memory segments. All channels offer reliable data transport with three run-time functions: read, write, and poll. These functions are non-blocking and are intended to be used with SCALE Threads, but can be used separately if needed. Configuration of these connection-oriented channels is performed at the beginning of a program and uses workstation hostnames instead of implementation-specific identifiers to create the connections.

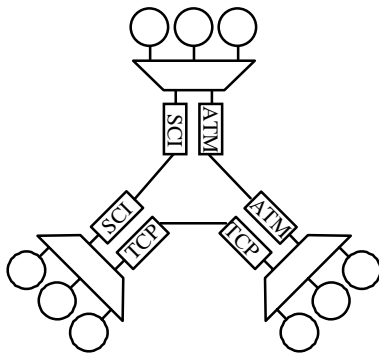
The AM layer manages access to each of these channels and provides the endpoint abstraction to the programmer. An AM endpoint is a logical destination of an active message where a specific handling function is invoked. UCB's AM specification provides a request-response type of transaction where a request for a service is sent to a specific endpoint using an active message. Upon receipt of the active message, the responder calls the handler function indicated in the active message to complete the request transaction. The response handler may reply back to the requester after some computation. Figure 7 shows the relationship of endpoints and channels.



**Figure 7. Endpoints are multiplexed across multiple channels**

Multiple endpoints can reside in a single workstation which may be connected on different networks simultaneously. The AM layer allows endpoints to be accessible from each of the networks concurrently.

A system of computers can be configured to use active messages by connecting channels to other workstations that need access to a certain endpoint. Name services are still under development to allow automatic and dynamic configuration. The channels implementation offers an abstracted view of different network devices and connections are made by specifying the hostname to connect to and the type of network device to use (e.g. SCI, ATM, Myrinet, etc.). A simple system is shown below in Figure 8 which demonstrates how multiple channels are multiplexed to multiple endpoints at each system and the channels between workstations may be heterogeneous.



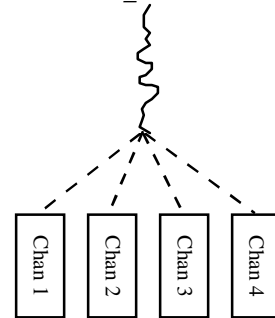
**Figure 8. Sample SCALE AM-Connected System**

This simple system shows how the abstract channels effectively hide the underlying network connections. Endpoints on any of the systems can directly communicate with any other endpoint without needing specialized transport functions for different networks.

### Techniques for Handling Active Messages

The AM transaction can be considered a low-latency remote procedure call. This type of message-driven execution is ideal for many parallel applications as the endpoints appear as execution servers where tasks can be sent for concurrent execution.

Two critical portions of the AM specification have been left for implementation: synchronicity and concurrent handling. The AM specification states that all messages must be polled for using *AM\_Poll()*. This function checks all endpoints for arrived messages, but when this poll occurs is left out of the specification. Opportunistic polling matches this functionality perfectly. To offer background message monitoring, a single thread can be created which blocks, waiting for *AM\_Poll* to find an arrived active message. The thread that calls *AM\_Poll* is also responsible for executing the handlers which, if used for parallel processing, may be computationally expensive. Figure 9 shows logically how the AM specification expects the *AM\_Poll* function call to operate.

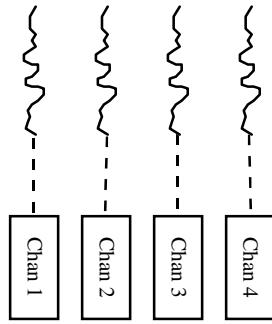


**Figure 9. *AM\_Poll* with Single Thread Servicing All Channels**

The *AM\_Poll* command handles messages from any endpoint sequentially. It returns after handling an implementation-specific number of messages to avoid the thread from being pinned down under a streaming load [MAIN95].

In the SCALE AM implementation, the *AM\_Poll* function is a single loop that calls each channel's poll function to determine if any new messages have arrived. Since each channel is independent, a thread can be created for each channel to allow concurrent polling of channels. Also, if the polling thread must handle the message, a long handler on one channel will not block other channels from operating as shown in Figure 10.

This parallel handling method is adequate for well structured parallel programs, but a more dynamic approach is needed for general parallelism. Instead of having each channel-attending thread handle the arriving active message, a new thread can be created to execute the handler, which leaves the attending thread free to accept new messages. Creating new threads for all arriving messages may be a costly technique if the thread creation time is large in comparison to the handler time. Instead, a flag can be embedded in the active message indicating whether or not to create a new thread to handle the message. In this way, long handler routines can be encapsulated in a thread to encourage concurrency while those handlers which the programmer realizes are short can be handled directly by the watching thread.



