Data-Parallel Programming in a Multithreaded Environment*

MATTHEW HAINES,[1] PIYUSH MEHROTRA,[2] AND DAVID CRONK[2]

[1]*Computer Science Department, University of Wyoming, Laramie, WY 82071-3682; e-mail: haines@meru.uwyo.edu*
[2]*Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Mail Stop 132C, Hampton, VA 23681-0001; e-mail: {pm,cronk}@icase.edu*

ABSTRACT

Research on programming distributed memory multiprocessors has resulted in a well-understood programming model, namely data-parallel programming. However, data-parallel programming in a multithreaded environment is far less understood. For example, if multiple threads within the same process belong to different data-parallel computations, then the architecture, compiler, or run-time system must ensure that relative indexing and collective operations are handled properly and efficiently. We introduce a run-time-based solution for data-parallel programming in a distributed memory environment that handles the problems of relative indexing and collective communications among thread groups. As a result, the data-parallel programming model can now be executed in a multithreaded environment, such as a system using threads to support both task and data parallelism.

1 INTRODUCTION

Data-parallel programming has emerged as the premiere programming model for distributed memory multiprocessors (DMMPs). This is primarily due to the fact that the parallelism stems from simultaneously performing the same (or similar) operations on different portions of a data set. This implies that the amount of parallelism that can be exploited increases with the size of the data sets. The result is that it is fairly easy to find and exploit data parallelism in most numerical-based applications.

Central to the data-parallel programming model is the idea that, on each processor, some agent (usually a process) is responsible for executing parallel operations on the section of data elements that reside in local memory. To implement this model, each agent is given a relative index from 0 to $n - 1$, assuming there are n agents. The relative index is used to identify each of the participating agents, and to designate which portions of code are to be executed by which agents. For example, consider the following data-parallel code for computing the sum of the elements of a vector that is evenly distributed over n processes:

```
global real sum
real lsum, vector(N)
integer i

. . .
if (myRank .eq. 0) sum = 0.0
lsum = 0.0
```

Received September 1995
Revised March 1996

*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.

Scientific Programming, Vol. 6, pp. 187–200 (1997)
CCC 1058-9244/97/020187-14

```
do i = lower_bound (my_Rank),
upper_bound (my_Rank)
    lsum = lsum + vector(i)
end do

sum = sum + lsum
if (my_Rank .eq. 0) print *, sum
```

If this example, the relative index is used to specify which process will initialize and display the global sum. The statement `sum += lsum` corresponds to a global reduction, in which each process must communicate its local sum value, `lsum`, to a designated lead process, who then combines the intermediate results. This operation demonstrates the second major requirement for implementing data-parallel programs: collective operations. Therefore, an implementation of the data-parallel programming model must provide for both relative indexing and collective operations.

Lightweight, user-level 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 [1–4]. In response to this increasing demand for parallel language support, several projects have emerged with the goal of providing standard lightweight thread support [5–7], and a committee has been formed to establish standard interfaces for such a run-time system [8].

In this article, we describe a run-time solution to the problem of supporting data-parallel execution using lightweight threads as the data-parallel agents. Thus, we describe the ability to provide relative indexing and collective operations among a group of threads, called a rope. Again, consider a simple data-parallel algorithm for computing the sum over a distributed array. 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 "member" threads that will contribute to the global reduction. Also, because the thread agents can (and likely do) have different thread identifiers on each process, a translation scheme is needed to map the local thread identifier for each thread agent to a relative index for the entire data-parallel computation. The concept of ropes as described in this article provides support for these mechanisms.

Relative indexing allows the programmer to specify spatial relationships among the parallel execution units, which express the natural "neighboring" relationships in data-parallel algorithms. Also, with proper support for mapping threads to processes, and processes to processors, relative indexing can be used to optimize performance by ensuring that an algorithm is correctly mapped onto the underlying communication topology.

Collective operations are typically supported at the process level by the underlying communication system [9] or by standard communication interfaces [10, 11]. For example, MPI [10] provides a mechanism for process scoping called process groups. However, support for grouping threads within processes is not currently supported by either MPI or by the new thread-based run-time systems—yet such support is clearly needed if threads are to perform collective operations on a subset of the threads in the system.

