Ropes: Support for Collective Operations Among Distributed Threads*

Matthew Haines Piyush Mehrotra David Cronk

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Mail Stop 132C
Hampton, VA 23681-0001

Abstract

Lightweight threads are becoming increasingly useful in supporting parallelism and asynchronous control structures in applications and language implementations. Recently, systems have been designed and implemented to support interprocessor communication between lightweight threads so that threads can be exploited in a distributed memory system. Their use, in this setting, has been largely restricted to supporting latency hiding techniques and functional parallelism within a single application. However, to execute data parallel codes independent of other threads in the system, collective operations and relative indexing among threads are required. This paper describes the design of *ropes*: a scoping mechanism for collective operations and relative indexing among threads. We present the design of ropes in the context of the Chant system, and provide performance results evaluating our initial design decisions.

*Research supported by the National Aeronautics and Space Administration under NASA Contract No. NASA-19480, while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23681.

1 Introduction

Lightweight threads are becoming increasingly useful for supporting parallelism and asynchronous events in applications and language implementations. In particular, many recent languages for parallel and distributed computing employ lightweight threads to represent functional parallelism, to overlap computations with communications, or to simplify resource management [1, 7, 12, 14]. In response to this increasing demand for parallel language support, several projects have emerged with the goal of providing standard lightweight thread support [4, 8, 9], and two committees have been formed to establish standard interfaces for such a runtime system [17, 18].

Currently lacking from these systems is the ability to support *collective operations* and *relative indexing* among the threads, which are commonly used in data parallel programs. For example, consider a simple data parallel algorithm for computing the sum over a distributed array (c.f. Figure 1). In this example, each process will compute its local sum and then participate in a global reduction to obtain the total sum. To execute this example as a set of distributed threads in the midst of other thread activity, and without involving the other threads, a scoping mechanism is needed for identifying the threads that will contribute to the global reduction. Ropes provides this mechanism.

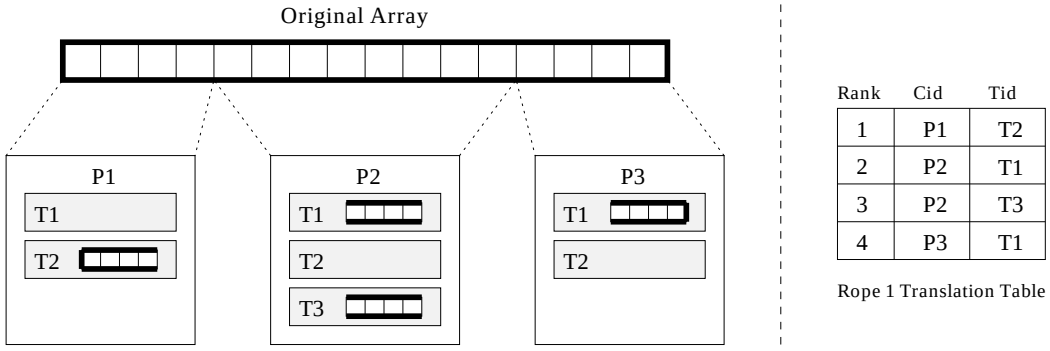

Figure 1: One-dimensional array distributed among four threads in a rope

The key to collective operations is the ability for the programmer (or compiler) to specify the *scope* of the operation; that is, the entities that will be involved in the operation. Collective operations are typically supported at the process level by the underlying communication system [11], or by standard communication interfaces [6, 22]. For example, MPI [6] provides a mechanism for process scoping called *groups*. However, support for grouping threads within processes is not currently supported by either MPI or the new thread-based runtime systems – yet such support is clearly needed if threads are to perform collective operations on a subset of the threads in the system.

Relative indexing allows the programmer to specify spatial relationships among the parallel execution units, which express the natural "neighboring" relationships in data parallel algorithms. Without support for relative indexing among threads, the programmer would be required to assign relative identifiers to threads. Also, with proper support for mapping processes to processors, relative indexing can also be used to optimize performance by ensuring that an algorithm is correctly mapped onto the underlying topology.

In this paper we describe the design and implementation of *ropes*, a mechanism for supporting

collective operations and relative indexing among threads. We describe the design of ropes in the context of Chant [9], a runtime system which supports both intra- and inter-processor communication between lightweight threads in a distributed system. However, the design issues we present are applicable to any thread-based runtime system that supports remote service requests. Our contribution is to provide a detailed examination of the issues that arise in supporting relative indexing and collective operations among lightweight threads. Additionally, we provide an implementation of ropes atop Chant and report initial performance results evaluating our design decisions.

The remainder of the paper is organized as follows: Section 2 provides background on Chant, a system supporting communication between threads, and Section 3 outlines the design of ropes within the Chant system. Section 4 addresses the issues of interfacing with ropes, particularly from the perspective of a data parallel compiler. Section 5 presents performance results evaluating our initial implementation. Section 6 outlines related research projects, and we conclude in Section 7.