**Figure 10. Channel Poll with a Dedicated Thread per Channel**

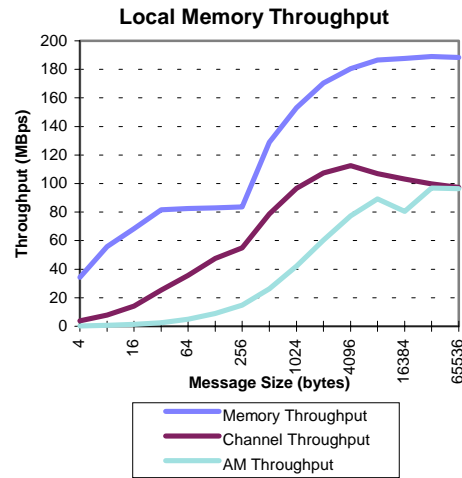
Concurrent handling of AM can be achieved by creating threads for each of the channels. In this way, concurrency is increased and a limiting number of handled messages is not needed to ensure forward progress. Each channel is handled separately and can deliver messages to the same endpoint in parallel.

#### 4. Low-Level Experiments

Although the low-level transport layer is portable, the shared-memory channel implementation has been designed especially for SCI shared memory and local memory. Remote shared memory is created by the SCI driver and is mapped and formatted by the channels driver. For performance reasons, two shared regions are created, one on each of the connecting machines. The local segment is used for incoming messages and the remote is used for outgoing messages. Flow control information is stored at the first two words in the shared segments. Since SCI provides guaranteed delivery and error free transport, retries of messages are not needed. Packets lost or truncated due to failures are not handled at this time and will be handled by a higher layer.

The protocol for shared-memory channels is very simple. The first flow control word is used as the current buffer offset while the second is used for acknowledgments. For sends, once a message has been completely copied to the remote segment, the first remote flow-control word is incremented by the size of the message. Incrementing this word allows the receiving channel's *poll* function to return "true" since the offset that the receiver has acknowledged is no longer equal to the current offset in its receive buffer. To receive, the node copies the data from its local buffer and increments the second word in its remote region (the sender's local region) to acknowledge the receipt and delivery of the message. The sender can send as many messages as can fit in the length of the buffer and uses its local region's second flow-control word to determine how much room is left to send new data. Sends that are blocked due to flow control are retried after voluntarily yielding.

The following charts indicate the throughput and latency of a direct *memcpy* interface, the buffered channel interface, and the full active message layer. Local memory performance is shown in Figures 11-12 to indicate maximum performance of these interfaces operating on a 167-MHz UltraSPARC-1 workstation running Solaris 2.5.1. The measured effective throughput is an acknowledged throughput, which means that subsequent sends must wait until the previous send has been delivered before sending the next message (i.e. unlike pipelined performance which would be based on the size of the channel and not give an accurate measure of guaranteed transmissions). The measured latency shown is acknowledged one-way latency. The AM latency represents the time taken to send an active message and receive an AM reply back including calling the handlers on both sides.



**Figure 11. Local Memory Throughput**

This chart illustrates the effective throughput and how the addition of a communication protocol reduces the available bandwidth from the underlying medium. Direct copying is used for messages below 256 bytes while the *memcpy* function is used for 256-byte and larger messages.

The local memory throughput using *memcpy* offers maximum performance since only a single copy is necessary to transfer a message directly. This test defeats the 1-MB cache of the processor by linearly copying messages which span more than 1-MB of address space. The channel performance takes advantage of the cache during the receipt of a message because the message size is smaller than the cache and is still resident when received. This method is how most transfers using the buffered channel will occur and represents an accurate measurement of expected channel performance. The AM layer predictably reduces the performance of the smaller messages since a 64-byte active message is first sent followed by the actual message to be transferred. This

performance shows the effective throughput of handled messages including an active message reply for delivery confirmation. Hence, two 64-byte active messages are delivered for each one-way data transfer.

The AM throughput approaches the channel throughput at the larger message sizes which peaks at 95-MBps. The local shared-memory channel can be compared with a UNIX pipe in functionality. Both provide a buffered, reliable data stream which can be between processes or threads. The performance of the local channel shows the benefit of eliminating kernel interaction for communications since the peak UNIX pipe throughput is approximately 50-MBps.

