The Nexus Approach to Integrating Multithreading and Communication

Ian Foster* Carl Kesselman† Steven Tuecke*

Abstract

Lightweight threads have an important role to play in parallel systems: they can be used to exploit shared-memory parallelism, to mask communication and I/O latencies, to implement remote memory access, and to support task-parallel and irregular applications. In this paper, we address the question of how to integrate threads and communication in high-performance distributed-memory systems. We propose an approach based on global pointer and remote service request mechanisms, and explain how these mechanisms support dynamic communication structures, asynchronous messaging, dynamic thread creation and destruction, and a global memory model via interprocessor references. We also explain how these mechanisms can be implemented in various environments. Our global pointer and remote service request mechanisms have been incorporated in a runtime system called Nexus that is used as a compiler target for parallel languages and as a substrate for higher-level communication libraries. We report the results of performance studies conducted using a Nexus implementation; these results indicate that Nexus mechanisms can be implemented efficiently on commodity hardware and software systems.

1 Introduction

Lightweight threads are often used to simplify the design and implementation of applications that must simulate or respond to events that can occur in an unpredictable sequence [33]. They are also, increasingly, used to exploit fine-grained concurrency in programs designed to run on shared-memory multiprocessors [20]. However, until recently lightweight threads were not used extensively in high-performance distributed-memory parallel computing, in large part because the focus of application development in this environment was on regular problems and algorithms. In these situations, the programmer can reasonably be assumed to have global knowledge of computation and communication patterns; this knowledge can then be exploited to develop programs in which a single thread per processor exchanges data by using send/receive communication libraries.

A number of factors are leading to an increased interest in multithreading in parallel computing environments, notably more irregular, heterogeneous, and task-parallel applications,

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439.
†Beckman Institute, California Institute of Technology, Pasadena, CA 91125.

in which communication patterns cannot be easily determined prior to execution; heterogeneous environments, in which parallel computers are embedded into networks, with variable and unpredictable latencies; and the integration of parallel computers into client-server applications. Multithreading has proven useful for overlapping computation and communication or I/O [7, 22], for load balancing [14], and for implementing process abstractions in parallel languages [10, 12, 25, 28, 34, 45].

A difficult issue that must be addressed if multithreading is to be used successfully in distributed-memory environments is the integration of threads and communication. In single-threaded send/receive communication models, a send operation in one process is matched by a receive in another, and serves both to transfer data and to enable execution in the other process. What should happen in a multithreaded environment, where many threads may be executing in a remote address space? Should a message be directed to a thread, or should threads and communication operations be decoupled? Should the code executed in response to an incoming message be specified by the sender or the receiver of the message? In this paper, we propose answers to these and other related questions. We advocate an approach in which *global pointers* are used to represent communication structures and *remote service requests* are used to effect data transfer and remote execution. We explain how these two mechanisms interact with each other and with threads, and how they can be used to implement a wide variety of parallel program structures.

Our proposed global pointer and remote service request mechanisms have been incorporated into a multithreaded communication library called Nexus [26], which we and others have used to build a variety of higher-level communication libraries [31, 27, 18] and to implement several parallel languages [10, 41, 25]. We use a Nexus implementation to perform detailed performance studies of our proposed communication mechanisms on several parallel platforms. The results of these studies provide insights into the costs associated with their implementation in different environments.

In brief, the principal contributions of this paper are threefold:

1. The design of a simple set of mechanisms for constructing multithreaded computations on distributed-memory computers.

2. Implementation techniques that permit efficient execution on commodity systems.

3. A detailed evaluation of the performance of these mechanisms on multiple platforms.

2 Background

We first review approaches to the problems of organizing computation, communication, and synchronization in parallel and distributed systems. We categorize various approaches in terms of the degree of global knowledge assumed of the programmer or compiler. In the strongly homogeneous environment of single-program, multiple-data (SPMD) programming, the programmer or compiler can reasonably be assumed to have global knowledge of the operations performed at each processor. This assumption allows the use of a message-passing model in which each communication operation involves the explicit participation of a sender and a receiver. Because communication patterns are predictable, the programmer or compiler can place a matching receive for each send in a program. Libraries supporting this model, such as p4 [9],

PVM [19], and MPI [24], are designed for programmer use but are also used as compiler targets by data-parallel languages such as High Performance Fortran (HPF) [37], Fortran-D [38], Vienna Fortran [11], and pC++ [30].

In less regular, loosely synchronous problems, a programmer or compiler may possess global knowledge of the successive phases executed by a program, but not of the the precise sequence of operations and communications performed in each phase on each processor. If the unknown sequence is repeated many times, a technique called *runtime compilation* can be used to obtain and cache knowledge about the communication patterns used in the program; this knowledge can then be applied by an SPMD message-passing library. This approach is used in the CHAOS library [42], for example. In other situations, the sequence of operations performed cannot feasibly be determined or reused. In some such situations, an adequate solution is to extend the message-passing model to allow *single-sided operations*. Here, a sender specifies the data that is to be transferred and a remote operation that is to be performed with the data. The data is then transferred and the remote operation performed without user code on the remote processor executing an explicit receive operation. The remote operation may be performed by specialized hardware (e.g., remote memory put/get operations on the Cray T3D) or by an interrupt handler. Active messages [**?**] are a well-known instantiation of this concept.

The set of operations that can be performed by a single-sided communication mechanism such as active messages is typically restricted because of the need to run in an interrupt service routine. For example, active messages cannot allocate memory or initiate communications to other processors. Yet we often wish to perform more complex operations on a remote processor: for example, allocate memory in order to expand a tree structure, or initiate another communication in order to dereference a linked list. These requirements motivate the introduction of more full-featured remote operations, and specifically the ability for these operations to create new execution contexts that can suspend, initiate communication, and so forth. This leads us to a more general *multithreaded* execution model, in which a lightweight thread of control is created for remote operations. These threads execute in a shared address space, with each thread typically having its own program pointer and stack. Multithreading becomes even more useful on a shared-memory multiprocessor, as different incoming requests can then execute in parallel with each other and with other computation.

A second motivation for introducing multithreading into parallel computations occurs on the sending side of a remote operation. As the degree of regularity in a problem decreases, the effort required to keep track of outstanding communication requests increases. Lightweight threads can address this problem by serving as loci of control for remote communication operations. For example, in a program in which several operations require remote values to proceed, each operation can be encapsulated in a separate suspended thread, to be resumed when the required remote value is available.

3 Integrating Communication and Multithreading

When introducing multiple threads of control into a distributed-memory computation, one must consider whether and how threads interact with low-level communication primitives. A minimal approach would be simply to combine a message-passing communication library, such as MPI [24], and a thread library, such as POSIX threads [39]. However, we argue that the

communication primitives provided by current message-passing libraries are not an appropriate basis for multithreaded applications. Reasons include the following:

- Point-to-point communication routines assume that threads exist on both the sending and receiving sides of a communication. While this assumption is reasonable in SPMD applications, the design of irregular task-parallel multithreaded computations can be significantly complicated if they must ensure that a thread exists and is waiting on the receive side. It is often more natural to imagine threads being created as a result of communication.

