On the Design of Chant: A Talking Threads Package*

Matthew Haines David Cronk Piyush Mehrotra

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center, Mail Stop 132C
Hampton, VA 23681-0001
[haines,cronk,pm]@icase.edu

April 27, 1994

Abstract

Lightweight threads are becoming increasingly useful in supporting parallelism and asynchronous control structures in applications and language implementations. However, lightweight thread packages traditionally support only shared memory synchronization and communication primitives, limiting their use in distributed memory environments. We introduce the design of a runtime interface, called Chant, that supports lightweight threads with the capability of communication using both point-to-point and remote service request primitives, built from standard message passing libraries. This is accomplished by extending the POSIX pthreads interface with global thread identifiers, global thread operations, and message passing primitives. This paper introduces the Chant interface and describes the runtime issues in providing an efficient, portable implementation of such an interface. In particular, we present performance results of the initial portion of our runtime system: point-to-point message passing among threads. We examine the issue of thread scheduling in the presence of polling for messages, and measure the overhead incurred when using this interface as opposed to using the underlying communication layer directly. We show that our design can accommodate various polling methods, depending on the level of support present in the underlying thread system, and imposes little overhead in point-to-point message passing over the existing communication layer.

1 Introduction

Lightweight thread packages are seldom used in distributed memory multiprocessors due to their inability to support direct communication between individual threads in separate address spaces. We introduce the term *talking threads* to represent the notion of two threads in direct communication with each other, regardless of whether they exist in the same address space or not. In this paper, we describe the design of a runtime system for *talking threads* called *Chant*. Chant is capable of supporting both point-to-point primitives and remote service requests (e.g., remote procedure call) using standard lightweight thread and communication libraries. Standard point-to-point message passing primitives [9] are needed to support most existing message passing programs, including those generated by parallelizing compilers [15, 19, 34] and portable communication

*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.

libraries [4, 30]. Remote service request primitives are needed to support RPC communications [26, 31] client-server applications, and irregular computations.

Threads are becoming increasingly useful in supporting parallelism and asynchronous events in applications and language implementations, for both parallel and sequential machines. Threads are used in simulation systems [11, 28] (to represent asynchronous events that can be mapped onto single or multiple processors); they are used in language implementations [21, 22, 27] (to provide support for coroutines, Ada tasks, and C++ method invocations); and they are used in generic runtime systems [10, 13, 33] (to support fine-grain parallelism, multithreading, and interoperability). In light of their increasing use, the POSIX committee has adopted a standard for a lightweight threads interface [16], and many independent lightweight thread libraries have been designed and implemented for workstations and shared memory multiprocessors [1, 3, 11, 18, 23, 29].

Despite their popularity in shared memory systems, lightweight thread packages for distributed memory systems have received little attention. This is unfortunate: in a distributed memory system, lightweight threads can overlap communication with computation (latency tolerance) [8, 12]; they can emulate virtual processors [25]; and they can permit dynamic scheduling and load balancing [6]. However, there is no widely accepted implementation of a talking threads package.

Our goal is to design and implement a runtime system capable of supporting talking threads based on accepted lightweight thread and communication libraries. Our design goals center on high portability, based on existing standards for lightweight threads and communication systems, and high efficiency, based on supporting point-to-point message passing without interrupts or extra message buffer copies. This system will then be used to support our extensions to the High Performance Fortran standard [14] for task parallelism and shared data abstractions [5], as well as providing support for other languages and systems.

The remainder of the paper is organized as follows: Section 2 provides background on lightweight threads, communication primitives, and related research. Section 3 discusses our design of a new talking threads package called Chant. We have implemented the bottom layer of Chant; Section 4 reports on two experiments that validate our design decisions for this layer.

2 Background

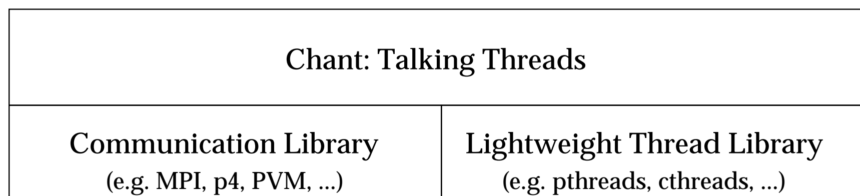


Figure 1: Chant runtime layers

Chant provides an interface for talking threads by extending the interfaces of a communication system and a lightweight thread system, as depicted in Figure 1. As far as Chant is concerned, these systems can be abstracted as two "black boxes" of systems: one box for communication packages, and one box for lightweight thread packages. Although there are distinguishing features among the systems within each box, Chant is only interested in the general characteristics of each box, as depicted in Figures 2 and 3. Rather than binding Chant to a particular lightweight thread or communication package, we allow for a design which can acommodate any system which provides this common set of capabilities.