2 Chant

The POSIX committee has recently established a standard for the interface and functionality of lightweight threads within an operating system process, called *pthreads* [10]. Since threads are defined within the context of a process, they share a single address space, and communication among threads is only defined in terms of shared memory primitives, such as events and locks. Thus, the interaction of pthreads in a distributed environment is undefined. Likewise, the Message Passing Interface Forum (MPI) has recently established a standard for communication between processes [6]. Although various extensions to the standard have already been proposed [19, 20], communication between lightweight threads within processes has yet to be supported by MPI. Therefore, Chant was designed to provide a simple mechanism for combining lightweight threads with interprocessor communication.

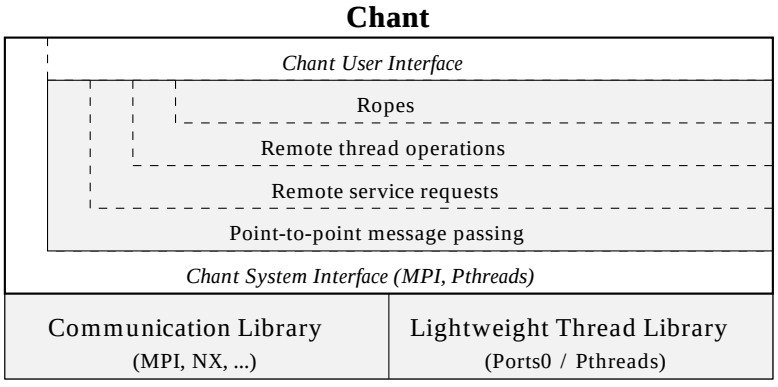

Figure 2: Chant runtime layers and interfaces

Chant [9]* is designed as a layered system (as shown in Figure 2), where efficient point-to-point communication provides the basis for implementing remote service requests and, in turn, remote thread operations. Chant relies on a *system interface* to achieve a high degree of portability, where

*For more information on Chant, please refer to `http://www.icase.edu/~haines/html/chant.html`

the underlying thread and communication systems are ports0[†] and MPI, respectively. Next, Chant supports *point-to-point communication* (i.e., send/recv) between any two threads in the system by utilizing the underlying message passing system (MPI). Issues to be addressed at this level include naming global threads in the system, avoiding intermediate copies for message buffers, and efficient polling for outstanding messages. Chant uses the concept of a *context* to represent an addressing space within a processor, where contexts represent a linear ordering of processes in the system as maintained by the underlying communication system (e.g., MPI uses `rank` in `MPI_COMM_WORLD`). Global threads within Chant are therefore identified using the doublet <`context_id,thread_id`>. Atop efficient point-to-point message passing, Chant supports *remote service requests* by instantiating, in each context, a service thread which is responsible for handling all incoming remote service requests (asynchronous messages) and delivering any necessary replies. Using the remote service request mechanism, Chant can easily support *remote thread operations*, such as remote thread create, by invoking the specified thread request on the desired processor and, possibly, by adding some software "glue" to make it work. Finally, Chant provides a *user interface* that is an extension of the *pthreads* standard, where access to each of the underlying layers can be made directly or indirectly. Thus it is still possible to access the underlying MPI or pthreads interfaces from within a Chant thread.

3 Design

3.1 Requirements

A system for implementing collections among a set of threads (i.e., ropes) must satisfy the following requirements:

1. The collections are entities whose members can span contexts, and thus their identifiers must be unique within the system.

2. Each collection must keep track of its constituent contexts and threads, and operations to add and delete from this list must be performed atomically.

3. Thread ranks within a collection must be unique so that there exists a one-to-one mapping between the thread identifier with respect to the context (global thread id) and the thread identifier with respect to the rope (relative index).

We now describe the design of such a collection of threads, called ropes, which satisfies these requirements. Our design builds upon the existing layers of Chant (Section 2), maintaining the goals of portability, efficiency, and utility.

3.2 Rope Servers

The requirements listed in Section 3.1 are typically satisfied by having a centralized name server responsible for allotting rope identifiers and for performing atomic updates to the internal data structures. Distributed algorithms for name servers [15] and atomic operations [13] are well known,

[†]Ports0 is a subset of the pthreads interface designed by the PORTS consortium. For more information, refer to `http://www.cs.uoregon.edu:80/paracomp/ports/`

but their added overhead and implementation complexity are often unwarranted in an initial design. However, a centralized solution for naming and updating ropes will certainly cause hot-spots. Therefore, our initial design is a two-level approach, derived from the idea of two-level page management schemes for distributed shared memory systems [3], that allows the user to control the contention among the servers by dividing the work between two types of centralized servers:

1. a single, global *name server* used to allot identifiers for new ropes, and
2. a *rope server* associated with each rope that is responsible for all modifications and requests pertaining to that rope.