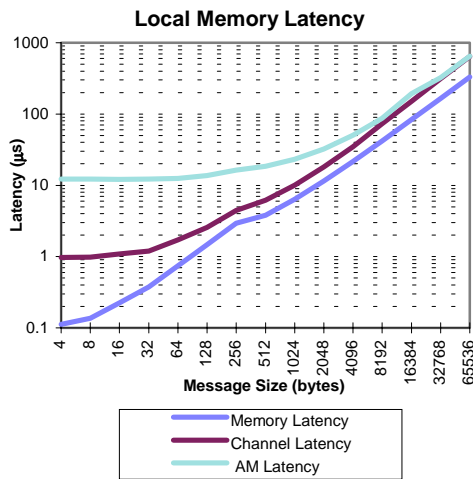


Figure 12. Local Memory Latency

This chart illustrates how the addition of a communication protocol increases the minimum latency through the underlying medium. Direct copying is used for messages below 256 bytes while the *memcpy* function is used for 256-byte and larger messages.

The local memory latency shows the effect of using a formatted and buffered communication medium. Starting from the *memcpy* latency of 0.11 microseconds, the channel protocol increases the minimum latency to about 1µs while the minimum AM latency is 12 microseconds for packets with 0-byte messages or payloads, which includes a request and reply handler call. The latency of the AM layer approaches the channel interface latency for larger messages but is always worse than direct copying due to buffering.

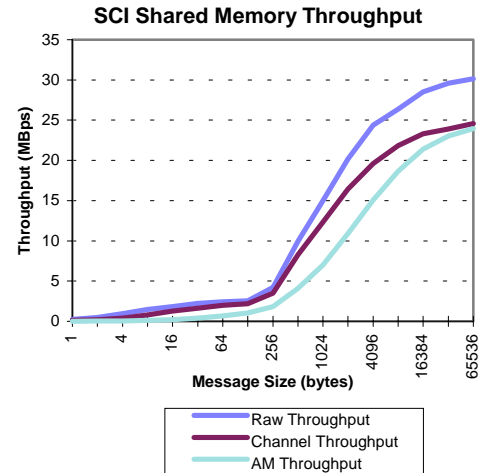


Figure 13. SCI Shared Memory Throughput

This chart illustrates the impact of using a communication protocol over shared memory via SCI/Sbus-2B. The throughput is only slightly degraded from the unformatted maximum throughput of the link.

For remote shared-memory access experiments, two UltraSPARC-1 workstations communicate in a ring of eight workstations connected via 1.6-Gbps/link Dolphin SCI/Sbus-2B adapters. A channel between these two workstations is created and compared against a raw *memcpy* into and out of the SCI shared-memory segments, and the results are shown in Figures 13-14.

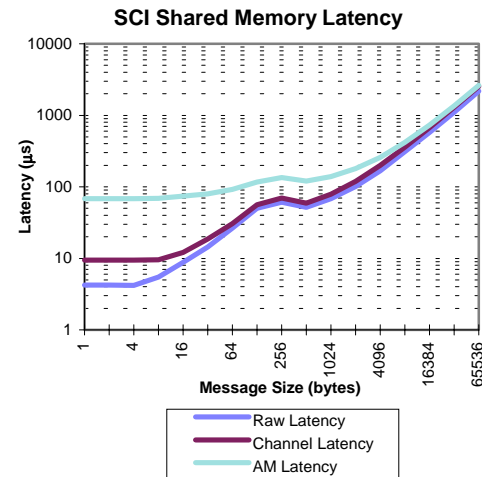


Figure 14. SCI Shared Memory Latency

This chart illustrates how the latency is increased by using a communication protocol. The minimum one-way latency is 4 µs for unformatted data, for the channel 10 µs, and for a two-way AM transaction 68 µs. Although this latency appears to be higher than other AM implementations, this AM time is an *application-to-application* measurement with full thread safety and handler invocation by the host processor.

The throughput using either channels or AM is very close to the raw throughput using *memcpy* (i.e. the peak performance in this case) since the time taken to buffer the information is smaller than the transfer time. The buffering is effectively masked because the communication is slow. The effects of the AM packets are seen as a lowering of effective throughput.

The AM latency is approximately an order of magnitude larger than other available AM implementations on other similar network hardware. It must be stressed that this implementation offers total functionality and the performance shown here is what actual multithreaded applications will experience. Results in other studies often do not include the impact of buffering and handling the active message by the host processor are showing the lower layer transport performance, not the AM-layer performance. The minimum AM ping-pong latency is 68 microseconds which depicts the actual handler-to-handler time.