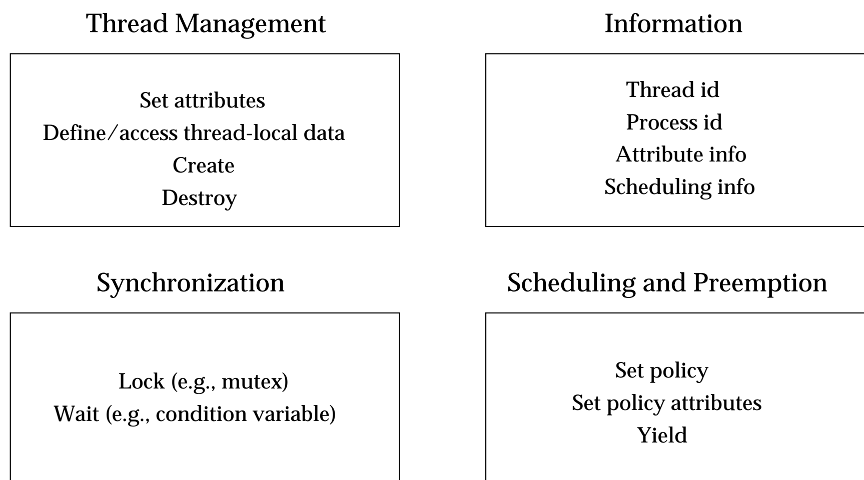


Figure 2: Desired lightweight thread package capabilities

2.1 Lightweight Thread Libraries

A thread represents an independent, sequential unit of computation that executes within the context of a kernel-supported entity, such as a Unix process. Threads are often classified by their "weight", which corresponds to the amount of context that must be saved when a thread is removed from the processor, and restored when a thread is reinstated on a processor (i.e. a *context switch*). The context of a Unix process includes the hardware register, kernel stack, user-level stack, interrupt vectors, page tables, and more [2]. The time required to switch this large context is typically on the order of thousands of microseconds, and thus a Unix processes represents a *heavyweight* thread. Contemporary operating system kernels, such as Mach, decouple the thread of control from the address space, allowing for multiple threads within a single address space and reducing the context of a thread. However, the context of a thread and all thread operations are still controlled by the kernel, which must often include more context than a particular application cares about. Context switching times for kernel-level threads are typically in the hundreds of microseconds, resulting in a medium or *middleweight* thread. By exposing all context and thread operations at the user-level, a minimal context for a particular application can be defined, and operations to manipulate threads may avoid crossing the kernel interface. As a result, user-level threads can be switched in the order of tens of microseconds, and are thus termed *lightweight*.

Most lightweight thread packages contain functionality for creating, deleting, scheduling, and synchronizing threads in a shared memory (uniprocessor or multiprocessor) environment. Other features, such as control over stacks, signal handling within threads, thread-local data, and priority scheduling are only available in certain systems. Table 1 lists several of these lightweight thread packages, including a comparison of their thread creation and context switching times for a Sun Sparcstation 10. In an attempt to provide a standard interface and set of functionality, the POSIX committee has drafted a standard threads interface [16].

None of the thread packages listed, including the proposed standard, provide support for direct communication between threads in separate address spaces. However, thread packages that do support interprocessor communication mechanisms of some form (from within the context of a lightweight thread) include:

- Nexus [10], a runtime interface designed to support interoperability among programming languages on distributed memory computer systems, providing a thread abstraction that is capable of interprocessor communication in the form of asynchronous remote procedure calls, or Active Messages [31]. However, standard send/receive primitives are not directly supported, and the overhead in providing

Thread Package	Create (μs)	Switch (μs)
cthreads [23], originally developed as the Mach user-level threads package; has been ported to many machines.	423	81
The REX lightweight process library [20], defines a minimal, non-preemptive, priority-based threads package for a number of workstations and shared memory multiprocessors.	230	60
pthreads [22], provides a library implementation of the POSIX pthreads standard interface, draft 6.	1300	29
The Sun Lightweight Process (LWP) library [29], provides a comprehensive set of thread routines supporting priorities, user-defined contexts, and stack management routines; only available under the SunOS 4x operating system.	400	25
Quickthreads [18], provides a low-level, portable set of stack primitives for writing efficient thread packages.	440	21

Table 1: Performance of several thread packages on a Sun SparcStation 10

this functionality is unknown. Also, the overhead to select, verify, and call the correct message handling routine without hardware and operating system support is expensive on most machines [26, 31]. Chant takes the opposite approach by providing a basis for efficient point-to-point communication (using well-known libraries), on top of which a remote service request mechanism is provided.

- NewThreads [8], an object-oriented runtime library that supports a non-preemptive, user-level threads class. Threads may communicate using special blocking point-to-point communication calls in which messages are sent to *ports*, and a port can be mapped into any thread on any node. A global name server is necessary to manage the unique global port identifiers. NewThreads is closest in spirit to the goals of Chant, but we extend its support in two directions:

  1. support for general, point-to-point message passing as supported by most communication library systems and the proposed message passing interface (MPI) [9] rather than blocking messages that must use a global naming server, and
  2. support for a minimal, yet powerful, lightweight thread interface that extends the POSIX standard, and can be quickly and efficiently implemented on a variety of machines using existing POSIX thread libraries or our own POSIX interface, implemented using the Quickthreads package.

- Various application-specific runtime systems which provide (either directly or indirectly) support for talking threads, including a runtime system for parallel simulations [25] and runtime systems for functional languages [7, 12]. However, these systems do not provide a general library or interface for talking threads. They are often only available under certain architectures and communication systems, and often use explicit thread management routines, encoded for a particular architecture, rather than a portable lightweight threads library.