3.3 Rope Creation

A rope is a set of threads that defines a scope for collective operations, and creating a rope is tantamount to specifying this set of threads. In some instances, it may be useful to create a set of *new* threads which will define a rope. For example, a host program may create a set of new threads as node programs for a data parallel computation, and the new threads are to employ collective operations and should thus comprise a rope. This would also be the model employed by a data parallel compiler. In other instances, it may be useful to add *existing* threads into an extant rope. For example, a threaded system may start with a single thread on each processor, and each of these threads may add themselves into a rope representing the global set of threads. If all newly-created threads also added to this rope, then this would be the thread-level equivalent to `MPI_COMM_WORLD`. Therefore, the rope creation mechanism must be capable of both creating new threads to comprise a rope, and adding existing threads to a rope. This is accomplished by separating the tasks of creating a rope and specifying membership (i.e. joining) a rope.

Creating a rope is done using the `rope_create` call, resulting in a message being sent from the source thread to the global name server, which returns the next available rope identifier. To avoid further messages and a more complicated protocol, the context of the calling thread is designated as the rope server for the new rope. Thus, distribution of the rope servers is accomplished by having different threads invoke the `rope_create` routine, which is under direct control of the user. The global name server keeps track of which context is the server for each active rope so that any thread in the system can always find out who the server is for a particular rope (via the global name server).

A newly-created rope is initially empty, and the user can use the following two mechanisms for specifying membership:

1. `rope_addnew`, which creates a specified number of threads on a set of contexts and adds them to a rope; and
2. `rope_addself`, which adds the calling thread to a rope.

In the case of `rope_addnew`, the calling thread sends a message to the server for the specified rope, indicating how many threads are to be created and on which contexts. The rope server assigns the ranks and sends messages to the specified contexts informing them to create the required number of local threads, and to update their local rope translation tables with the ranks of the new threads. The individual contexts send the thread identifiers for the new threads back to the rope server so

that the rope server can update the master copy of the translation table. If the rope is using the strong consistency model (see Section 3.4), then an image of the new rope translation table is propagated to all member contexts. The `rope_addself` is a special-case of `rope_addnew`, in which the step of creating the threads is simply bypassed.

3.4 Relative Indexing

Spatial relationships play an important role in data parallel algorithms, and most communication systems provide a linear ordering of the participating processes, which allows for *relative indexing* of the processes independent of their actual system address. In addition to supporting collective operations, ropes provide a relative ordering for a set of threads that is independent of their actual global address. Thus we say that each thread within a rope is assigned a unique *rank*, starting from zero and linearly increasing. This makes it possible to send a message from thread i to thread $i+1$ within a rope, without regard to the physical location of those threads. Spatial ordering can also be used to gain performance by exploiting the underlying connectivity of the architecture. However, for this to happen the user must be able to specify a mapping of threads to processes (allowed in Chant) and processes to processors (currently *not* allowed in MPI).

To support relative indexing, the system must provide a one-to-one mapping between the rank within a rope (<`rope_id,rank`>) and the global address of a thread (<`context_id,thread_id`>)[‡]. This is accomplished via a rope translation table (see Figure 1) to store and retrieve this mapping information. If the translation table is kept in a centralized location, then remote references would be necessary for translating all relative indices, which would be prohibitively expensive. Therefore we replicate this information and keep a copy of the table on each participating context for the rope. Figure 3 depicts the data structure for the local rope list.

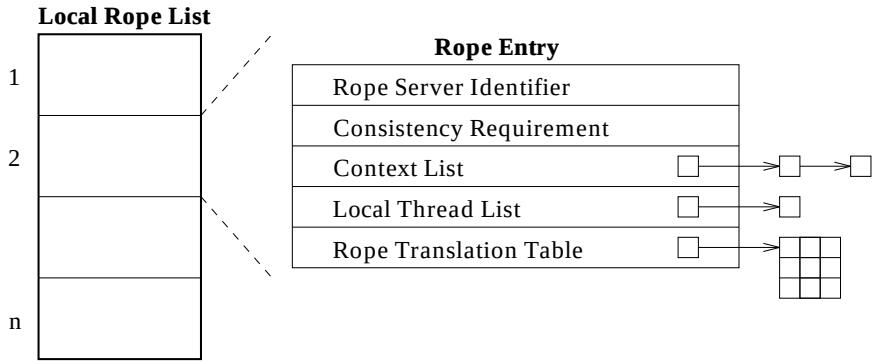

Figure 3: Data structure for local rope list

Again, borrowing from earlier work in area of page coherence for distributed shared memory systems [2], we adopt two options for keeping the distributed translation tables consistent: new information is broadcast so that all tables are kept up-to-date at all times (*strong consistency*), or tables are allowed to remain out-of-date until a reference for a thread is generated, causing the information to be retrieved and stored (cached) in the local table (*weak consistency*). If each thread in a rope communicates with only a small number of other threads in the rope, then the

[‡]As defined by Chant, each process is assigned a unique context identifier, and all threads are given unique thread id's within a context. Therefore, a global thread is identified by this doublet.

weak consistency model should result in better performance, since the creation cost is so much less. If, on the other hand, each thread in a rope will communicate with many other threads in the rope, the strong consistency model should result in better performance. Determining the crossover point for a given application is an open question depending on the overheads of the two approaches (see Section 5). Therefore, the system supports both strong and weak consistency on a per-rope basis by providing an argument to the `rope_create` routine to specify the consistency requirement.