  Another way of viewing this problem is that message-passing communication binds the destination endpoint of a communication to a specific thread of control. However, in multithreaded computations, communication could be serviced by (a) a preexisting thread, (b) one thread out of a collection of preexisting threads, or (c) a thread created specifically to service the communication. We believe that low-level mechanisms should support all three alternatives.

- Irregular, multithreaded computations will often create, and must be able to refer to and communicate with, large numbers of dynamically created entities. Yet because point-to-point communication mechanisms direct messages to processes, they provide only a limited namespace for communication. Message tags and communication contexts [24] help to some extent, but depend on either global knowledge of how the tag space will be used, or a synchronous, collective operation to establish a new context. Neither option is attractive in the dynamic, asynchronous environment one expects to see in multithreaded computations.

- Message-passing communication implies a synchronization point at the receiving end. In multithreaded computations, such synchronization may not be desired or make sense. This is especially true when the purpose of a communication is to create dynamically a new thread of control in a remote location.

We address these problems by introducing mechanisms that decouple the specification of a communication's destination endpoint from the specification of the thread of control that responds to that communication. In our approach, a communication endpoint is represented by an object called a *global pointer* (GP), while communication is initiated and remote computation invoked by an operation called a *remote service request* (RSR: see Figure 1). By structuring computations in terms of these mechanisms, we are able to resolve the problems just enumerated. No thread need exist to receive an incoming communication; endpoints can be created dynamically, without the involvement of other nodes; and remote operations can be performed without synchronization at the receiving node.

The GP and RSR mechanisms are implemented as part of a system called Nexus that we have developed to support experimentation with multithreading and communication. Nexus is structured in terms of five basic abstractions: nodes, contexts, threads, GPs, and RSRs. A computation executes on a set of *nodes*, and consists of a set of *threads*, each executing in an address space called a *context*. (For the purposes of this paper, it suffices to assume that a context is equivalent to a process.) An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context.

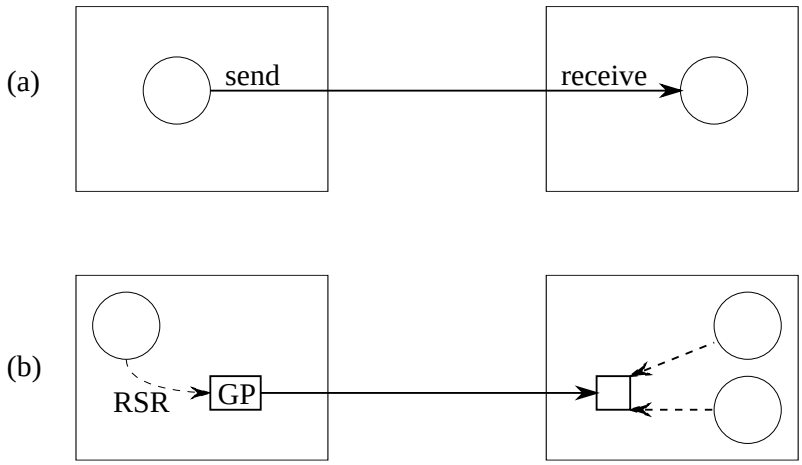

Figure 1: *The message-passing and RSR communication models contrasted. The large rectangles represent contexts, the circles threads, and the dashed lines local pointers. In (a), a thread uses message passing to send a message directly to another thread. In (b), a thread invokes an RSR on a GP referencing an endpoint in another context.*

In the rest of this section, we introduce the GP and RSR mechanisms, and explain how they support computations in which processes, communication structures, and threads can be created and destroyed dynamically, and in which threads can be created and destroyed independently of the communication structures over which they transfer data.

3.1 Global Pointers

As stated above, a GP represents a communication endpoint: that is, it specifies a destination to which a communication operation can be directed by an RSR. GPs can be created dynamically; once created, a GP can be communicated between nodes by including it in an RSR. A GP can be thought of as a capability [43] granting rights to operate on the associated endpoint.

As its name implies, a GP is associated with a memory location in the address space (context) to which it refers. This association of an address with an endpoint is not essential to the realization of the endpoint concept. However, the global address space that is provided by this association accommodates the requirements of the irregular applications that are likely to exploit multithreading. SPMD computations naturally associate communication with the section of code that consumes the data. Less regular computations need to associate communication with a data structure or code segment known to the sender but not necessarily to the receiver. These computations can use GPs to construct complex, distributed data structures. In addition, we note that GPs are consistent with the trend toward direct hardware support for a global shared namespace in distributed-memory computers.

3.2 Remote Service Requests

As explained in Section 2, irregular computations can often benefit from the ability to perform remote operations without the explicit intervention of user code on a remote processor. This requirement motivates the introduction of the *remote service request* (RSR). An RSR is invoked by specifying a handler function, a GP, and the data to be communicated; it causes the specified handler function to execute in the context pointed to by the GP. The handler is invoked asynchronously in that context; no action, such as executing a receive, needs to take place in the context in order for the handler to execute. The data transferred between contexts by an RSR is specified by a buffer and packing and unpacking functions similar to those found in MPI [24] and PVM [19]. As we will demonstrate in Section 5, the buffer manipulation routines for RSRs can exploit the same buffer management optimizations that are used by traditional message passing systems; hence, no specialized data-transfer mechanisms are required. An RSR invocation proceeds in three steps:

1. The RSR is initialized by providing a GP and handler identifier. In order to support heterogeneous architectures and dynamic composition of programs, a handler identifier can be either a string or an integer identifier. The initialization operation returns a buffer.

2. Data to be passed to the remote handler is placed into the buffer; in addition to the usual datatypes, GPs can be included. The buffer uses the GP provided at initialization to determine whether any data conversion or encoding is required.

3. The RSR is performed, with the GP provided at initialization being used to determine which low-level protocols should be used to perform the communication. (For example, a machine-specific communication library is typically used if the node on which the context resides is in the same computer, while TCP/IP is used in other circumstances.)

The handler is invoked in the destination context with the local address component of the GP and the message buffer as arguments. The passing of the local address component makes it possible to distinguish between different local objects.

A difficult issue that must be addressed in designing systems such as Nexus is the functionality supported in a handler. In general, the less a handler is allowed to do, the faster it can execute, since less must be done in order to create a local context for its execution. However, simpler is not necessarily better: if handler functionality is too constrained, applications must implement potentially complex protocols on top of the handler mechanism. Another complicating factor is that the range of things that can be done "cheaply" can vary significantly from system to system. Given these uncertainties, it is not surprising that researchers have investigated a range of approaches. Remote-memory put/get operations and active messages are examples of relatively high-performance, low-functionality mechanisms. Many systems support only nonblocking handlers [6, 15]. At the lower-performance, higher-functionality end of the scale we see systems such as Panda [3] and Isis [4], which create a new thread for every incoming RPC. Compromise approaches are also possible, such as optimistic active messages [46], which creates a thread only when runtime tests determine that a handler would suspend.

In designing the Nexus system, we took into account the fact that we are targeting non-SPMD applications, in which relatively complex handler functions are commonplace. In addition, we observe that programmers and compilers are often able to identify handler invocations—such as remote memory operations—that can be guaranteed not to suspend execution. Hence,

we chose to support two types of RSR, the *threaded* RSR, in which a new thread of control is created to execute the handler, and the more lightweight *nonthreaded* RSR, which allows the handler to execute in a preallocated thread. The programmer or compiler can use the latter form if they can guarantee that a handler will terminate without suspending.