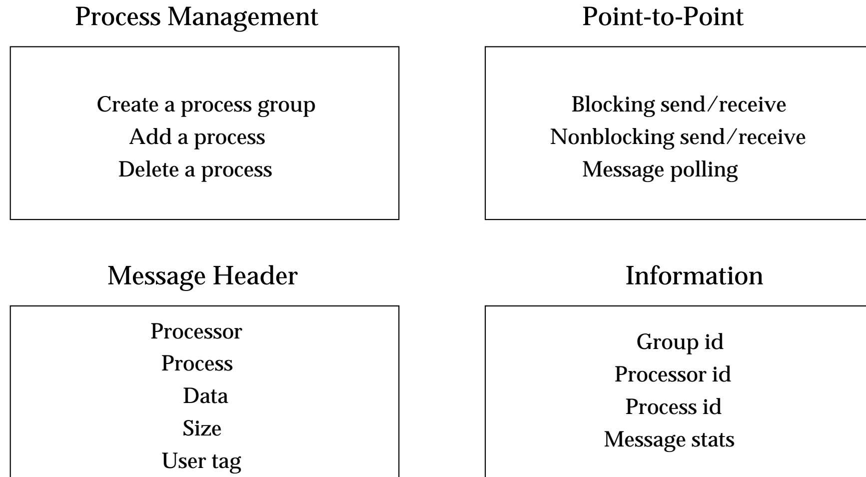


Figure 3: Desired communication package capabilities

2.2 Communication Libraries

Communication systems for distributed memory architectures have traditionally been provided by the vendors, such as the Intel NX primitives [17] and nCUBE Vertex primitives [24]. In response to the increasing demands of portability, several communication libraries have been established that provide a portable message passing interface over a wide variety of systems. Among these libraries, p4 [4] and PVM [30] have received the most attention. Then, in an effort to unify the message passing community and entice vendors to support a single message passing interface, the Message Passing Interface Forum was established to prepare a standard interface that could be supported directly by vendors (for efficiency) and would provide a portable interface for application's programmers and compilers. The result is the message passing interface standard (MPI) [9]. However, neither the MPI standard nor the other communication libraries provide direct support for message passing among threads. Although it is possible to uniquely name each thread within some of these systems, the issues of message delivery and thread scheduling as a result of message polling are not supported. As a result, there is not a general method for supporting lightweight threads from within one of these systems. Chant extends the functionality of these message passing systems to support the notion of naming, message delivery, and polling within a thread system.

3 Design

Figure 4 provides another look at Chant in the context of a layered runtime system, but this time we illustrate the intermediate layers of the Chant system itself. Chant is composed of four sub-layers: a point-to-point communication system, a remote service request mechanism, an extension of the lightweight threads interface to account for global threads, and a coherent interface based on an extension of the pthreads standard interface. Remote service requests are built upon the point-to-point layer, and thread extensions are, in turn, built upon the remote service request layer (as Figure 4 depicts). We now present a discussion of the design issues for supporting efficient point-to-point communication (Section 3.1), remote service requests (Section 3.2), and global thread operations (Section 3.3). The proposed Chant interface, based on the pthreads standard, is given in Appendix A. Each of these sections begins with an iconic representation of Figure 4, in which the highlighted layer corresponds to the given section.

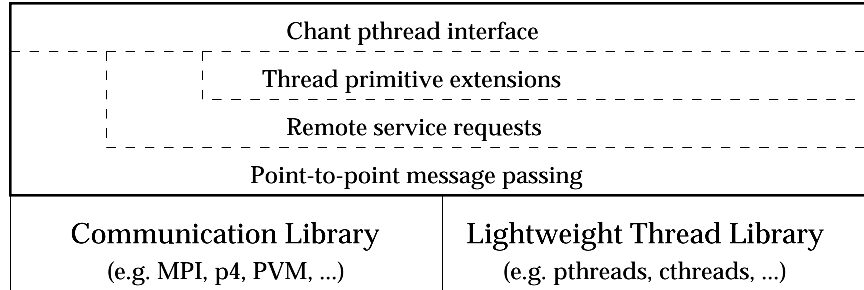
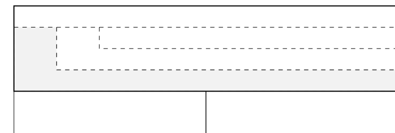


Figure 4: Chant runtime layers: exposed view



3.1 Point-to-Point Communication

We now present a discussion regarding the design issues in providing efficient message passing communication between threads using existing communication systems. Efficiency dictates that Chant cannot make intermediate copies of the messages nor allow processor interrupts that would disrupt the code and data caches, and thus our design of a point-to-point layer takes these issues into account. A measure of the overhead incurred from the point-to-point layer is presented in Section 4.1.

Point-to-point communication is defined by the fact that both the sending thread and receiving thread agree that a message is to be transferred from the sending thread to the receiving thread. Although there are various forms of send and receive primitives, the understanding on both sides that a communication is to occur is necessary. As a result of this understanding, it is possible to avoid costly interrupts and buffer copies by registering the receive with the operating system before the message actually arrives. This allows the operating system to place the incoming message in the proper memory location upon arrival, rather than making a local copy of the message in a system buffer. Chant ensures that no message copies are incurred that wouldn't otherwise be made by the underlying communication system, which is paramount to efficiency.

The basic point-to-point operations are *send* and *receive*, where a send operation creates a message and places it into the network with a given destination, and the receive operation takes a message from a specified source and remove it from the network. Both of these basic operations can be *blocking* or *nonblocking*, with different degrees of blocking (such as *locally-blocking* or *globally-blocking*). For a more thorough treatment of message passing concepts, the user is encouraged to read the MPI standard [9], or related message passing documents.

To support message passing from one thread to another, Chant must provide solutions to the problems of naming global threads within the context of an operating system entity (which we'll refer to as a process), delivering messages within a process, and polling for outstanding messages.