To translate a relative index into a Chant global thread identifier[§], the following steps are taken:

1. If the local rope table does not have an entry for the given rope identifier (i.e., the calling thread is in a context which is not a member of the specified rope), a message is sent to the server for the rope (via the global name server) requesting that the translation table for this rope be returned.

2. The translation table for the rope is accessed to determine if the specified rank has a valid entry. If not, then either it is an error (strong consistency) or the entry is requested from the rope server (weak consistency), returned, and cached for future requests.

3.5 Collective Operations

MPI provides the `group` facility for specifying which processes will participate in a collective operation, and ropes extends this idea to the thread level. To do this, each context participating in a rope must know the other contexts in the rope as well as the list of local threads in the rope, and so this information is maintained for each rope in a rope table (refer to Figure 3).

In order to take advantage of system-specific optimizations for collective operations among processors, all collective operations among threads are performed in two steps: at the thread level and at the context level. For example, consider the `rope_barrier` operation, which performs a barrier synchronization among all threads in a rope. The barrier is performed first at the thread level within a context, and then at the context level, as described by the following algorithm:

1. Each thread, upon executing the barrier command, will send a message to an accumulator thread for that context, which will accumulate the count for the number of messages received. After sending the message, the calling thread is blocked on an appropriate event.

2. After the local accumulator thread has collected the number of barrier messages equal to the number of threads in the rope on this context (this information is stored in the rope table), a message is sent to the rope server for this rope. The accumulator thread then waits for a reply from the rope server.

3. When the rope server has collected a message from each context in the thread, a message is returned to the accumulator threads on the participating contexts, informing them that the barrier is complete.

4. The accumulator threads then triggers the events for the local waiting threads, thus completing the barrier.

[§] <rope_id, rank> → <context_id, thread_id>

Ideally, we would like to utilize the context-level primitives from MPI, such as `MPI_BARRIER`, for replacing steps 2 and 3 in our algorithm. However, the `MPI_BARRIER` call invoked by the local accumulator threads would block the entire process, including any other threads in that context not related to the rope, until all participating contexts had invoked the `MPI_BARRIER` call. This would remove one of the key features of a multi-threaded system: the ability to overlap useful computation (in the form of ready, waiting threads) with long-latency, blocking operations. As a result, our design does not use the `MPI_BARRIER` call, but rather a simple message-combining scheme that allows other ready threads to execute while the barrier operation proceeds. Whenever possible, we utilize the MPI collective operations, and should the MPI committee see fit to extend the standard with a non-blocking barrier operation, we would certainly incorporate it into the design as mentioned.

Other collective communication operations, such as reduction functions, can be implemented in a similar two-level fashion.

3.6 Co-scheduling

The concept of co-scheduling (or gang scheduling) was first introduced in [16], and refers to the idea of simultaneously scheduling all related processes of a parallel program on a set of processors. For machines that are time-shared (as opposed to space-shared), this is a realistic concern, since many parallel algorithms are tightly coupled with respect to communication phases. Allowing the processes to become out-of-phase can seriously degrade the performance of the algorithm.

Ropes provide the capability for defining membership in a set that can be exploited by a thread-level gang scheduling mechanism. However, it remains unclear whether the overheads of co-scheduling are justified given the fine granularity of threads (as opposed to processes). Also, co-scheduling would require modification of the underlying thread-level scheduler, which violates the portability goal of Chant, since access to the scheduler is not allowed by pthreads. This remains an area for future investigation.

4 Interfacing with Ropes

In this section we address the issues of interfacing with ropes from the perspective of a data parallel compiler targeting a multithreaded system, such as Chant. For reference, Figure 9 gives the interface for the rope calls as currently implemented in Chant.

A primary goal of ropes is to allow a data parallel compiler, such as an HPF compiler, to easily produce data parallel code that is targeted for a thread-based implementation. To examine the issues involved, let us consider the HPF fragment in Figure 4.

Here, we have an array, A, distributed by block across a set of processors. The number of processors is to be determined externally at runtime. The `forall` statement shifts the array one element to the left. The scalar variable, X, is assumed to replicated, and thus the value of A(1) has to be broadcast to all other processors.

Most HPF compilers would convert the above code into SPMD code to be executed by a context on each processor. A typical version of the code, using Intel's NX communication calls, is shown in Figure 10.

```
!HPF$ processors P(number_of_processors())
      real A(N)
      real X
!HPF$ distribute A(block)
      ...
      forall (I=1,N-1) A(I) = A(I+1)

      X = A(1)
      ...
```

Figure 4: HPF program fragment

In the code in Figure 10, each context determines the total number of contexts participating in the execution and allocates enough memory for the local portion of the array A along with an overlap area (in this case 1) to accommodate the required non-local data. The forall statement is translated into a strip-mined do loop such that each context loops over its local portion of the array. Before the loop, each context (except context 0) communicates its left boundary element to its neighbor on the left and then receives the value sent by its neighbor and places it in the overlap area. The assignment, on the other hand, turns into a broadcast from the context owning element $A(1)$ to all other contexts.