The nonthreaded RSR is an important optimization because it can avoid the need to allocate a new thread; in addition, if a parallel computer system allows handlers to read directly from the message interface, it avoids the copying to an intermediate buffer that would otherwise be necessary for thread-safe execution. As an example, a handler that implements the get and put operations found in Split-C [16] can take advantage of this optimization.

In addition to the RSR mechanism just described, Nexus also provides routines for creating and managing threads within a context. These functions are modeled on a subset of the POSIX threads standard [39] and provide for local thread creation, termination, condition variables, mutual exclusion, yield of the current thread, etc. The use of a POSIX-compliant thread enhances portability; a disadvantage is that POSIX was designed as an application program interface, with features such as real-time scheduling support that may add overhead for parallel systems. A lower-level interface designed specifically as a compiler target would most likely result in better performance [1, 8, 21] and will be investigated in future research.

3.3 Discussion

It is instructive to compare and contrast Nexus mechanisms with other parallel programming libraries that seek to integrate multithreading and communication.

Distributed-memory computers are most commonly programmed by using communication libraries that support point-to-point and collective communication operations. Early libraries of this sort often incorporated implicit state such as default message buffers, and hence were not easily integrated with thread libraries [13, 23]. The more modern Message Passing Interface (MPI) [32] is designed to be thread safe, but provides no explicit support for communication between threads. The Chant runtime system [35] augments MPI calls with a thread identifier. In Mach [47] and NewThreads [22], threads communicate by sending messages to specific ports; different threads can receive from a port at different times. Tags, communicators, thread identifiers, and ports all allow messages to be sent to or received from a specific thread. However, they do not provide the decoupling of communication and endpoint provided by the GP and RSR. In particular, each send must be matched by an explicit receive.

The message-passing and RSR mechanisms are also distinguished by the techniques used to support the dynamic integration of new computational resources into computations. In message-passing systems, a thread with a well known name (the "initial thread") must be created on each node; this initial thread then responds to requests to create new threads, etc. This requirement leads to a fundamental asymmetry in the multithreaded model: for example, the initial thread presumably cannot be terminated. In Nexus, new contexts—created at program startup time, or later—initially have no thread running in them. (The one exception is the initial context, which runs the main thread for the program.) When new contexts are created, a GP to the new context is returned. Additional threads can be created by using this GP to make RSRs to the new context. Hence, threads are created as a byproduct of communication, and no initial threads are required to bootstrap a computation.

Other systems with similar goals to Nexus are Panda [3], Cilk [6], Athapascan [15], and

Converse [40]. Panda provides threads, RPCs, and ordered group communication as primitive operations. We discuss the suitability of RPC as a primitive below. Ordered group communication can be an important primitive in distributed-computing applications [4], but is of less relevance in parallel computing. Cilk and Athapascan provide run-to-completion threads to provide a substrate for automatic load-balancing algorithms; the more full-functioned threads required for many parallel applications are not supported. Converse supports a message-driven execution model using a construct similar to an RSR, but does not support the concept of a GP.

The RSR and GP mechanisms are also related to the remote procedure call (RPC) [5, 2]. However, while research has identified RPC as an important structuring principle for client-server systems, we argue that it is not an appropriate primitive mechanism for parallel computing. Synchronous RPC has the advantage of semantic simplicity but imposes too much policy by enforcing an explicit two-way synchronization on every communication. Parallel programs are not typically structured in terms of synchronous requests for remote services. Rather, they use more complex, custom-designed synchronization patterns implemented in terms of one-way data transfers or synchronization. The argument marshaling performed by RPCs upon call and return is another area in which RPC mechanisms prove too restrictive. The rich set of message buffer management mechanisms provided in MPI illustrates the importance of buffer management to high performance. A third disadvantage of the RPC is that it does not provide an inherently multithreaded semantics. If we make two RPCs to a server, one to extract a message from a list and the other to add a message to a list, we may not be guaranteed that one will not block the other.

In conclusion, we note that RSRs, message-passing, RPC, and other mechanisms can also be compared and contrasted with respect to the implementation complexity and performance implications of attempting to layer each one on top of the other. For example, an RPC is naturally implemented a pair of RSRs, but an RSR implementation in terms of RPCs introduces unnecessary synchronization points. Similarly, message passing can be implemented in terms of RSR (a send makes an RSR to deposit data in a remote queue: Section 5 evaluates some of the costs involved), while a message-passing implementation of RSRs requires the introduction of a specialized thread dedicated to serving RSR requests.

4 Implementation Issues

In this section and the next, we first describe general implementation issues that arise in implementations of the GP and RSR mechanisms, and then evaluate the cost of these mechanisms in several systems.

As described in the preceding section, the RSR mechanism allows a thread to invoke computation in another context. A critical implementation issue relates to how the computation invoked by the RSR is scheduled in the remote context. There can be complex tradeoffs between the rapidity with which an RSR can be scheduled for execution (and hence the timeliness of any response), the functionality supported in an RSR, and the overheads incurred in the context in which the RSR executes. The techniques used in a particular system will depend also on whether or not we wish (or are able) to modify operating system (OS) code, and on the functionality supported by the OS and associated thread and communication systems. In

particular:

- Can the operating system (OS) suspend a thread that is waiting for communication, and resume that thread when a message is detected? In many contemporary systems, a thread that calls a blocking communication function will also block all other threads in the process in which it executes.
- Is the underlying communication system reentrant? Many contemporary communication systems are not, in which case locks must be used to ensure mutual exclusion during communication calls, and care must be taken to prevent deadlock.
- Are threads scheduled preemptively? (This means that a running thread can be descheduled without an explicit yield, either because a timer interrupt signals end of a *scheduler time slice*, or because some other OS event causes a higher-priority thread to be scheduled.) Many thread systems do not support preemption.
- Are thread priorities supported by the scheduler? Many thread systems do not support priorities (which complicate the algorithm used to determine the runnable thread). Instead, they support only nonprioritized FIFO scheduling (in the absence of preemption) or round-robin scheduling (if preemption is supported).

In each case, an affirmative answer tends to simplify RSR implementation, but also increases the cost of thread management and related operations, by requiring more complex logic in the scheduler and/or introducing more critical sections that must be protected by locking.

In the following, we examine four general approaches to RSR scheduling, based on a probing communication thread, a blocking communication thread, computation-thread probing, and interrupt handlers.

4.1 A Probing Communication Thread

We first discuss implementation approaches that are appropriate for systems (e.g., the MPL library on the IBM SP2 running AIX 3.2) that do not support the blocking of a single thread on a communication operation. In these systems, it is necessary to perform nonblocking reads (probes) to detect pending RSRs. In the approach that we describe here, a specialized *communication thread* performs these probes. The use of a dedicated communication thread allows us to exploit a preemptive scheduler to ensure that probing is performed at regular intervals. It also avoids difficulties due to (for example) stack overflow that might arise if a handler were executed in an existing thread. Notice that the communication thread is an artifact of the particular implementation strategy and is not visible to the application programmer. The communication thread is defined as follows:

```
for ever
   probe_for_incoming_rsrs
   yield
endfor
```

Each time it is scheduled, this communication thread checks for and processes any pending messages, and then relinquishes the processor. Notice that in this approach, the communication thread may execute even when no RSRs are pending. A number of behaviors are possible, depending on the functionality of the thread system, and providing different latency/overhead tradeoffs:

1. If the thread system is preemptive and supports priorities, then by making the communication thread high priority we can ensure that an RSR will be scheduled with a delay no greater than the scheduler time slice. (This assumes that a yield allows a lower priority thread to execute; a disadvantage is that we pay the cost of a communication thread schedule and poll at every timeslice or thread switch.)