1. *The naming issue.* Similar to the way in which processes are named relative to a particular processing element, threads will be globally named relative to a particular process. Chant uses a 3-tuple to identify global threads, composed of a processing element identifier (pe), a process identifier, and a local thread identifier. The type of the local thread identifier is determined by the thread type of the underlying thread package and, although this will vary for different thread pacakges, allows the global threads to behave normally with respect to the underlying thread package for operations not concerned with global threads. This allows Chant to easilly inherit much of the underlying thread interface.

```
receive ( args )
{
        ireceive ( args );
        while ( probe ( args ) != true )
                yield;
        receive ( args );
}
```

Figure 5: Pseudo-code for a blocking receive operation, thread polls

2. *The delivery issue.* Most communication systems support delivery to a particular process within a specified processing element, but do not provide direct support for naming entities within a process. All message passing systems, however, support the notion of a message header, which is used by the operating system as a *signature* for delivering messages to the proper location (process). Messages also contain a body, which contains the actual contents of the message. In order to ensure proper delivery of messages to threads, and without having to make intermediate copes, the entire global thread name (pe, process, thread) must appear in the message header. Some communication systems, such as MPI, provide a mechanism by which thread names can easilly be integrated into the message header. MPI accomplishes this using the *communicator* field, which is similar to the process field in most other communication systems except that it can be used to represent multiple entities within the same process. However, most communication systems, such as p4, do not provide explicit support for the addition of a thread identifier to the message header. For these systems, we must overload one of the existing fields: typically the user-defined tag field. This approach has the disadvantage of reducing the number of tags allowed, typically to half the number of bits, where the thread id would occupy half of the tag field and the tag would occupy the other half. An alternative approach would be to place the thread id in the body of the message, leaving the existing header intact. However, this would force an intermediate thread to receive all incoming messages, decode the body, and forward the remaining message to the proper thread. In addition to being time consuming, this method would require the message body to be copied on both the sending (to insert the thread id) and receiving (to extract the thread id) sides. To maintain efficiency, message copies must be avoided, and thus placing the thread identifier in the body of the message is not an acceptable option.

3. *The polling issue.* Although Chant supports, at the user interface, both blocking and nonblocking message operations, only nonblocking communication primitives from the underlying communication system are utilized. This is to prevent a blocking call from suspending the entire process, thus preventing other ready threads from executing.

   When a non-blocking operation is performed, the communication system returns a “handle” that can be used to check the completion of the operation at a later point in time. To implement a blocking receive, the calling thread issues a corresponding nonblocking receive and waits until the operation has completed. However, rather than block the processing element, we wish to schedule other ready threads for execution and only return to the calling thread when the receive operation has been completed. This leads to the question of how to perform the required polling operations necessary to determine when the nonblocking operation has completed. A thread scheduling policy must therefore take message polling into account when scheduling ready threads for execution. There are two basic alternatives to polling for message completion: having the thread poll for itself whenever it is rescheduled for execution (refer to Figure 5), or having the scheduler issue the polling request on behalf of the thread whenever it is between scheduling operations (see Figure 6). The former method has the advantage of not having to register receive operations with the scheduler, but will cause context switching overheads in the case when a thread is re-scheduled but cannot complete the receive operation. The latter method requires all threads to register a receive with the scheduler, and then are removed from the ready queue and

```
receive ( args )
{
        ireceive ( args );
        if ( probe ( args ) != true )
                add probe request to scheduler table;
                yield;
        endif;
        // rescheduled at this point only when the
        // scheduler-activated probe succeeds.
        receive ( args );
}
```

Figure 6: Pseudo-code for a blocking receive operation, scheduler polls

placed on a blocking queue until the message arrives. This avoids the overhead of scheduling a thread that is not really ready to run, but forces the scheduler to poll for outstanding messages on each context switch.

In Section 4.2, we address this scheduling decision in detail and evaluate the various polling options and their influence on the scheduling of ready threads. Our goal is to determine which polling option yields the best schedules over a variety of simulated workloads. Our results indicate that, although having the scheduler poll for the threads results in the best performance, having the threads poll for themselves is only slightly worse in the average case scenario. This is significant because some underlying lightweight thread packages won't allow modification of the scheduler's activities, but all packages can safely implement the policy in which the thread polls for itself.

3.2 Remote Service Requests

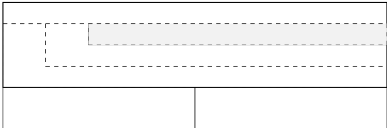
Having established a mechanism by which lightweight threads located in different addressing spaces can communicate using point-to-point mechanisms, we now address the problem of supporting remote service requests, which builds on our designs for point-to-point message passing. Remote service request messages are distinguished from point-to-point messages in that the destination thread is not expecting the message. Rather, the message details some request that the destination thread is to perform on behalf of the source thread. The nature of the request can be anything, but common examples include returning a value from a local addressing space that is wanted by a thread in a different addressing space (*remote fetch*), executing a local function (*remote procedure call*), and processing system requests necessary to keep global state up-to-date (*coherence management*).

Most remote service requests require some acknowledgment to be sent back to the requesting thread, such as value of a remote fetch or the return value from a remote procedure call. To minimize the amount of time the source thread remains blocked, we wish to process the remote service request as soon as possible on the destination processor, but without having to interrupt a computation thread prematurely. Interruptions are costly to execute and can disrupt the data and code caches which, as processor states increase, will continue to have a detrimental effect on the efficiency of a program [26]. Also, the MPI standard [9] does not support interrupt-driven message passing, thus utilizing interrupts in a design would preclude the use of the MPI communications layer. Therefore, we need a polling mechanism by which remote service requests can be checked without having to prematurely interrupt a computation thread when such a request arrives.

```
repeat forever
{
        ireceive ( remote-service-request-message-type );
        if ( probe ( args ) != true )
                add probe request to scheduler table;
                yield;
        endif;
        message = receive ( args );
        handler = unpack ( message );
        *handler ( message );
}
```

Figure 7: Pseudo-code for the server thread, scheduler polls.