We describe our design for data-parallel support among lightweight threads in the context of Chant [7], a run-time system which supports both intra- and interprocessor communication between lightweight threads in a distributed system. However, the design issues we present are applicable to any thread-based run-time system that supports some form of communication between threads. 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 these concepts atop Chant and report some performance results.

The remainder of the article is organized as follows: Section 2 provides background on Chant, a system supporting communication between threads, and Section 3 outlines the design of a run-time approach to support data-parallel programming in a multithreaded environment. 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 Portable Operating System Interfaces for Computer Environments (POSIX) committee has recently established a standard for the interface and functionality of lightweight threads within an operating system process, called *pthreads* [12]. Because threads are defined within the context of a process, they share a single address space, and communication among threads is

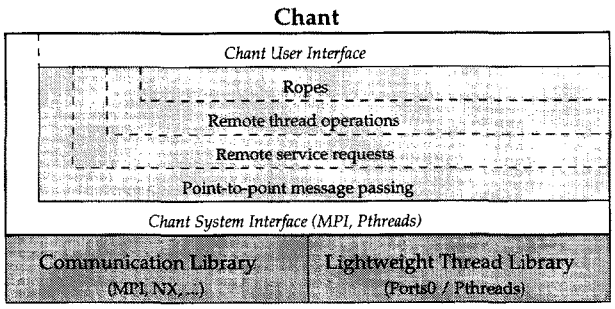


FIGURE 1 Chant run-time layers and interfaces.

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 [10]. Although various extensions to the standard have already been proposed [13, 14], 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.

Chant [7] is designed as a layered system (as shown in Figure 1), where efficient point-to-point communication provides the basis for implementing remote service requests and, in turn, remote thread operations. Each layer is accessible to the user so that the proper amount of support and performance can be obtained. Chant relies on a system interface to achieve a high degree of portability, where the underlying thread and communication systems are pthreads and MPI, respectively.

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) and providing solutions to the problems of naming global threads in the system, avoiding intermediate copies for message buffers, and efficient thread-level polling for outstanding messages. It uses the concept of a context to represent an addressing space within a processor. Chant assumes that a linear ordering of contexts in the system is maintained by the underlying communication system. For example, MPI uses `rank` within `MPI_COMM_WORLD` to linearly order all processes (addressing spaces) within a system. Therefore, threads within Chant are globally 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 support remote thread operations, such as remote thread create, by invoking the specified request on the desired process and, possibly, by adding some software "glue" to make it work. For example, implementing a remote join operation, in which the calling thread will block until the specified thread in a different context has finished its execution, is not as simple as invoking a local join in the remote context. Instead, the exit handler of the desired thread must be modified to send a message to the calling context upon exit. The effect of the message will be to awake the calling thread from its suspended state.

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, this is still possible to access the underlying MPI or pthreads interfaces from within a Chant thread.

3 DESIGN

In this section we outline a run-time-level design that supports data-parallel programming among threads. After reviewing concepts (Section 3.1) and the requirement for such support (Section 3.2), we discuss the issues of rope servers (Section 3.3), rope creation (Section 3.4), relative indexing (Section 3.5), and collective operations (Section 3.6).

3.1 Concepts

First, we give definitions for common terms so as to avoid any confusion over their meaning:

1. A processor is a central processing unit capable of executing instructions. Examples include the MIPS R4400, the Intel Pentium, and the Sun SPARC.
2. A process is a Unix process, complete with its own address space, register set values, and execution stack. It provides a single, sequential execution stream, and time-shares the processor with other processes. The execution order of processes is controlled by the operating system.
3. A context is an address space, and is typically mapped to a Unix process. We use this term to capture the concept of a single addressing space without the connotation of a single execution unit, as is the case with a Unix process.