Targeting a thread-based runtime system would require that the compiler to generate code for threads rather than processes. If support for ropes is not provided, it becomes the compiler's responsibility to generate code which keeps track of all the threads participating in the data parallel computation including the translation table required for relative indexing. Thus, before a communication, the thread would have to determine the global thread id of the thread it is communicating with. Similarly for collective communications, such as the broadcast in the above case, the compiler would have to generate code to multicast to the set of threads participating in the execution.

```
! Initialization code executed in the lead context
      call pthread_rope_create(rope_id, WEAK)
      call pthread_rope_addnew(rope_id, NULL, UserF, NULL, 0, P, ALL, 1)
      call pthread_rope_join( rope_id )
```

Figure 5: Sample initialization code for ropes

The design of ropes, as described here, provides runtime support for the translation table and the multi-casting in the context of a multithreaded environment. The code produced by a compiler targeting such an interface would have two parts. First, the rope would have to be initialized by creating the appropriate threads, and then the user code would be executed as an SPMD function in each of the threads. Part of the initialization code is shown in Figure 5.

We assume here that the P contexts participating in the execution have been already setup.

The lead context then creates a rope with weak consistency and adds one thread per context in all the contexts. The lead context then waits for the rope to exit before continuing on. The user code is executed by each thread in the rope using the subroutine *UserF* is depicted in Figure 11.

In comparing the two compiler-generated codes (Figure 10 and Figure 11), we see that the differences are minor. The ropes based code has some extra setup code. Also the individual thread code has to determine its rope before it starts execution and that the message send and broadcast have to provide an extra rope id. Each thread also calls `pthread_rope_exit` at the end of its execution so that the lead context can be notified when the data parallel computation has finished. Overall, an HPF compiler has to put in a few extra calls to runtime routines to target a ropes interface. This offers the advantage of re-targeting a data parallel compiler for a thread-based runtime system with little modification.

5 Experimental Results

Having described the design for our ropes implementation, we now examine the performance of creating a rope, adding new threads to a rope, using relative indexing to send messages between threads given their rope and rank identifiers (as opposed to context, thread identifiers), and performing collective operations. We measure the results under both strong and weak consistency models, and all measurements are performed atop Chant on an Intel Paragon.

5.1 Rope Creation

Creating a rope results in a remote service request being sent to the global name server, and its reply, which requires about 375 μs. After the rope server has returned the new rope identifier to use, some data structures are initialized and the rope creation is complete. If the number of `rope_create` operations being executed on all contexts in the system at the same time is one, then the creation time is independent of the number of contexts; otherwise, contention for the global rope server will degrade rope creation time depending on the number of simultaneous `rope_create` calls.

Adding new threads to a rope, using the `rope_addnew` call, requires sending a message to the rope server, indicating how many threads are to be created on the specified list of contexts. The rope server must then broadcast a request to those contexts, informing them to create the new threads. After creating the threads, the participating contexts send a message back to the rope server, detailing the thread identifiers of the new threads so that the rope server can complete the rope translation table. Finally, if the consistency mode for the rope is `STRONG`, then the rope server must broadcast the new rope table to the participating contexts. Therefore, the total number of message exchanges required to add threads on N contexts is $2N+1$ for weak consistency and $3N+1$ for strong consistency. Figure 6 depicts the execution times for `rope_addnew`, where 4 new threads are created on $2-32$ contexts and added to an existing rope.

5.2 Relative Indexing

Figure 7 depicts the time (averaged over 10 runs) required to exchange a message (i.e., send/receive pair) on the Intel Paragon using four different mechanisms for message passing:

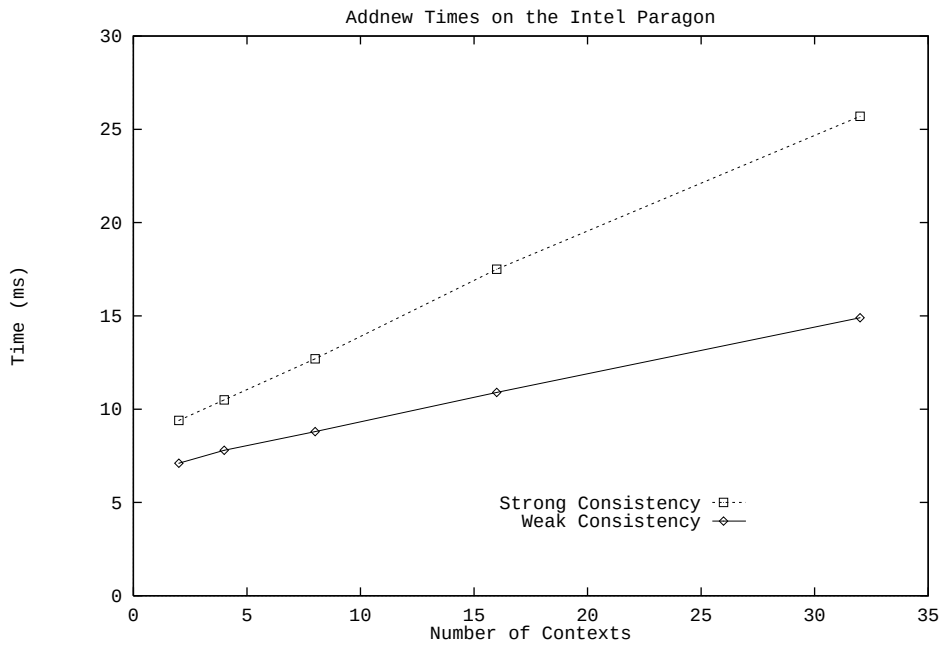

Figure 6: Execution times for `rope_addnew`, 4 threads per context

1. The bottom line corresponds to MPI communication directly between processes.

2. The next line represents Chant thread-to-thread communication using global thread identifiers.

3. The next line represents Chant thread-to-thread communication using the relative rank within a thread. Thus, the system must first translate the <`rope, rank`> pair into a global thread identifier, <`context, thread_id`>, and then invoke the normal `chant_send` function. These numbers assume that the rope table entry containing the global thread identifier is valid (either strong consistency or weak consistency already cached).

4. The top line represents Chant thread-to-thread communication using the relative rank within a rope (as with the previous line), but assuming that the rope table entry for this rank is not valid (weak consistency, not cached) and therefore must be fetched from the server for this rope, which is on a different context from the calling thread. Thus, the exchange time is double that of the previous line, accounting for the exchange needed to get the translation information from the rope server. This is the worst-case situation for the `rope_send`.

The results from Figure 7 indicate that relative indexing adds an insignificant overhead to the cost of sending a message between two threads when the translation information is present, and doubles the cost when the translation information must be retrieved from the server.

5.3 Collective Operations

All of the collective operations follow essentially the same algorithm (detailed in Section 3.5), so we will only address the `rope_barrier` operation in this section. Figure 8 gives the execution times

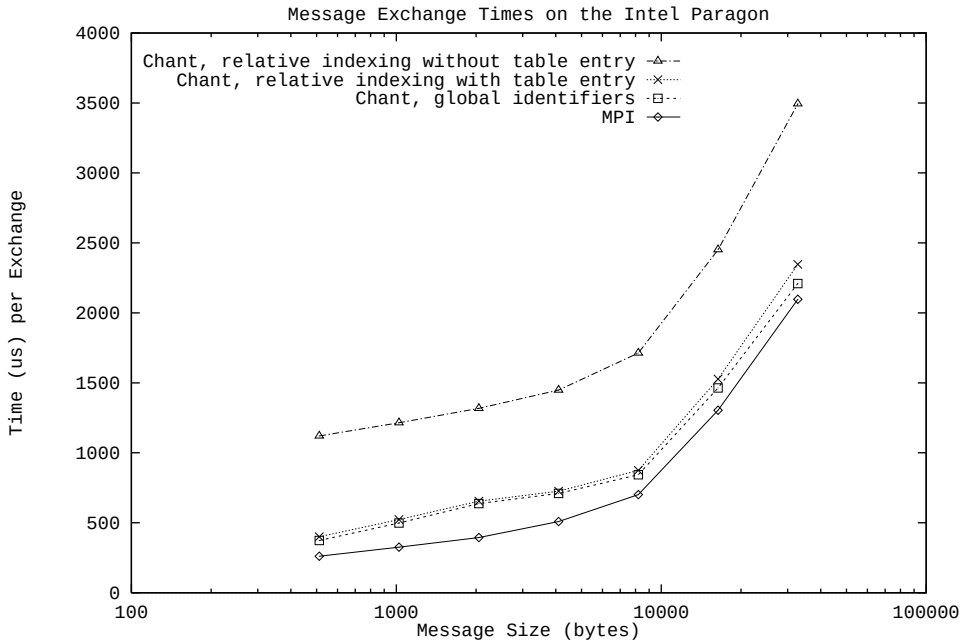

Figure 7: Execution times for point-to-point communication

(μs) for a `rope_barrier` operation on a varying number of contexts (one thread per context), along with the corresponding cost for an MPI barrier on the same number of processes. With only one thread per context, this is a worst-case scenario in comparing with MPI, since our design must account for the possibility of multiple threads per context. Also, as outlined in Section 3.5, we are unable to take advantage of the underlying `MPI_Bcast` routine.

6 Related Research

Other related work includes systems for supporting distributed threads, such as Nexus [8], and Panda [4], though these systems do not currently support the notion of ropes.

The term "rope" was first coined in the pthreads++ system [21], in which a rope is a C++ class that provides support for data parallel execution of a task in a shared memory environment, and later extended to a distributed memory environment.