Since the main problem with remote service requests is that they arrive at a processor "unannounced", we simply introduce a new thread, called the *server thread*, which is responsible for receiving all remote service requests. Using one of the polling techniques outlined in Section 3.1, the server thread repeatedly issues nonblocking receive requests for any remote service request message, which can be distinguished from point-to-point messages by virtue of being sent to the server thread rather than a computation thread. When a remote service request is received, the server thread assumes a higher scheduling priority than the computation threads, ensuring that it is scheduled at the next context switch point. Pseudo-code for the server thread is given in Figure 7, showing how remote service requests are built upon point-to-point messages.



3.3 Supporting Global Thread Operations

As well as adding communication primitives to a lightweight thread interface, Chant must support the existing lightweight thread primitives that are inherited from the underlying thread package. These primitives provide functionality for thread management, thread synchronization, thread scheduling, thread-local data, and thread signal handling. We divide these primitives into two groups: those affected by the addition of global thread identifiers in the system, and those not affected. For example, the thread creation primitive must be capable of creating remote threads, but the thread-local data primitives are only concerned with a particular local thread. Our goal in designing Chant is to provide an integrated and seamless solution for both groups of primitives. This is accomplished in two ways.

1. Global thread identifiers are 3-tuples consisting of a processing element id, a process id, and a local thread id whose type matches the thread type inherited from the underlying system. This makes it possible to extract the local thread specification for the primitives that are not concerned with global threads, such as the thread-local data operations. Chant provides a primitive (`pthread_chanter_pthread`) that returns the local thread portion of a global thread identifier for this purpose.

2. Thread primitives that are affected by the global identifiers either take a thread identifier as an argument (such as join) or return a thread identifier (such as create). In either case, the primitive must handle the situation of a thread identifier that refers to a remote thread. For example, consider the thread creation operation, which creates a local thread within the specified processing element and process. Since thread resources (such as a stack) must be allocated by the processing element on which

the thread is to be executed, creating a remote thread may require the help of another processing element.

Having described the details of how Chant supports remote service requests (in Section 3.2), we can now utilize this functionality in the form of a remote procedure call. Similar to how Unix creates a process on a remote machine [32], Chant utilizes the server thread and the remote service request mechanism to implement primitives which may require the cooperation of a remote processing element. Returning to our example of a thread creation operation, if it is determined that the new thread is to be executed on a remote processing element, a remote service request is sent to the specified processing element, informing it to create the desired thread (allocate resources) and insert it into the local thread queue.

4 Experimental Results

It is not our goal to argue that threads themselves are useful for programming distributed memory multiprocessors (the argument can certainly be made: for example, consider latency tolerance and dynamic load balancing capabilities, which are natural extensions of a thread-based implementation). Instead, our goal is to demonstrate that our design decisions were effective in implementing a talking threads interface, namely Chant. To prove the effectiveness of these decisions, we perform two different experiments on the point-to-point layer of our system (refer to Figure 4), which is the layer we have currently implemented. The other two layers, remote service requests and thread primitive extensions, have been designed and are now being implemented atop our point-to-point layer, and we hope to report on them soon. The first experiment is designed to measure the overhead of thread-based point-to-point communication as opposed to point-to-point communication as supported by the underlying communication system. The second experiment is designed to test the various scheduling techniques that are available when polling for outstanding messages in point-to-point communication. All of the experiments are carried out on an Intel Paragon machine using the NX message passing library and a small lightweight thread library as the underlying components.

4.1 Thread-Based Point-to-Point Communication Overheads

If we assume that threads can simplify programming for such optimizations as latency tolerance and dynamic load balancing, then we wish to know the cost at which this simplification comes. That is, what is the tradeoff, in terms of execution time overhead, for using thread-based point-to-point communication over using the point-to-point communication mechanism directly provided by the underlying system. To answer this question, we measure the cost of sending and receiving messages on the Paragon using two processes and the NX primitives, and compared this to the cost of sending and receiving messages between two threads on two Paragon processors (one per processor) using Chant. In a sense, this is a worst-case scenario for a threads system, since we are using the exact same calls as the process version, but having to add some amount of overhead per message to handle the thread naming and delivery issue, as well as a possible context switch. In the general case, we expect that there would be multiple threads per processor, and a context switch would result in overlapping communication with useful computation.