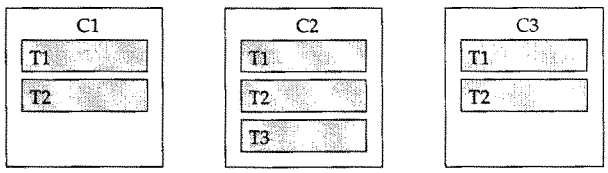


FIGURE 2 An example of three contexts containing seven threads.

4. A thread is a sequential execution stream that executes within the address space of a process. Multiple threads can execute within the same address space, or context. Lightweight, or user-level, threads are controlled explicitly by the user rather than the operating system. Medium-weight, or kernel-level, threads are controlled by the operating system. For the purposes of this article, we are only concerned with light-weight threads.

Figure 2 depicts three contexts (address spaces), labeled C1, C2, and C3. Context C1 has two threads, labeled T1 and T2; context C2 contains three threads labeled T1, T2, and T3; and context C3 contains two threads labeled T1 and T2. Because thread identifiers are local to a context, our global naming scheme of each of these threads is <context_id, thread_id>. Thus, these seven threads are labeled <C1, T1>, <C1, T2>, <C2, T1>, <C2, T2>, <C2, T3>, <C3, T1>, and <C3, T2>.

We define a **rope** to be a collection of threads capable of spanning context boundaries that define a scope for collective operations and support relative indexing (renaming). Figure 3 depicts the same set of contexts and threads as are presented in Figure 2, except that threads <C1, T2>, <C2, T1>, <C2, T3>, and <C3, T1> have been organized into a rope. A rope translation table, which translates between the relative index of a rope member and its global thread identifier, is also given.

3.2 Requirements

A system for implementing collections among distributed threads must satisfy the following requirements:

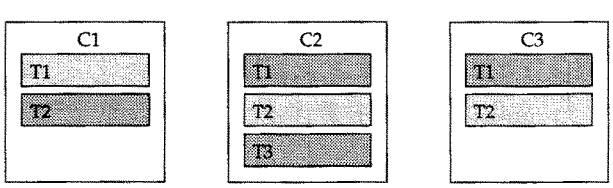
1. The collections are entities whose members can span contexts, and thus their identifiers must be unique within the system.
2. Each collection must keep track of its constituent contexts and threads, and operations to add and delete from this list must be performed atomically.
3. Thread ranks within 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). That is, for each rope thread index t_r in the set of rope identifiers R, there exists a corresponding context thread identifier t_c in the set of context identifiers C, and a mapping function MAP that provides the translation:

$$\forall t_r \in R, \exists t_c \in C \; s.t. \; C(t_c) = MAP(R(t_r))$$

We now describe the design of such a thread collection that satisfies these requirements. Although we describe our design as an additional Chant layer, the same principles can be applied to any system of light-weight threads supporting communication between threads in separate address spaces.

3.3 Rope Servers

The requirement that each rope possess a unique identifier spanning all contexts is typically satisfied by having a centralized namer server responsible for allotting rope identifiers and for performing atomic updates to the internal data structures. Although distributed algorithms for name servers [15] and atomic operations [16] are well known, their added overhead and implementation complexity are often unwarranted in an initial design. However, a completely centralized



Rope Translation Table

Rope rank	Context Id	Thread Id
1	C1	T2
2	C2	T1
3	C2	T3
4	C3	T1

FIGURE 3 An example of a rope and its translation table.

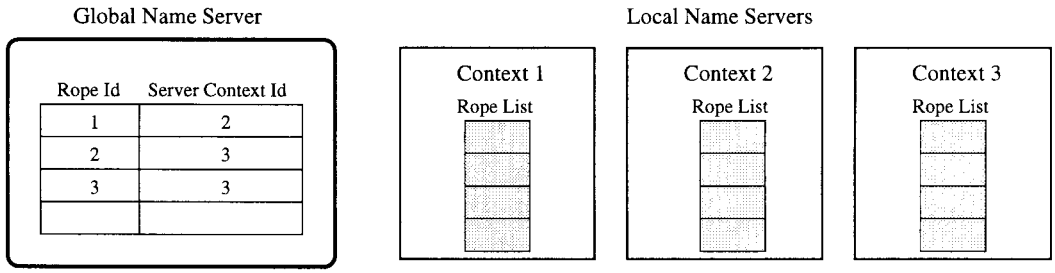


FIGURE 4 The rope name-server mechanism: rope lists are expanded in Figure 5.

solution for naming and updating ropes will certainly cause hot-spots. Therefore, our initial design for a name server utilizes a two-level approach, as depicted in Figure 4, consisting of a single global name server and a number of local name servers. This design is derived from the idea of two-level page management schemes used in distributed shared memory systems [17].