A rope is the thread-level analogy to the process-level scoping mechanisms provided by most communication packages, such as process groups in MPI [6]. MPI does not currently support the notion of threads as addressable entities within a process, nor the ability to group such threads. However, there has been a lot of recent research activity combining MPI and threads. Besides Chant, which was outlined in Section 2, other projects include [5], which addresses the issue of making MPI (using P4) "thread-safe" with respect to internal, worker threads designed to improve the efficiency of MPI, but not intended to be user-accessible entities (i.e., they cannot execute user code); and [20], which addresses many possible extensions to the MPI standard, including the addition of long-lived threads capable of executing user code. Suggestions for altering the role and functionality of communicators would allow for multiple threads per communicator, thus permitting

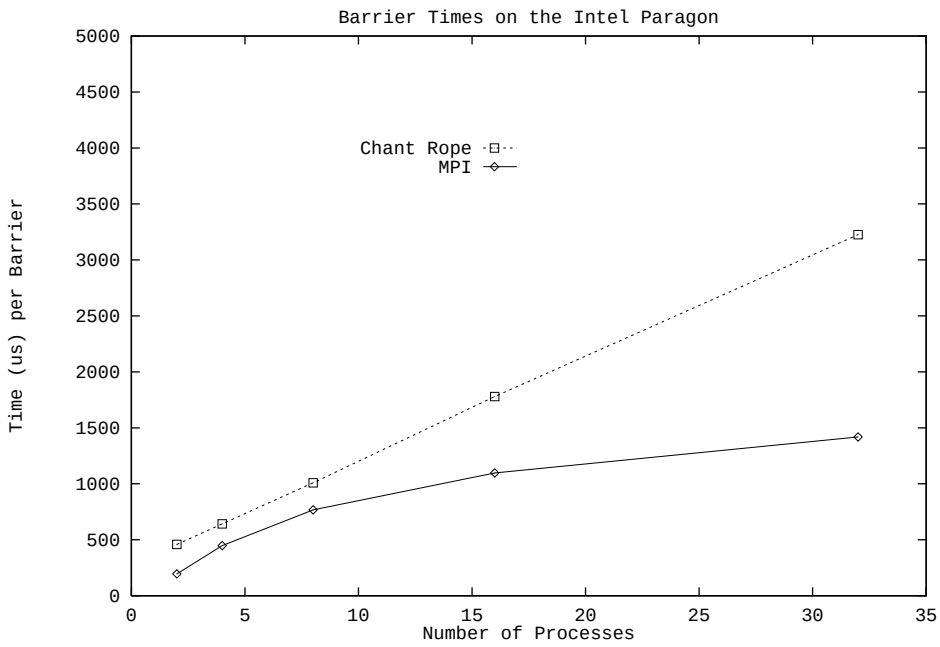

Figure 8: Execution times for barrier

collective operations among the threads.

The contribution of this paper is to provide a design for a thread-level scoping mechanism that is not predicated on MPI constructs and their extensions, but rather on a simple communicating thread model that supports remote service requests. Thus, until the MPI community sees fit to extend the standard for user-accessible, long-lived threads, our approach provides a clean and efficient mechanism for supporting data parallel execution in a multithreaded environment.

7 Conclusions

Recently, several runtime systems have been designed to support interprocessor communication between lightweight threads within a process. Although collective operations and relative indexing are common operations for most message passing systems, support for these operations at the thread level has received little attention.

This paper addresses the issues of supporting collective operations and relative indexing among threads in a distributed memory environment. We provide the design for ropes in the context of the Chant system, where a rope defines the scope of collective operations with respect to threads. Our design builds on the Chant system, which provides point-to-point communication between threads in a distributed memory environment.

We plan to utilize this extension to the Chant runtime system for supporting data parallel codes in a multithreaded environment, and will report on the results of this effort in future work.

References

[1] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[2] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. Technical Report Rice COMP TR89-98, Rice University, November 1989.

[3] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. Technical Report Rice COMP TR90-109, Rice University, March 1990. Appears in Proceedings of ISCA 17.

[4] Raoul Bhoedjang, Tim Rühl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, pages 213–226, San Diego, CA, September 1993.

[5] Aswini K. Chowdappa, Anthony Skjellum, and Nathan E. 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, April 1994.

[6] Message Passing Interface Forum. *Document for a Standard Message Passing Interface*, draft edition, November 1993.

[7] I. T. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. Technical Report MCS-P327-0992 Revision 1, Mathematics and Computer Science Division, Argonne National Laboratory, June 1993.

[8] Ian Foster, Carl Kesselman, Robert Olson, and Steven Tuecke. Nexus: An interoperability layer for parallel and distributed computer systems. Technical Report Version 1.3, Argonne National Labs, December 1993.

[9] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing 94*, pages 350–359, Washington, D.C., November 1994. Also appears as ICASE Technical Report 94-25.

[10] IEEE. *Threads Extension for Portable Operating Systems (Draft 7)*, February 1992.

[11] Intel Corporation, Beaverton, OR. *Paragon OSF/1 User's Guide*, April 1993.

[12] Jenq Kuen Lee and Dennis Gannon. Object oriented parallel programming experiments and results. In *Proceedings of Supercomputing 91*, pages 273–282, Albuquerque, NM, November 1991.

[13] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[14] Piyush Mehrotra and Matthew Haines. An overview of the Opus language and runtime system. In *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computers*, pages 346–360, New York, November 1994. Springer-Verlag Lecture Notes in Computer Science, 892. Also Appears as ICASE Technical Report 94-39.

[15] R.M. Needham. Names. In Sape Mullender, editor, *Distributed systems*, chapter 5, pages 89–101. ACM Press, 1989.