Table 2 (also depicted in Figure 8) gives the results of this experiment in terms of average time per message (μs) and overhead relative to the process-based method. To get accurate results, each test consisted of 100,000 message exchanges, and each test was repeated four times. The numbers given represent an average of these runs. We actually implemented two different thread-based approaches to demonstrate how the overhead is affected by message polling. In first method, *Thread (TP)*, threads poll for their own outstanding message (as depicted in Figure 5). Since there is only one thread per processor, the scheduler simply returns without having to perform a context switch each time. In the second method, *Thread (SP)*, the scheduler polls for outstanding requests on behalf of the thread (as depictd in Figure 6), forcing a

Message size	*Process* Time (μs)	*Thread (TP)* Time (μs)	*Thread (TP)* Overhead (%)	*Thread (SP)* Time (μs)	*Thread (SP)* Overhead (%)
1024	667.1	710.8	6.4	773.7	15.9
2048	917.0	973.2	6.1	1126.5	22.8
4096	1639.3	1701.2	3.8	1828.8	11.5
8192	2873.5	2998.8	4.3	3130.8	8.9
16384	5531.8	5624.8	1.7	5689.0	2.9

Table 2: Average time per message (μs) and overhead for thread-based point-to-point communication, based on process-based communication times for various-length messages (bytes)

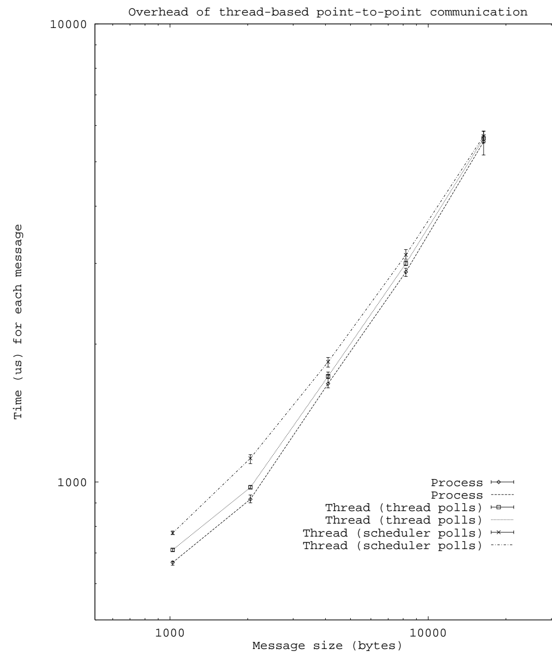


Figure 8: Execution times for native and thread-based communication

```
loop
{
        compute (alpha);
        send ();
        compute (beta);
        recv ();
}
```

Figure 9: Pseudo-code for threads, polling exercise.

context switch for each message received. We study polling methods in more detail in Section 4.2. Our results indicate that the overhead of the thread-based point-to-point layer is low, adding only 15% overhead to the base message passing layer in the worst case.

4.2 Thread Scheduling for Message Polling

In Section 3.1 we introduced the problem of polling for outstanding messages in point-to-point communication. We now take a closer look at that problem, measuring three scheduling algorithms, and determine their effect on the performance of the system.

Recall that there are essentially two methods of polling for an outstanding message: thread polls (see Figure 5), which we will refer to as the *Thread polls* algorithm, and scheduler polls (see Figure 6). The scheduler polls method is based on a list of polling requests that are examined at each scheduling point to see if any outstanding messages have arrived. Ideally, this would be implemented as a single call to the communication system, inquiring whether *any* of the outstanding receive requests have been satisfied. If so, the value returned from the check would designate a waiting thread, which could then be enabled for execution. On some communication systems this functionality is provided. For example, MPI provides the `MPI_TEST_ANY` primitive, which will test for the completion of any outstanding requests from a given process. However, on other systems, such as the Intel NX system Chant is currently using, this functionality is not supported. Therefore, the algorithm needs to be modified so that each outstanding request will be tested in turn. This implies that all outstanding messages are checked at *each* context switch. When a polling request is finally satisfied, the corresponding thread is then scheduled for execution. We refer to this version of the scheduler polling algorithm as *Scheduler polls (WQ)*, where WQ stands for "waiting queue".

Another variation on having the scheduler poll for outstanding requests on behalf of threads is to eliminate the list of polling requests altogether. Each thread stores its polling request in its thread control block (TCB), which is a data structure that defines a thread, similar to how a process control block (PCB) defines a process [2]. When the scheduler is invoked to perform a context switch, it selects the next available TCB from the thread queue and determines if a request is pending. If not, the thread's context is restored. Otherwise, the pending message is polled for by the scheduler. If the message has arrived, the thread is restored, otherwise the TCB is placed back on the thread queue and the next TCB is retrieved. This method eliminates polling for all outstanding requests at each scheduling point, and does not fully restore a thread's context until its request has been satisfied. The disadvantage of this approach is that some thread packages may not allow modification of the scheduler activities necessary to poll for a thread before completing the thread switch. We refer to this version of the scheduler polling algorithm as *Scheduler polls (PS)*, where PS stands for "partial switch".

Using the point-to-point communication layer of Chant, we encoded all three polling algorithms (thread polls, scheduler polls using a list of requests, scheduler polls using partial context switch) and tested the system to measure their relative performance using the thread code depicted in Figure 9, where the parameters `alpha` and `beta` represent the number of iterations for a generic computation, and were modified to affect

alpha	*Thread polls* Time	CtxSw	msgtest	*Scheduler polls (PS)* Time	CtxSw	msgtest	*Scheduler polls (WQ)* Time	CtxSw	msgtest
100	2730	6655	2662	**2413**	5580	2011	5950	5488	11817
1000	2860	6655	2693	**2515**	5630	2010	6090	5489	11942
10000	4000	7029	3057	**3660**	5579	2535	6123	5509	11875
100000	7260	7977	3975	**6815**	5649	3723	9990	5534	13238

Table 3: Execution times (ms), average number of threads waiting on outstanding receive requests, and total number of `msgtest` calls attempted for all three polling algorithms, `beta` = 100