The operation of the rope name server mechanism proceeds in two levels as follows:

1. The global name server simply provides the unique identifier when a rope is created, and keeps track of which context will host the local name server responsible for the rope configuration and proper execution of rope operations. This server is needed so that any thread can find out which context is responsible for serving a particular rope. For this reason, the global server is always on a fixed, known context.
2. The local name servers are responsible for translating a relative index based on a rope identifier (<rope_id, rank>) into the global address of a thread (<context_id, thread_id>). The local name server is also responsible for processing queries regarding the state of a given rope, such as the number of total or local threads that are in the rope. All of this information is kept in a data structure as depicted in Figure 5.

3.4 Rope Creation

Because a rope is a group of threads that defines a scope for collective operations, creating a rope is tantamount to specifying this set of member 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 correspond to a task parallel model of computation, in which a single controlling task, the leader, is responsible for creating worker tasks, each with a certain amount of work to do. In other instances, it may be useful to add existing threads into an extant rope. This would correspond to a situation in which extant threads enter a data-parallel phase of the computation and then add themselves into a rope. For example, a threaded system may start with a single thread on each processor, and each of these threads may add itself into a rope representing the global set of threads. If all newly created threads were also added to this rope, then this would be the thread-level equivalent to MPI's `MPI_COMM_`. 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 in 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:

*The interface for all the rope calls, as currently implemented in Chant, is given in Figure 6.

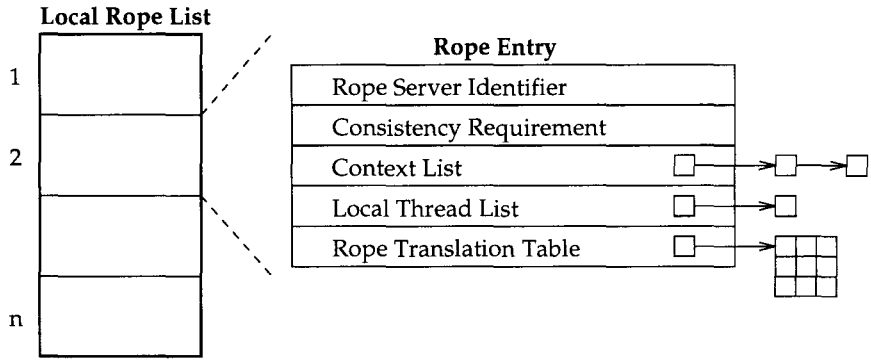


FIGURE 5 Data structure for local rope list.

1. `rope_addnew`, which creates a specified number of threads on a set of contexts and adds them to a rope.
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 thread lists with the ranks of the new threads. More details on how the addnew operation works is given in Section 3.5, after discussing how the translation tables work.

3.5 Relative Indexing

Spatial relationships play an important role in data-parallel algorithms. Thus, most communication systems, such as MPI, NX, etc., provide a linear ordering of the participating processes, which allows relative indexing of the processes independent of their actual system address. Ropes provide a similar 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, which stores the associated global identifier for each relative index within a rope (refer to Figure 3). 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 5 depicts the data structure for the local rope list, including the rope translation table.

Again, borrowing from earlier work in an area of page coherence for distributed shared memory systems, [18], 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 weak consistency model should result in better performance, because 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. When a `rope_addnew` is performed, 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, then an image of the new rope translation table is propagated to all member contexts.