2. If the thread system is preemptive but does not support priorities, then an RSR can be delayed by the product of the number of active threads and the scheduler time slice.

3. If the thread system is nonpreemptive, then the delay will depend on when in the computation the next scheduling point occurs, and whether or not the scheduler supports priorities.

In Section 5 below, we evaluate the performance of the communication-thread strategy in an nonpreemptive environment.

If the thread scheduler supports priorities, then an alternative approach is possible in which a *low* priority communication thread executes an infinite loop that calls a blocking read and a thread yield. Because it runs at low priority, this thread is scheduled only when no application threads can execute, and then blocks until an RSR is available. Advantages of this approach are that the communication thread runs only when there is no other work to do, and then does not consume resources while waiting. In addition, a blocking receive often responds more quickly to an incoming message than a polling loop. Because this approach can yield unbounded RSR delays if used in isolation, it is typically most appropriately used in conjunction with probing (see below).

4.2 Scheduling via Probing

While conceptually simple, the communication thread as just described provides only limited control over when RSRs are detected and processed. (Even if preemption is supported, a typical timeslice value is 1/10 of a second.) An alternative approach that can provide greater responsiveness in some situations is to make computation threads perform probe operations that check for pending RSRs. Probe operations can be performed whenever a computation thread calls a thread or communication operation. (This approach is used successfully to dispatch active message handlers in some implementations of active messages [**?**].) They can also be introduced into user computation code automatically, by a compiler or object-code instrumentation tool. In both cases, it may be feasible for the computation thread that detects RSRs to execute the handlers (or create threads) directly; alternatively, the computation thread can enable a high-priority communication thread.

Clearly, there are tradeoffs to be made between the frequency of probing and the responsiveness with which RSRs are processed. On some machines, the cost of detecting a pending

RSR can be quite low: for example, an MPL probe can be performed in less than 0.5 μsec on the IBM SP2. In other situations, probe operations are expensive (e.g., more than 100 μsec for an TCP/IP probe on the SP), in which case probing is less attractive.

In preemptive systems a communication thread can be useful as an adjunct to probing, as it allows us to ensure that probe operations are performed within a bounded interval.

4.3 Blocking Threads

We next consider the situation in which an OS allows a single thread to block waiting for a communication operation, without blocking other threads in the computation. The OSF/AD OS used on the Intel Paragon is an example. When the communication operation completes, the OS is notified and the blocked thread resumes execution. In this situation, we can define a communication thread as follows:

```
for ever
   blocking_receive
   process_rsr
endfor
```

When an RSR arrives, this thread can execute a nonthreaded handler directly, or create a new thread to execute a threaded handler.

If priorities are not supported (this is the case on the Paragon, for example), then the delay in processing a pending RSR is bounded by the OS's thread scheduling preemption timeslice, multiplied by the number of active threads. If priorities *are* supported, we can reduce the delay between RSR arrival and processing by running the communication at high priority so that an incoming RSR preempts the execution of computation threads. However, while certain specialized systems support low-overhead preemption (e.g., the MIT J Machine [17]), in most existing systems preemption is an expensive operation involving a processor interrupt.

4.4 Interrupt-driven Scheduling

A fourth approach to RSR implementation uses a processor interrupt to notify a thread scheduler of a pending RSR. In principle, an interrupt can be used to stop the currently executing thread so that the thread scheduler can examine the system state and decide when to process the RSR. In practice, the cost of delivering an interrupt up through the OS to an application is high. For example, on an IBM SP2 the time to deliver a zero-length message using an interrupt-driven mechanism (the MPL RECVNCALL function) is 220 μsec. For comparison, one-way latency for a conventional message-passing communication is 60 μsec. In addition, the interrupt-driven mechanism, like an active message, places restrictions on the actions that a function can perform, hence restricting the range of applications for which it is useful.

Franke and Rivière [29] have implemented a specialized multithreading and communication system on the IBM SP2 in which an incoming communication uses an interrupt to perform a preemptive thread switch to a designated communication thread. This thread executes nonthreaded RSRs directly and creates threads for threaded RSRs. This approach allows full-featured RSRs, but incurs the cost of an interrupt-driven receive plus additional thread management costs associated with the switch to the communication thread.

4.5 Discussion

The above discussion emphasizes the wide range of implementation strategies that may be applied when implementing systems that integrate threads and communication. In principle, one might wish to switch between alternative strategies dynamically, according to application requirements. Nevertheless, it has been our experience that sensible defaults work well for a wide variety of applications. In essence, these defaults correspond to the use of a communication thread in all environments; the use of preemptive scheduling, when this is available, to bound latency; the use of blocking communication, when this is available; and the use of probe operations in Nexus calls, and also in the communication thread if blocking communication is not supported.

The one problematic issue that remains in our implementation of the RSR mechanism is the potentially high latencies that can be encountered, particularly in computations that do not make frequent communication calls. This issue become particularly important in applications that use RSRs to achieve remote memory access, and when we attempt to provide fast collective operations among threads, as is required (for example) when a multithreaded computation incorporates SPMD components [36]. A general solution to this problem will require new OS structures that integrate communication and multithreading more tightly than current systems. We are also hopeful that automatic insertion of probe operations will be helpful in this regard, and propose to investigate this technique in future research.

5 Performance Studies

We noted in Section 4 that different RSR implementation approaches can have radically different performance characteristics. In this section, we report results of experiments that enable us to quantify some of these costs in selected environments.

We emphasize that these experiments are performed not with an artificial test harness, but with a full-featured multithreaded communication library that is in regular use in a variety of large-scale application projects and compiler research projects. Parallel libraries and languages that use Nexus include nPerl [27], an RPC extension to the Perl 5 scripting language used for writing systems administration and management programs; an implementation of the MPI message-passing standard based on the MPICH [31] framework; the CAVEcomm communication library [18], used to achieve client-server interactions between high-performance computing applications and the CAVE virtual reality system; and a variety of high level parallel programming languages including CC++ [10], CYES-C++ [41], and Fortran M [25].

The experiments that we describe use simple benchmark programs that perform point-to-point communication between two threads using a variety of mechanisms. These apparently trivial benchmark programs are valuable because they allow us to measure accurately the overheads associated with various forms of RSR. Notice that this single-threaded, synchronous scenario represents a worst case situation for a multithreaded system, in that threads cannot be used to advantage and a native message-passing code has complete knowledge of when data will arrive. Hence, we should not be surprised that our results are expressed in terms of overheads relative to low-level message passing.