[16] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of Distributed Systems Computing Conference*, pages 22–30, 1982.

[17] Parallel compiler runtime consortium. http://www.extreme.indiana.edu/extreme/pcrc/index.html.

[18] Portable runtime systems (ports) consortium. http://www.cs.uoregon.edu:80/paracomp/ports/.

[19] Anthony Skjellum, Nathan E. Doss, and Kishore Viswanathan. Inter-communicator extensions to MPI in the MPIX (MPI eXtension) library. Technical report, Computer Science Department and NSF Engineering Research Center, Mississippi State University, July 1994. Submitted to ICAE Journal Special Issue on Distributed Computing.

[20] Anthony Skjellum, Nathan E. Doss, Kishore Viswanathan, Aswini Chowdappa, and Purushotham V. Bangalore. Extending the message passing interface (MPI). Technical report, Computer Science Department and NSF Engineering Research Center, Mississippi State University, 1994.

[21] Neelakantan Sundaresan and Linda Lee. An object-oriented thread model for parallel numerical applicaitons. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, pages 291–308, Sunriver, OR, April 1994.

[22] Vaidy Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

`pthread_rope_create`	(	`pthread_rope_t *rid, int coherence_mode );`	*Create a new rope with strong or weak consistency and return its identifier.*
`pthread_rope_addnew`	(	`pthread_rope_t rid, pthread_attr_t attr, const char *function, any_t args, int argsize, int ncontexts, int context_list[], int nthreads );`	*Create* `nthreads` *new threads on each specified context and add them to the specified rope.*
`pthread_rope_addself`	(	`pthread_rope_t rid );`	*Add the calling thread to the specified rope.*
`pthread_rope_send`	(	`any_t buffer, int count, msgdata_t datatype, pthread_rope_t rid, int rank, int tag );`	*Send a message to the thread specified by the relative index <rid,rank>.*
`pthread_rope_barrier`	(	`pthread_rope_t rid );`	*Participate in a barrier for the specified rope.*
`pthread_rope_bcast`	(	`any_t buffer, int count, msgdata_t datatype, pthread_rope_t rid, int root_rank );`	*Participate in a broadcast for the specified rope, originating from the specified thread within the rope.*
`pthread_rope_exit`	(	`pthread_rope_t rid );`	*Initiate exit; rope will terminate when all member threads have invoked this function.*
`pthread_rope_join`	(	`pthread_rope_t rid );`	*Wait for the specified rope to exit.*
`pthread_rope_self`	(	`pthread_rope_t *rid );`	*Return the rope identifier for the calling thread.*
`pthread_rope_rank`	(	`pthread_rope_t rid, int *rank`	*Return the rank of the calling thread in the specified rope.*
`pthread_rope_maxrank`	(	`pthread_rope_t rid, int *rank );`	*Return the maximum rank (number of threads) for the specified rope.*

Figure 9: Ropes interface in Chant

```
      real, allocatable :: A()
      real X
      integer myRank, nContexts, myPid
      integer localSize, uBound, tag

      myRank= mynode()
      myPid = mypid()
      nContexts = numnodes()

      localSize = N / nContexts
      allocate( A(localSize+1) )
          ....

! send and receive boundary data
      if (myRank .ne. 0)
            send(tag, A(1), 4, myRank-1, myPid)
      if (myRank .ne. nContexts - 1)
            recv(tag, A(localSize+1), 4)

      uBound = localSize
      if (myRank .ne. nContexts - 1) uBound = localSize - 1
      do i = 1,  uBound
          A(i) = A(i+1)
      enddo

! broadcast A(1)
      if ( myRank .eq. owner(A(1) ) then
           X = A(1)
           call csend(1000, X, 4, -1, myPid)
      else
           call crecv(1000, X, 4)
      endif
      ....
```

Figure 10: Resulting data parallel code from HPF fragment

```
   subroutine UserF()

      real, allocatable :: A()
      real X
      integer myRank, nThreads, rootRank
      pthread_rope_t  myRope
      integer localSize, uBound, tag, status

      call pthread_rope_self( myRope )
      call pthread_rope_rank( myRope, myRank )
      call pthread_rope_maxrank( myRope, nThreads )

      localSize = N / nThreads
      allocate( A(localSize+1) )
          ....

! send and receive boundary data
      if (myRank .ne. 0)
            pthread_rope_send(A(1), 1, REAL, myRope, myRank-1, tag)
      if (myRank .ne. nThreads - 1)
            pthread_chanter_recv( A(localSize+1), 1, REAL, ANY, tag, status)

      uBound = localSize
      if (myRank .ne. nThreads - 1) uBound = localSize - 1
      do i = 1,  uBound
          A(i) = A(i+1)
      enddo

! broadcast A(1)
      rootRank = owner( A(1))
      if ( myRank .eq. owner( A(1)) ) X = A(1)
      call pthread_rope_bcast( X, 1, REAL, myRope, rootRank )
      ....
      call pthread_rope_exit( myRope )
   end
```

Figure 11: Resulting data parallel code from HPF fragment, with ropes