To translate a relative index (<rope_id, rank>) into a Chant global thread identifier (<context_id, thread_id>), 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 global thread id of relative index within this rope be returned.
2. If the local rope tables does have an entry for the given rope identifier, 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.

3.6 Collective Operations

MPI provides the group facility for specifying which processes will participate in a collective operation, and ropes extend this idea to the thread level. To do this, each context participating in a rope must known 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 5).

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 context will maintain an accumulator thread per rope which is responsible for accumulating the number of local threads that have participated in a global operation.
2. Each thread, upon executing the barrier command, will send a message to the accumulator thread for the local 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.
3. After the local accumulator thread has collected the number of barrier messages equal to the number of local 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.
4. 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.
5. The accumulator threads then triggers the events for the local waiting threads, thus completing the barrier.

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 called 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 multithreaded 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 nonblocking barrier operation, we would certainly incorporate it into the design as mentioned.

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

4 INTERFACING WITH ROPES

In this section we address the issues of interfacing with ropes from the perspective of a data-parallel compiler, such as a compiler of high-performance Fortran (HPF) [19], targeting a multithreaded system, such as Chant. For reference, Figure 6 gives the interface for the rope calls as currently implemented in Chant.

The primary goal of a run-time-based implementation of ropes is to allow for an off-the-shelf HPF compiler to generate data-parallel code that can be executed in a multithreaded environment. For scientific computing, this allows for the same HPF program to execute in a variety of environments, including one that supports task parallelism.

`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 6 Ropes interface in Chant.

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

FIGURE 7 HPF program fragment.

To examine the issues involved, let us consider the HPF fragment shown in Figure 7 in which an array, A, is distributed by block across a set of processors. The number of processors is to be determined externally at run-time. The `forall` statement shifts the array one element to the left. The scalar variation, X, is assumed to replicate, 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 8. In this code, 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 nonlocal 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 to the replicated scalar variable X, on the other hand, turns into a broadcast from the context-owning element A(1) to all other contexts.

```
      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 8 Resulting data-parallel code from HPF fragment.

Targeting a thread-based run-time system would require that the compiler 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.

The design of ropes, as described here, provides run-time support for the translation table and the multicasting in the context of a multithreaded environment. The code produced by a compiler targeting such as interface would have two parts. First, the rope would have to be intialized 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 9.

We assume here that the P contexts participating in the execution have already been set up. 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 exist before

```
! 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 9 Sample initialization code for ropes.

continuing. The user code is executed by each thread in the rope using the subroutine *UserF* depicted in Figure 10.