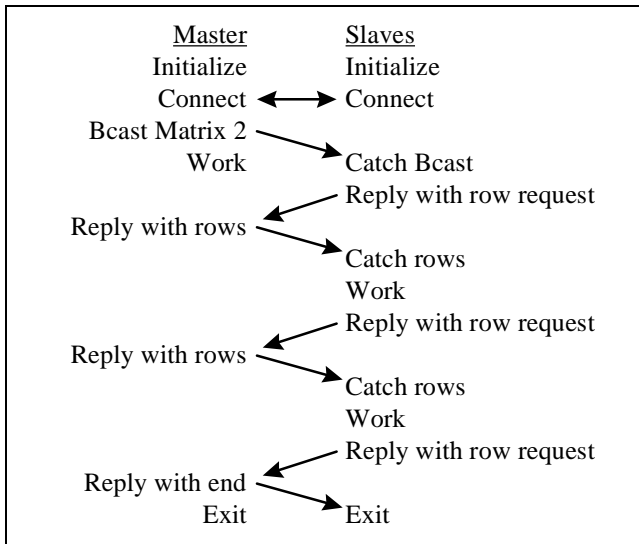


Figure 15. Flowchart for Parallel Matrix Multiply

This shows the message-driven nature of AM programs designed for parallel processing.

## 5. High-Level Experiments

A cluster of eight 128MB UltraSPARC-1 workstations running Solaris 2.5.1 and interconnected by 1.6-Gbps/link Dolphin SCI/Sbus-2B adapters serves as the platform for a series of parallel processing experiments using mechanisms in the SCALE Suite. To demonstrate our implementation of Active Messages, SCALE Channels, and SCALE Threads, one such parallel program developed and executed on the SCI-based cluster matrix is the classic  $C = A \times B$  matrix multiply. This program uses an agenda parallel approach similar to programs demonstrated in [GEOR96a], but the newest version has

been modified to use the AM primitives and the SCALE Threads library. The algorithm follows the flowchart in Figure 15.

This program has two performance tuning parameters: granularity and concurrency. The *granularity* is the number of rows that are sent by the master to a slave in response to a request for work. The *concurrency* is the number of outstanding requests a slave is allowed to have. The concurrency also determines the number of computing threads on the master node. Matrix  $B$  is assumed to be already distributed to the nodes before the program commences, while matrices  $A$  and  $C$  are distributed and collected by and at the master node respectively over SCI as part of the execution of the parallel matrix multiply program and these times are included in the performance measurements. A number of permutations were applied with respect to granularity and concurrency parameters, and the speedup and execution-time results from the optimal combinations of these parameters are shown in Figures 16 and 17 respectively. These results illustrate that despite the increased levels of functionality, flexibility, portability, and reliability afforded by the lightweight communication protocols and multithreading elements of the SCALE Suite, near-linear speedup on such widely-used, computationally-intensive applications as the matrix multiply can still be achieved.

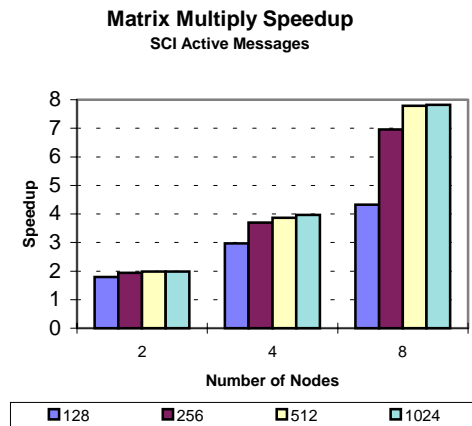
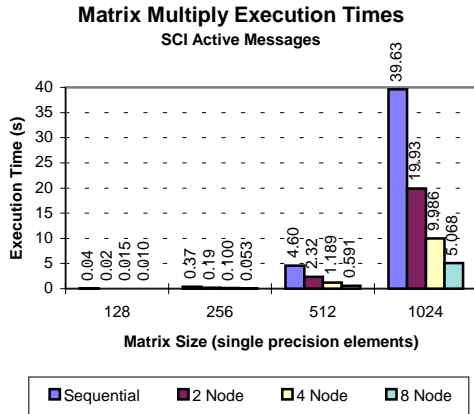


Figure 16. Parallel Matrix Multiply Speedup

The speedup of the AM matrix multiply program is shown in these charts, which runs over SCI via Active Messages.



**Figure 17. Matrix Multiply Execution Times**

The execution times for a square matrix multiply are shown above for reference purposes. The matrix sizes were chosen to ensure no needless paging occurred.