Three different machines were used for the experiments: an IBM SP2 with Power 2 processors and running AIX 3.2.5 (referred to as the SP in the following), an Intel Paragon/MP, and

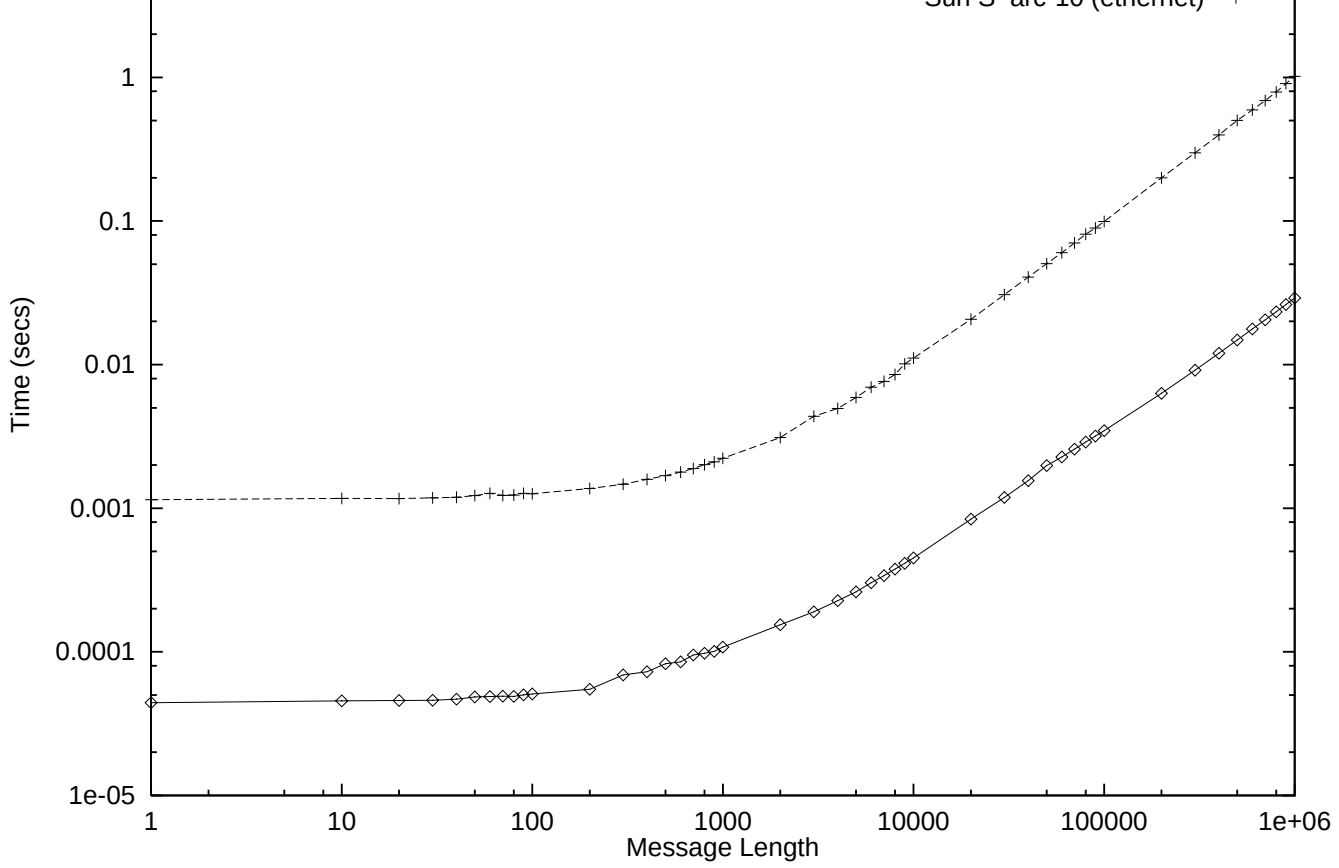

Figure 2: *One-way message latency as a function of message size using low-level message protocols on the SP and Sparc. Notice the use of log scales. On the SP, communication is achieved using the MPL library's blocking send (*`mpc_bsend`*) and blocking receive (*`mpc_brecv`*) functions. On the Sparc, we use read and write operations to communicate over a TCP stream.*

two Sparc-10 workstations running Solaris 2.3 and connected by an Ethernet (referred to as the Sparc). Since Paragon results mirrored SP results in most respects, for brevity we report results only for the SP and Sparc.

The nodes on each machine run a multitasking operating system. On the Sparc, threads are supported within the kernel and can block on a communication operation without blocking other threads executing in the same process. On the SP, threads are implemented by a user-level library, and blocking is not supported.

5.1 Native Communication

To provide a basis for comparison, a native test code was constructed that bounces a vector of fixed size back and forth between two processors a large number of times. This native code uses the MPL message-passing libraries on the SP and TCP sockets on the Sparc. Every attempt was made to optimize performance: for example, blocking receives were used on the SP. Figure 2 summarizes the one-way communication costs measured with this code. Notice that these results represent a best case for the native communication library, since data is not copied. In realistic situations, copy operations will often be required on the sending and/or receiving processor to move data to and from complex data structures.