In comparing the two codes (Figure 8 and Figure 10), we see that the differences are minor. This implies that a compiler has to do very little work to re-target from a process-based system to a thread-based system. The rope-based code has some extra setup code. Also, the individual thread code has to determine its rope before it starts execution and the message sent 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 run-time routines to target a ropes interface. This offers the advantage of re-targeting a data-parallel compiler for a thread-based run-time system with very little modification.

```
  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 10 Resulting data-parallel code from HPF fragment, with ropes.

5 EXPERIMENTAL RESULTS

Having described the design of a run-time-based approach for supporting data-parallel programming in a multithreaded environment, we now discuss performance figures for our ropes implementation atop Chant. We will briefly discuss the performance of the inherent design, including rope creation, rope addition, and message passing using relative indexing, as well as the performance of a data-parallel program for a Jacobi-like computation. The target machine is a 64-node Intel Paragon located at the NASA Langley Research Center.

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 be used, local data structures are initialized and the rope creation is complete. If only a single `rope_create` operation is being executed on all contexts in the system, 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 thenew 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 $2n + N + 1$ for strong consistency, where N is the total number of contexts involved in this rope. By using the exchange time shown in Figure 11, it is straightforward calcula-

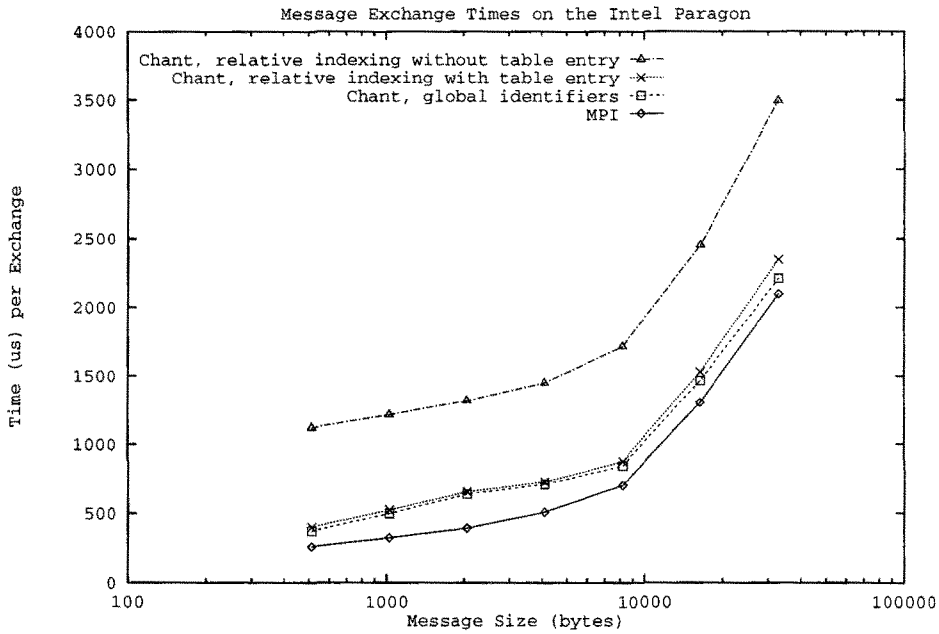


FIGURE 11 Execution times for point-to-point communication.

tion to determine the cost of creating new threads on n contexts. Table 1 shows the times (in microseconds) for creating four new threads on each of two to five contexts and adding them to an existing rope.

5.2 Relative Indexing

Figure 11 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:

1. The bottom line corresponds to MPI communication directly between processes.
2. The next line represents Chant thread-to-thread communication using global thread indentifiers.
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.

Table 1. rope_addnew Times (ms) for Weak and Strong Consistency Models

Contexts	Weak (ms)	Strong (ms)
2	4.65	5.48
3	5.72	7.15
4	6.68	8.22
5	7.90	9.64

The results from Figure 11 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 A Data-Parallel Computation

For this experiment, we implement a data-parallel algorithm for smoothing a two-dimensional array of

Table 2. Execution Times for an N × N × N Unbalanced Jacobi Running on a 2 × 2 × 2 Intel Paragon

N	Two Ropes in Sequence (ms)	Two Ropes in Parallel (ms)
2	11.5	11.1
4	48.5	43.3
8	369.3	293.6
16	3650.0	2660.0

values using a Jacobi-like computation. The computational kernel of a two-dimensional Jacobi algorithm is a five-point stencil over a two-dimensional array. Specifically, each new $A_{i,j}$ element is computed as follows:

$$A_{i,j} = \begin{cases} A'_{i,j} & \text{if } i = 1,\ i = n,\ j = 1,\ j = n \\ A'_{i,j}/2{,}0 + (A'_{i-1,j} + A'_{i+1,j} + A'_{i,j-1} + A'_{i,j+1})/8.0 & \\ \text{otherwise} & \end{cases}$$

where A represents the current iteration and A' represents the previous iteration.

Two copies of the algorithm are mapped onto two ropes over a set of 2 × 2 × 2 processors, where the solution grid size of N × N × N varies from N = 2 to N = 16. The two ropes are executed both in sequence and in parallel, and the results are depicted in Table 2 and Figure 12.

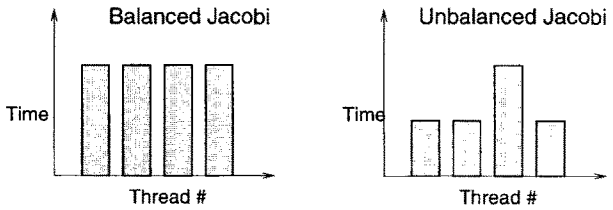


FIGURE 13 Balanced and unbalanced Jacobi.

For a balanced Jacobi iteration (c.f., Figure 13), all threads would get a similar number of array elements and the computation required to smooth each element would be the same for all elements. With this approach, each thread will take about the same amount of time, so a parallel execution of two ropes will not reveal how the threads can actually take advantage of wasted processor cycles. However, if we implement an unbalanced Jacobi computation, where the smoothing function varies for each array element, then some threads in the rope will complete their execution before other threads, resulting in wasted computation gaps. An unbalanced Jacobi corresponds to real-world situations where, for example, the iterative solution of an element within a grid varies depending on its location within the grid.

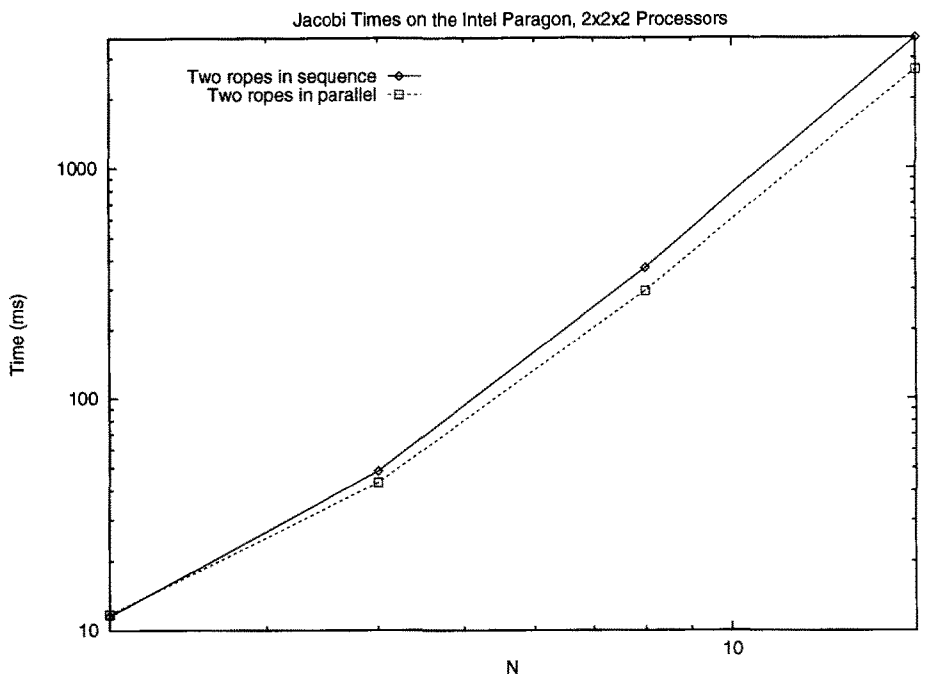


FIGURE 12 Execution times for an N × N × N unbalanced Jacobi running on a 2 × 2 × 2 Intel Paragon.

In our experiment, we implement the unbalanced Jacobi, hence there is a difference between the sequential and parallel rope executions. The difference, in fact, corresponds to the gap between the longest-running thread in the first rope and the thread in the first rope that is mapped to the same processor as the longest running thread in the second rope. This difference is illustrated in Figure 14. Were both long-running threads mapped to the same processor, the gap would be zero and thus the execution times the ropes in parallel would be the same as the execution times for the ropes in sequence.

6 RELATED RESEARCH

There are several systems which support distributed threads, such as Nexus [6], and Panda [3], although these systems do not currently support the notion of ropes.

The term *rope* was first coined in the pthreads++ system [20], 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 [10]. 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 [21], 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 [14], 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 collective operations among the threads.

The contribution of this article 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.

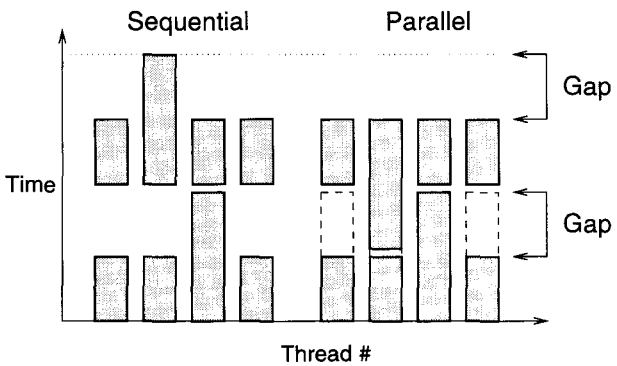


FIGURE 14 Sequential and parallel unbalanced Jacobi using two ropes.

7 CONCLUSIONS

Recently, several run-time 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 article 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 run-time system for supporting data-parallel codes in a multithreaded environment, and will report on the results of this effort in future work.

REFERENCES