## 6. Conclusions and Future Research

With the SCALE project we hope to develop methods by which workstation clusters can be extended beyond their current limitations to provide parallel and distributed processing solutions for a wide breadth of applications in a fashion that is open, portable, high-performance, distributed, fault-tolerant, and scalable. Clusters of workstations are by far the most accessible and affordable of all potential methods for high-performance computing, and a family of techniques and tools that make it possible to construct complete parallel systems around them in a modular and scalable fashion is particularly promising. Perhaps the most critical element of any SCALE or scalable system is its software system, and a software suite known as *SCALE Suite* is being developed in the HCS Research Lab which will support the development and execution of software for parallel and distributed computing in a scalable fashion.

In this paper new mechanisms for low-latency multithreading known as *SCALE Threads* have been developed and presented. Performance experiments on a cluster of 167-MHz UltraSPARC-1 workstations connected by 1.6-Gbps/link Dolphin SCI/Sbus-2B interface adapters in a ring topology have been conducted and initial results indicate a substantial improvement in multithreading performance as compared to conventional UNIX run-time libraries. A new extension of UCB's Active Messages specification is also currently under development in the HCS Research Lab, and initial performance results with this extension operating over channels custom-designed for SCI/Sbus interfaces have been provided which show a modest amount of overhead while supporting advanced features heretofore not seen in

such protocols. In addition to low-level performance experiments with multithreading, channels, and messages, a series of high-level performance experiments have been conducted, and the results of several have been included. With the combination of SCI, SCALE Threads, and a highly-efficient protocol design which meets the Active Messages specification, virtually linear speedup has been achieved on applications whose inherent algorithmic granularity has typically been too fine-grained for conventional cluster implementation.

Future research consists of efforts in all of the core research areas of the SCALE project which incorporate SCI in tandem with other interconnects such as ATM. These core areas include concepts, techniques, and tools for: (1) heterogeneous computing and algorithm-to-application development; (2) parallel programming languages, translators, and operating systems; (3) communication protocols, application programming interfaces, and multithreaded run-time libraries; and (4) communication networks, interfaces, interoperability, and management. Specific extensions planned for the mechanisms presented in this paper include support for other networks including ATM and Myrinet, support for processor-network interfaces, and linkage to high-level parallel programming and coordination languages including MPI and Linda.

## Acknowledgments

We gratefully acknowledge our sponsors at the National Security Agency, the Office of Naval Research, and the Naval Air Warfare Center, Aircraft Division for their support. We also wish to thank Mr. Anthony Muoio of Dolphin Interconnect Solutions for equipment and software support.

## References

- [CARR92] Carriero, N. and Gelernter, D., *How to Write Parallel Programs: A First Course*. The MIT Press, Cambridge, MA. 1992.
- [GEOR96a] A.D. George, W. Phipps, R.W. Todd, D. Zirpoli, K. Justice, M. Giacoboni, and M. Sarwar, "SCI and the Scalable Cluster Architecture Latency-hiding Environment (SCALE) Project," *Proceedings of the 5<sup>th</sup> International Workshop on SCI-based High-Performance Low-Cost Computing*, September 1996.
- [GEOR96b] A.D. George, R.W. Todd, W. Phipps, M. Miars, and W. Rosen, "Parallel Processing Experiments on an SCI-based Workstation Cluster," *Proceedings of the 5<sup>th</sup> International Workshop on SCI-based High-Performance Low-Cost Computing*, March 1996, pp. 29-39.

- [GEOR95] A.D. George, R.W. Todd, and W. Rosen, "A Cluster Testbed for SCI-based Parallel Processing," *Proceedings of the 4<sup>th</sup> International Workshop on SCI-based High-Performance Low-Cost Computing*, November 1995, pp. 109-114.
- [MAIN95] Mainwaring, A.M., "Active Message Applications Programming Interface and Communication Subsystem Organization," Draft Technical Report. Computer Science Division, University of California, Berkeley, 1995.
- [MPIF93] Message Passing Interface Forum, "MPI: A Message Passing Interface," *Proceedings of Supercomputing 1993*, IEEE Computer Society Press, pp. 878-883, 1993.
- [ROBB96] Robbins, K.A and S. Robbins, *Practical UNIX Programming*, Prentice Hall PTR, Upper Saddle River, New Jersey, 1996.
- [SCI93] *Scalable Coherent Interface*, ANSI/IEEE Standard 1596-1992, IEEE Service Center, Piscataway, New Jersey, 1993.