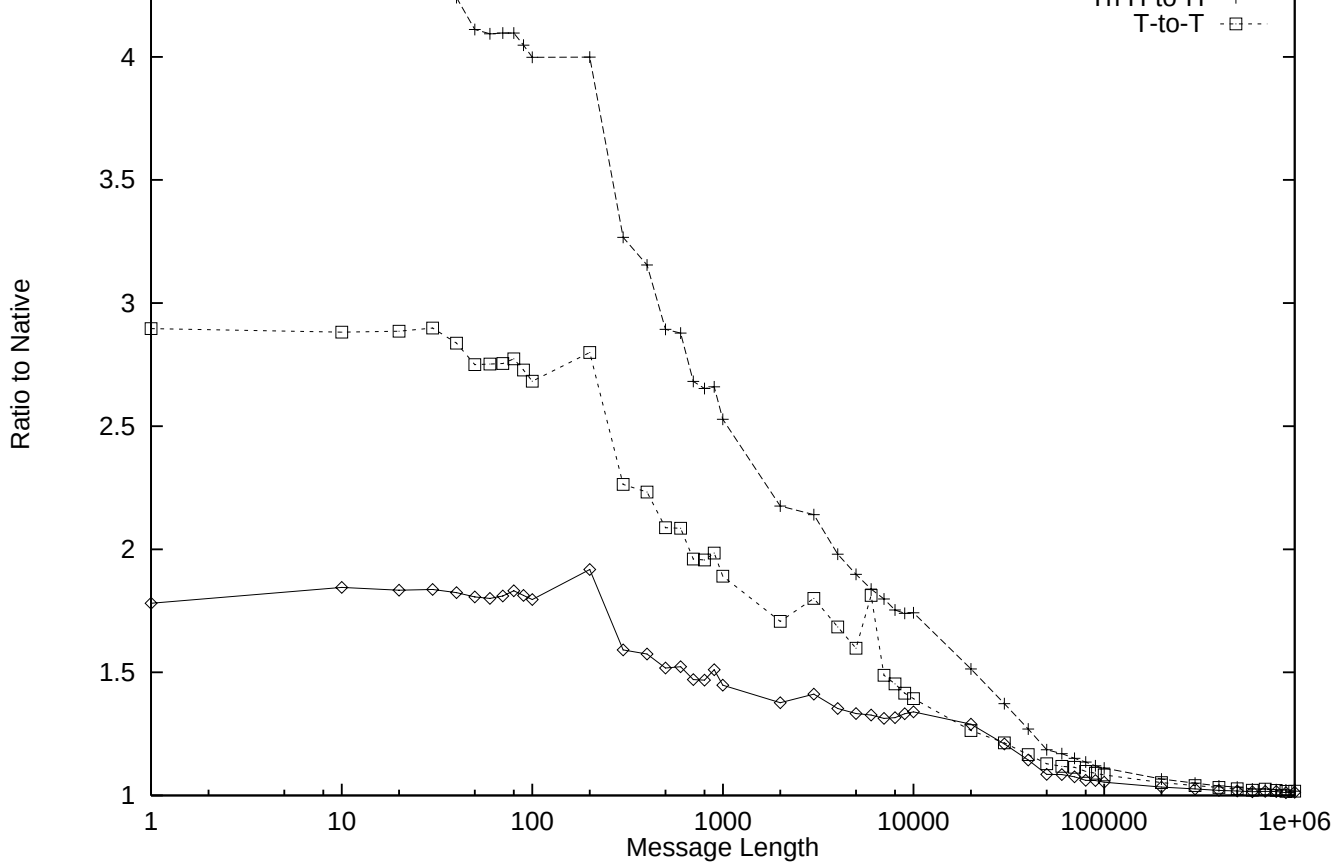

Figure 3: *One-way message latency on the SP as a function of message size, relative to latencies achieved using low-level protocols, for three different cases, explained in the text: nonthreaded handler-to-handler, threaded handler-to-threaded handler, and thread-to-thread.*

5.2 Handler-to-Handler Communication

The first Nexus benchmark is a nonthreaded handler-to-handler test case, *H-to-H*, in which the send/receive pair in the native test case is replaced by an RSR to the remote node. The RSR handler that executes on the remote node invokes an RSR back on the originating node. This code typifies the behavior of a program placing data in a remote location, as no new threads are created to execute handlers. The line labeled *H-to-H* in Figures 3 and 4 shows the overhead associated with this code, expressed as a ratio of execution time to that of the native communication test.

To provide greater insight into the costs of using the RSR mechanism, we examine its performance in detail on the SP. For a zero-length message, native one-way latency is 44 μsec. *H-to-H* takes 79 μsec: 35 μsec (80%) longer. As messages get larger, the time spent in actual communication increases, and the *H-to-H* overhead decreases. For 100,000 byte messages, the overhead is 327 μsec (10%). Clearly, the overhead associated with the RSR mechanism is significant only for small messages, and then is not large in absolute terms.

As *H-to-H* overhead is highest for zero-length messages, we focus on this case and attempt to identify the nature and costs of the additional operations performed on the *H-to-H* critical path. We first say a few words about the techniques used to implement RSRs. As discussed in Section 4, a nonthreaded RSR handler can be executed in an existing thread. In this experiment, the RSR handler executes in a communication thread created by the Nexus implementation for

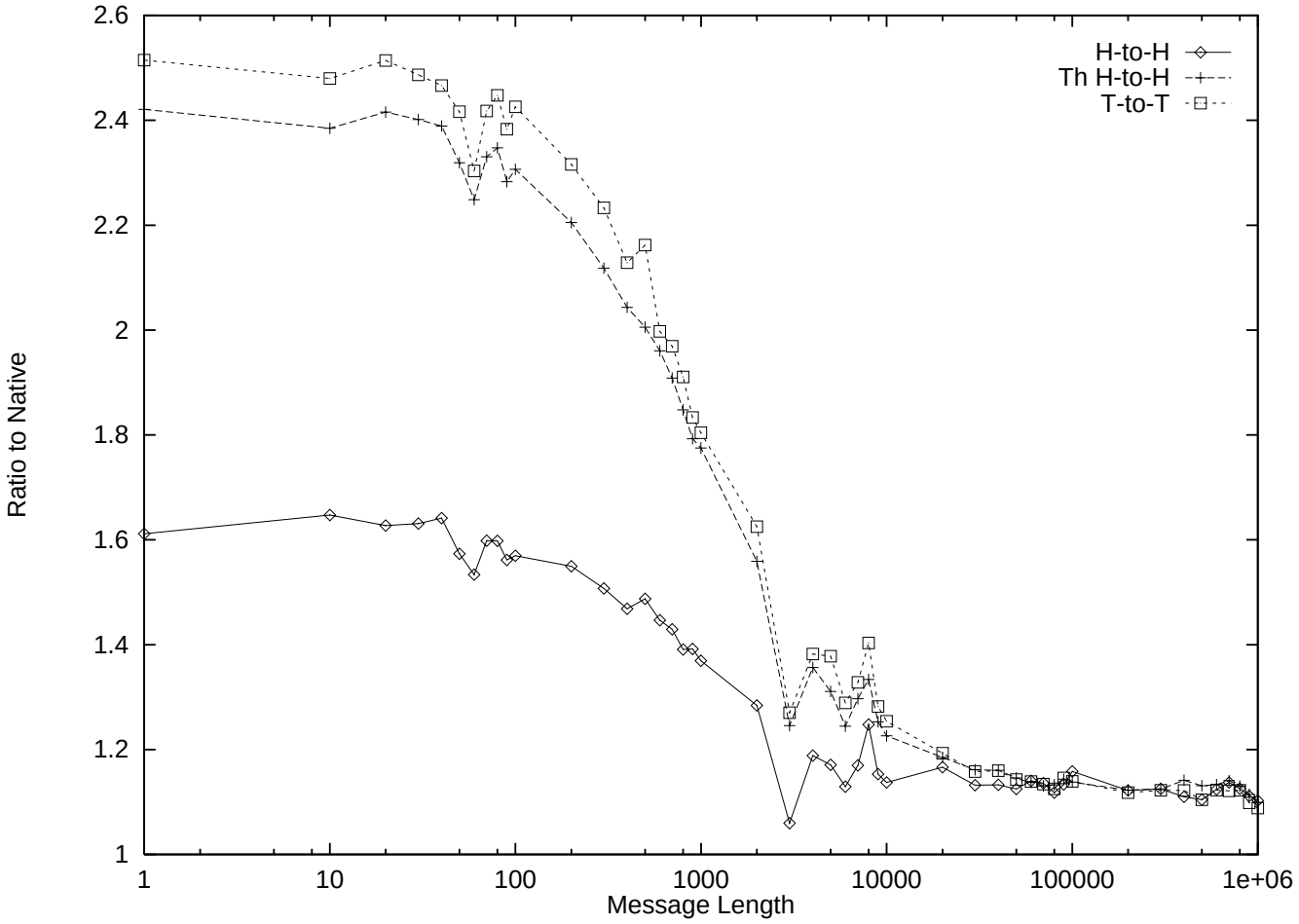

Figure 4: *One-way message latency on the Sparc as a function of message size, relative to latencies achieved using low-level protocols, for three different cases, explained in the text: non-threaded handler-to-handler, threaded handler-to-threaded handler, and thread-to-thread.*

this purpose. Hence, *H-to-H* incurs no thread creation or scheduling costs. Overhead consists of additional costs initiating and transferring the RSR to the remote processor, and additional costs incurred receiving the RSR and dispatching the handler. These costs are explained below.