[1] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Orca: A language for parallel programming of distributed systems," *IEEE Trans. Software Eng.*, vol. 18, pp. 190–205, March 1992.

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

[3] J. K. Lee and D. Gannon, "Object oriented parallel

programming experiments and results," in *Proc. Supercomputing 91*, 1991, p. 273.

[4] P. Mehrotra and M. Haines, "An overview of the Opus language and runtime system," in *Languages and compilers for Parallel Computers*. New York: Springer-Verlag Leture Notes in Computer Science, vol. 892, 1995, pp. 346–360.

[5] R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek, "Panda: A portable platform to support parallel programming languages," in *Symposium on Experiences with Distributed and Multiprocessor Systems IV*, 1993, p. 213.

[6] I. Foster, C. Kesselman, R. Olson, and S. Tuecke, "Nexus: An interoperability layer for parallel and distributed computer systems," Argonne National Labs, Argonne, IL, Tech. Rep. Version 1.3, Dec. 1993.

[7] M. Haines, D. Cronk, and P. Mehrotra, "On the design of Chant: A talking threads package," in *Proc. Supercomputing*, 1994, p. 350.

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

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

[10] Message Passing Interface Forum, *Document for a Standard Message Passing Interface*, version 1.1 ed., June 1994. http://www.mcs.anl.gov/mpi/.

[11] V. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency Practice Exp.* vol. 2, pp. 315–339, Dec. 1990.

[12] IEEE, "Threads extension for portable operating systems (Draft 7)," Feb. 1992.

[13] A. Skjellum, N. E. Doss, and K. Viswanathan, "Intercommunicator extensions to MPI in the MPIX (MPI eXtension) library," Computer Science Department and NSF Engineering Research Center, Mississippi State University, Tech. Rep., July 1994.

[14] A. Skjellum, N. E. Doss, K. Viswanathan, A. Chowdappa, and P. V. Bangalore, "Extending the message passing interface (MPI)," Computer Science Department and NSF Engineering Research Center, Mississippi State University, Tech. Rep., 1994.

[15] R. M. Needham, "Names," in *Distributed Systems*, S. Mullender, Ed. New York: ACM Press, 1989, pp. 89–101.

[16] M. Maekawa, "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," *ACM Trans. Computer Systems*, vol. 3, pp. 145–159, May 1985.

[17] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Adaptive software cache management for distributed shared memory architectures," Rice University, Tech. Rep. Rice COMP TR90-109, March 1990. (Appears in Proceedings of ISCA 17.)

[18] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed shared memory based on type-specific memory coherence," Rice University, Tech. Rep. Rice COMP TR89-98, Nov. 1989.

[19] High Performance Fortran Forum, *High Performance Fortran Language Specification*, version 1.1 ed., November 1994. http://www.erc.msstate.edu/hpff/home.html.

[20] N. Sundaresan and L. Lee, "An object-oriented thread model for parallel numerical applications," in *Proc. Second Annual Object-Oriented Numerics Conf.* 1994, p. 291.

[21] A. K. Chowdappa, A. Skjellum, and N. E. Doss, "Thread-safe message passing with P4 and MPI," Computer Science Department and NSF Engineering Research Center, Mississippi State University, Tech. Rep. TR-CS-941025, April 1994.