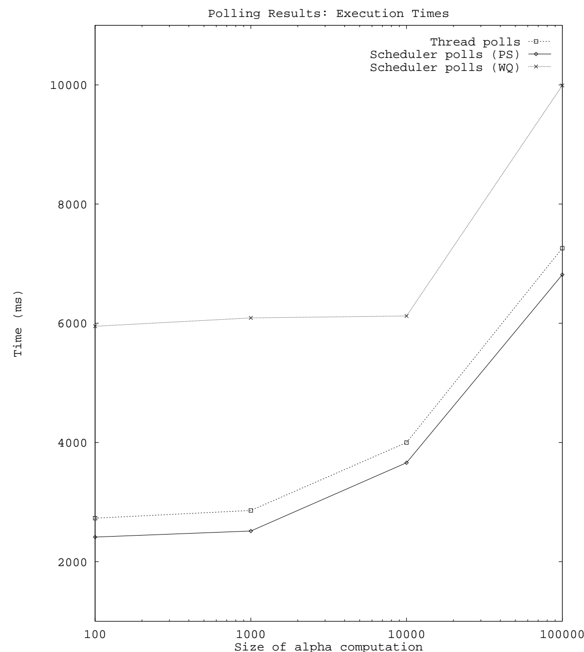


Figure 10: Execution times for polling experiments, `beta` = 100

the average number of outstanding receive requests (or waiting threads).

First, we fix `beta` at 100 and vary `alpha` from 100 to 100000, then run the experiment with two processors and 12 threads per processor, with each thread performing 100 iterations of the outer send/receive loop. Table 3 presents these results, where *Time* represent the total running time (ms) of the test, *CtxSw* represents the total number of complete context switches performed, and *msgtest* represents the total number of `msgtest` calls attempted. The data indicates that having the *Scheduler poll (PS)* algorithm yields the lowest running times for the three approaches (depicted in Figure 10). This is because the *Thread polls* algorithm must complete full context switches at each scheduling opportunity to check for the completion of a message, while the *Scheduler polls (PS)* algorithm need only perform a partial switch to check for outstanding messages. This is seen in comparing the total number of context switches for the two methods (depicted in Figure 11). However, the *Thread polls* algorithm performs, on average, only 10% worse than the *Scheduler polls (PS)* algorithm. Thus, for a thread package that does not support the ability to modify the scheduler's behavior as required for the *Scheduler polls (PS)* algorithm, we have found that only a 10% degradation of performance will result from using the *Thread polls* algorithm, which can be applied to any lightweight

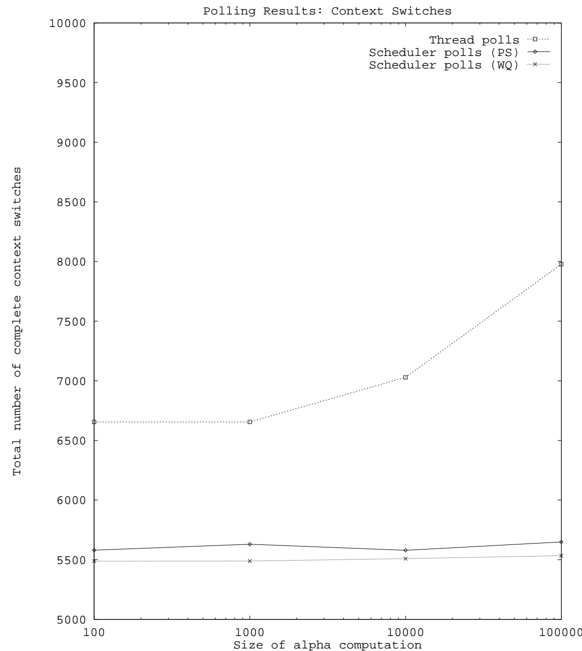


Figure 11: Total context switches for polling experiments, `beta` = 100

thread package. The data also shows that the `Scheduler polls (WQ)` algorithm performs much worse than the other two, as a result of having to check all outstanding requests at each scheduling opportunity., As we can see from the number of `msgtest` calls performed by the three algorithms (depicted in Figure 12), the *Scheduler polls (WQ)* algorithm performs far more `msgtest` calls than the other two algorithms, accounting for its degraded performance. However, the *Scheduler polls (WQ)* algorithm does achieve the lowest number of context switches of the three methods (see Figure 11), since threads are only switched when they are ready to run. For systems that could implement this algorithm as origionally intended, with a single `msgtestany` call rather than a test for each individual message, we expect the relative performance of this algorithm to change. We hope to test this hypothesis on a future version of Chant using the MPI communication system, which supports the `msgtestany` functionality.

Figure 13, which plots the average waiting time versus `alpha`, confirms that by increasing `alpha`, we successfully increases the number of threads waiting for outstanding requests. Intuitively, this means that increasing the time between when a receive is posted and the corresponding send is sent will increase the number of threads waiting for messages.

Finally, to support our conclusions, we repeated the experiments for `beta` values of 1000 and 0, presented in Tables 4 and 5, respectively. These additional results confirm our earlier analysis regarding the relative performance of the three polling algorithms.

5 Conclusions

Threads are an emerging model for supporting parallelism and asynchronous events in applications and language implementations, for both parallel and sequential machines. Despite their popularity and utility, lightweight thread packages for distributed memory systems have received little attention.