We also need to understand the protocol used to transfer the data associated with an RSR. We consider the case in which a high-level language compiler has indicated that data is to be placed into the address pointed to by the GP. This allows the Nexus implementation to avoid additional copy operations that would otherwise be necessary on systems such as the SP that do not allow user-level code to read directly from a communication channel. For generality, Nexus also allows an RSR handler to place data in other locations. This would be required, for example, if the data received was to be stored in a complex data structure such as a tree. However, note that in this situation a message-passing library must also copy data: so, in absolute terms, the Nexus abstractions do not incur additional overhead.

If the RSR data is small, a single message is generated containing both an RSR header and the data. If RSR data is large (>2500 bytes on the SP), the header is sent in a single message and the data in a subsequent message. This two-phase protocol keeps overheads small for all message sizes. For small messages, we incur the cost of an additional copy on the sending and receiving side; for large messages, we incur the cost of one additional communication. Note that all receive operations can be posted ahead; this avoids copy operations from system buffers on some machines.

We now examine the overhead incurred by the processor that generates an RSR. A communication operation in a high-level program is compiled to an `Init_RSR` call followed by a series of `Put` calls, and terminated by a `Send_RSR` call. The `Init_RSR` call obtains an RSR header (20 bytes) and a default data buffer from a free list. Since we are operating in a multithreaded environment, the free list must be protected by a lock (1.1 μsec). Once the buffer is obtained, a small number of copies must be performed to initialize the header. `Init_RSR` takes about 5 μsec, of which about 2.5 μsec is spent moving data into the header. We note that the lock is not strictly necessary. A minor modification of the `Init_RSR` call interface would allow the compiler to allocate message headers and buffers without locking.

On non-zero length messages, `Put` routines are used to transfer data into the RSR buffer. These routines are constructed so as to prevent any copying additional to that performed by the underlying communication system. Finally, a `Send_RSR` call is made to dispatch the RSR. This call costs 4 μsec more than the corresponding native send. Part of this overhead is because a lock must be used to protect the nonthread-safe SP communication library.

Overhead is also incurred on the processor to which the RSR is sent. As discussed above, the RSR is serviced by a dedicated communication thread. In *H-to-H*, this is the only thread executing. The communication thread posts a receive and then uses a blocking "message done" function to detect the arrival of an RSR. The use of the message done function rather than the blocking receive used in the native code incurs an overhead of 5 μsec. The RSR header is decoded to identify the handler function that is to be executed, the handler function pointer is retrieved, and the handler is invoked. On completion of the handler function, the communication thread again polls for new requests; however, since the handler has already issued an RSR back to the initial node, this poll is not in the critical path. This RSR dispatch process takes about 7 μsec on the SP. The dispatch of the handler is protected by a lock until the handler function is called. In addition, calls to the POSIX functions "get specific" and "set specific" (1.2 μsec) are required to update the context pointer that Nexus associates with each

thread.

In summary, we can account for an overhead of 9 μsec on the sending side and 12 μsec on the receiving side, for a total of 21 out of an observed difference of 33 μsec. We suspect, but have not been able to demonstrate, that the unaccounted-for overhead is due in part to cache effects. The native code consists of a tight loop in which a send call is immediately followed by a receive. In more realistic situations, data references between the send and receive would likely degrade cache performance, increasing native communication times.

A similar analysis can be performed on the Sparc. The overhead is large: about 700 μsec for a zero-length message. About 124 μsec is due to an extra read (for the header: 62 μsec), buffer management costs (41 μsec), and the handler call (21 μsec). The rest is due to specific attributes of the *H-to-H* test case that should not apply in realistic programs. Because Solaris supports blocking of a thread in a communication library, the Nexus implementation on the Sparc can replace the polling loop in the communication thread with a blocking socket read. (In general, a Nexus thread must block rather than poll, to avoid taking resources from other concurrently executing threads.) In contrast, the native test case uses a polling loop built on *nonblocking* reads. In *H-to-H*, only one thread active at is a time, and hence a blocking read results in a context switch from the reading process to some other process. As a result, context switches increase from 1.1 to 1.8 per round trip, and performance decreases accordingly. Realistic programs typically have other threads that can continue execution when the communication thread blocks, and furthermore perform less frequent communication. Hence, we would not expect to encounter this problem in most practical situations.

5.3 Thread-to-Thread Communication

The second Nexus benchmark generalizes *H-to-H* so that data is passed between two computation threads. The RSR handler uses a condition variable to enable a blocked thread which issues an RSR back to the originating processor. This RSR in turn enables a blocked thread on that processor. This code typifies the behavior of a program in which two long-lived threads exchange data using a message-passing model. The results of this experiment are labeled *T-to-T* in Figures 3 and 4.

The cost of thread-to-thread communication is higher than the cost of hander-to-handler communication. For a zero-length message on the SP, the difference is about 50 μsec. This additional cost results from synchronization using condition variables, switching between the communication thread and the thread that issues the reply RSR (a yield costs 5.5 μsec), and communication thread operations that were not in the critical path in *H-to-H*. In particular, polling for new RSR requests is now in the critical path, since the reply RSR is issued from the thread and not the handler as in *H-to-H*.

5.4 Remote Thread Creation

In the previous two experiments, we examined the cost of RSR-based communication between two preexisting threads. The third benchmark examines the cost of combining thread creation with RSR execution. *H-to-H* is modified to use a threaded handler. The resulting program typifies the behavior of programs that use RSRs to implement long-lived remote computations. The results of this experiment are labeled *Th H-to-H* in Figures 3 and 4.

For a zero-length message on the SP, one-way latency in this experiment is about 60 μsec greater than in *T-to-T*. The principal additional cost incurred in the critical path is the creation of a new thread to execute the RSR handler. On the SP, thread creation costs at least 32 μsec. Presumably this cost could be reduced by using a lighter-weight thread package.

5.5 Discussion

The results reported in this section suggest that the basic Nexus mechanisms can be implemented with reasonable efficiency on a range of commodity systems. With a moderate level of optimization, we achieve, in a worst-case scenario, overheads relative to low-level message passing of a few tens of microseconds for small messages, and a few percent for large messages. On multicomputers, except for small messages, the overhead is significantly less than is incurred by introducing a single additional copy on either the sending or receiving side. The results also provide insights into where optimization efforts should be directed in future systems. For example, high thread creation costs have a significant impact on threaded handler performance.

The significance of these results is that a program that uses Nexus mechanisms to perform message-passing operations will be only slightly slower than an equivalent *single-threaded* program that uses low-level message-passing libraries. *Multithreaded* programs that use other techniques to provide thread-to-thread communication will certainly incur most of the overhead measured for the *T-to-T* case, because of the need to first receive and then deliver messages to specific threads. These programs may incur other overheads also, depending on the tool used.

6 Conclusions

We have proposed an approach to the integration of multithreading and communication based on global pointers and remote service requests. We have explained how these constructs can be used to develop a wide variety of parallel program structures, and how they can be implemented on a variety of hardware and software platforms. We have also evaluated costs associated with implementations of these mechanisms on several commodity systems.

Threads, global pointers, and remote service requests are fundamental components of the Nexus runtime system, a parallel library that also supports dynamic management of computational resources and address spaces. Nexus has been implemented on a wide range of parallel and distributed computer systems, and is used as a runtime library by numerous parallel language and parallel library projects. These experiences demonstrate the robustness of the mechanisms and implementation.

In future work, we will investigate refinements to the GP and RSR mechanisms. For example, an expanded interface can allow certain buffer allocation, initialization, and locking operations to be avoided or amortized over multiple RSRs. The size of the header can be reduced if a linker can provide unique handler identifiers. In TCP/IP protocol modules, we can eliminate one of two reads currently used when receiving messages, if we perform some simple buffering. We are also investigating extensions to the underlying protocol architecture, with the goal of supporting additional low-level protocols (e.g., multicast and unreliable unicast) and higher-level functionality (e.g., security).

More significant performance improvements can be achieved by modifications to the underlying system software or hardware. A thread-safe communication library can reduce the need

for locks, as can a tighter integration of messaging, threading, and other system functions. The cost of thread creation and scheduling can be reduced by using lighter-weight threads [8, 21]. It also appears straightforward to map certain classes of RSR to hardware `get` and `put` operations when these are available. Hence, we anticipate future developments in both software and hardware tending to reduce the overhead associated with Nexus mechanisms.

Acknowledgments

We are grateful to Hubertus Franke, John Garnett, Jonathan Geisler, David Kohr, Tal Lancaster, Robert Olson, and James Patton for their input to the Nexus design and implementation. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by the National Science Foundation's Center for Research in Parallel Computation under Contract CCR-8809615.

References

[1] T. Anderson, E. Lazowska, and H. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, 1989.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *IEEE Transactions on Computer Systems*, 9(2):175–198, May 1991.

[3] R. Bhoedjang, T. Rumlhl, R. Hofman, K. Langendoen, and H. Bal. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pages 213–226, September 1993.

[4] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[5] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computing Systems*, 2:39–59, 1984.

[6] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. Symp. on Principles and Practice of Parallel Programming*, pages 207–216. ACM, 1995.

[7] R. Boothe and A. Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. *ACM SIGARCH Computer Architecture News*, 20(2), 1992.

[8] P. Buhr and R. Stroobosscher. The μsystem: Providing light-weight concurrency on shared-memory multiprocessor systems running Unix. *Software Practice and Experience*, pages 929–964, September 1990.

[9] R. Butler and E. Lusk. Monitors, message, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[10] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*, pages 281–313. The MIT Press, 1993.

[11] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[12] B. M. Chapman, P. Mehrotra, J. Van Rosendale, and H. Zima. A software architecture of multidisciplinary applications: Integrating task and data parallelism. In *Proc. CONPAR94-VAPP VI 3rd Intl Conf. on Vector and Parallel Processing, LNCS 854*, pages 664–676. Springer Verlag, September 1994.

[13] A. Chowdappa, A. Skjellum, and N. Doss. Thread-safe message passing with p4 and MPI. Technical Report TR-CS-941025, Computer Science Department and NSF Engineering Research Center, Mississippi State University, 1994.

[14] N.P. Chrisochoides. A unified appoach for static and dynamic load balancing of computations for parallel numerical grid generation. In *Proc. 4th Intl Conf. on Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, 1994.

[15] M. Cristaller, J. Briat, and M. Rivière. ATHAPASCAN-0: Concepts structurants simples pour une programmation parallèle efficace. *Calculateurs Parallèles*, 7(2):173–196, 1995.

[16] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273. ACM Press, 1993.

[17] W. J. Dally et al. The J-Machine: A fine-grain concurrent computer. In *Information Processing 89*, 1989.

[18] T. L. Disz, M. E. Papka, M. Pellegrino, and R. Stevens. Sharing visualization experiences among remote virtual environments. In *International Workshop on High Performance Computing for Computer Graphics and Visualization*, pages 217–237. Springer-Verlag, 1995.

[19] J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. In *Computers in Physics*, April 1993.

[20] N. Suzuki (ed.). *Shared Memory Multiprocessing*. The MIT Press, 1992.

[21] D. Engler, G. Andrews, and D. Lowenthal. Filaments: Efficient support for fine-grained parallelism. Technical Report 93-13, Dept. of Computer Science, U. Arizona, Tuscon, Ariz., 1993.

[22] E. Felton and D. McNamee. Improving the performance of message-passing applications by multithreading. In *Proc. 1992 Scalable High Performance Computing Conf.*, pages 84–89. IEEE Computer Society Press, 1992.

[23] A. Ferrari and V. S. Sunderam. TPVM: Distributed concurrent computing with lightweight processes. Technical Report CSTR-940802, University of Virginia, 1994.

[24] Message Passing Interface Forum. Document for a standard message-passing interface, March 1994. (available from netlib).

[25] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[26] I. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proc. 1st Intl Workshop on Parallel Processing*, pages 457–462. Tata McGraw Hill, 1994.

[27] I. Foster and R. Olson. A guide to parallel and distributed programming in nPerl. Technical report, Argonne National Laboratory, 1995. `http://www.mcs.anl.gov/nexus/nperl/`.

[28] I. Foster and S. Taylor. A compiler approach to scalable concurrent program design. *ACM Transactions on Programming Languages and Systems*, 16(3):577–604, 1994.

[29] H. Franke and M. Rivière. An efficient MPI-threaded implementation for an executive parallel kernel. Technical report, IBM T.J.Watson Research Center, 1995. Internal report.

[30] D. Gannon et al. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of Supercomputing '93*, November 1993.

[31] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Technical Report ANL/MCS-TM-213, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1996.

[32] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1995.

[33] D. Grunwald. A user's guide to AWESIME: An object-oriented parallel programming and sumulation system. Technical Report CU-CS-552-91, Department of Computer Science, University of Colorado at Boulder, 1991.

[34] M. Haines and W. Bohm. An evaluation of software multithreading in a conventional distributed-memory multiprocessor. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 106–113. IEEE Computer Society Press, 1993.

[35] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing '94*, pages 350–359, 1993.

[36] M. Haines, P. Mehrotra, and D. Cronk. Ropes: Support for collective operations among distributed threads. Technical Report 95-36, Institute for Computer Application in Science and Engineering, 1995.

[37] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, January 1993.

[38] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[39] IEEE. IEEE P1003.1c/D10: Draft standard for information technology – portable operating systems interface (POSIX), September 1994.

[40] L. V. Kale, M. Bhandarkar, N. K. Jagathesan, and S. Krishnan. Converse: An interoperable framework for parallel programming. Technical report, Dept. of Computer Science, UIUC, 1994.

[41] R. Pandey. *A Compositional Approach to Concurrent Programming*. PhD thesis, The University of Texas at Austin, August 1995.

[42] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. Technical Report CS-TR-3055, Department of Computer Science, University of Maryland, 1993.

[43] A. Silberschatz, J. Peterson, and P. Galvin. *Operating Systems Concepts*. Addison-Wesley, 1991.

[44] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266. ACM Press, May 1992.

[45] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 1992.

[46] D. A. Wallach, W. C. Hsieh, K. Johnson, M. F. Kaashoek, and W. E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. Technical report, MIT Laboratory for Computer Science, 1995.

[47] M. Young et al. The duality of memory and communication in Mach. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76. ACM Press, 1987.