In this paper, we introduce the notion of *talking threads* as a set of lightweight threads capable of com-

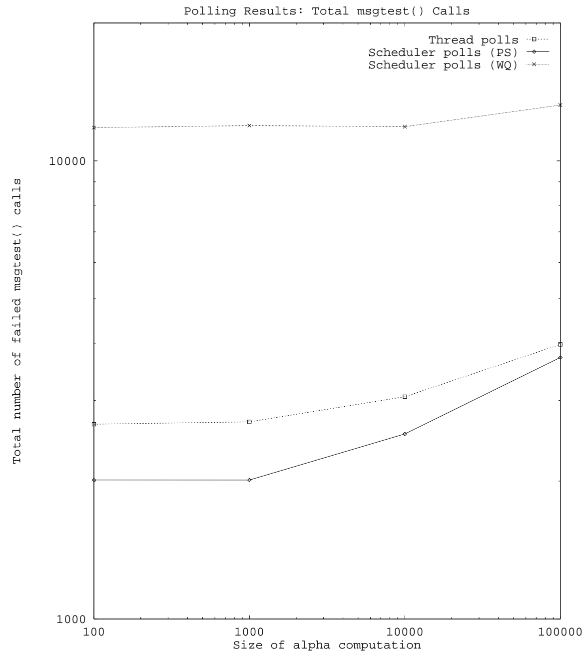


Figure 12: Total `msgtest` calls attempted for polling experiments, `beta` = 100

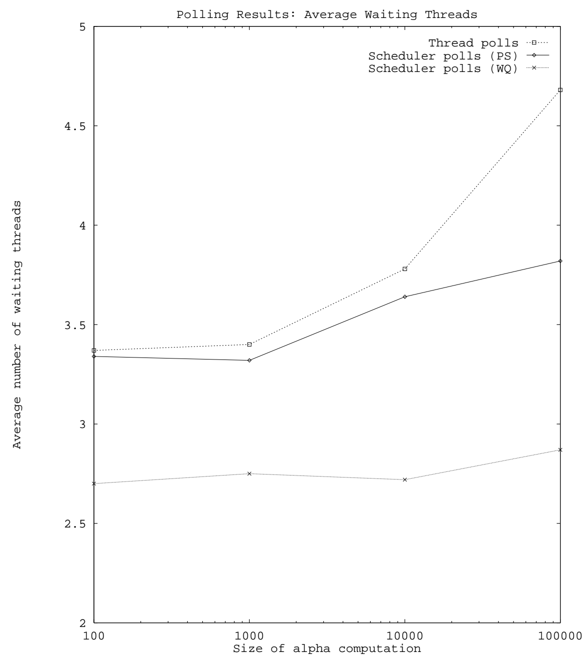


Figure 13: Average number of waiting threads for polling experiments, `beta` = 100

alpha	*Thread polls*			*Scheduler polls (PS)*			*Scheduler polls (WQ)*		
	Time	CtxSw	msgtest	Time	CtxSw	msgtest	Time	CtxSw	msgtest
100	6765	6945	2909	**6480**	5514	2415	10065	5485	12323
1000	6960	6888	2837	**6660**	5523	2564	10262	5508	13496
10000	8000	6950	2887	**7670**	5530	2311	11350	5512	12676
100000	10980	7246	3239	**10560**	5537	2532	14100	5532	12405

Table 4: Execution times (ms), average number of threads waiting on outstanding receive requests, and total number of `msgtest` calls attempted for all three polling algorithms, `beta` = 1000

alpha	*Thread polls*			*Scheduler polls (PS)*			*Scheduler polls (WQ)*		
	Time	CtxSw	msgtest	Time	CtxSw	msgtest	Time	CtxSw	msgtest
100	3290	5792	3578	**2715**	3628	3514	4940	3130	9845
1000	3460	5864	4646	**2725**	3622	3550	5120	3174	10000
10000	4570	6100	4887	**3980**	3608	4335	6080	3110	10310
100000	7805	7206	5977	**7343**	3630	6631	9263	3144	13024

Table 5: Execution times (ms), average number of threads waiting on outstanding receive requests, and total number of `msgtest` calls attempted for all three polling algorithms, `beta` = 0

munication in a distributed memory environment, and describe the design of a talking threads system called *Chant*. Chant is capable of supporting both point-to-point communication (e.g., send/receive) and remote service request communication (e.g., remote procedure call). Portability and efficiency are achieved by providing a minimal interface over existing thread and communication libraries, such as pthreads, Quickthreads, NX, and MPI.

We have designed Chant in three layers: (1) point-to-point communication among threads, (2) remote service requests using point-to-point communication and special server threads, and (3) global thread operations using remote procedure calls. We have also designed an interface to all of these layers which is based on an extension of the POSIX pthreads standard.

We have implemented the point-to-point communication layer of Chant, and measured its overhead relative the the underlying communication system. We found that in a worst-case scenario, the overhead caused by Chant's point-to-point layer is low (about 15%), but that this can be halved by avoiding a context switch when only a single thread exists on a processing element.

We have also implemented and measured three scheduling policies that poll for outstanding receive operations. We found that the *Scheduler polls (PS)* policy, in which the scheduler polls for threads after performing a partial context switch, is superior to the other two methds: having the threads poll for themselves (*Thread polls*) and having the scheduler poll for the threads using a waiting queue (*Scheduler polls (WQ)*. However, since some thread packages do not allow the scheduler to be manipulated as required to implement the *Scheduler polls (PS)* algorithm, we found that the *Thread polls* algorithm performs only slightly worse, yet can be implemtned using any lightweight thread pacakge.

We are continuing to develop Chant and its interface, and plan to use this runtime system to support various parallel languages and programming systems. We plan to report on the status of Chant and these support efforts in the near future.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Symposium on Operating Systems Principles*, pages 95–109, 1991.

[2] Maurice J. Bach. *The Design of the UNIX Operating System*. Software Series. Prentice-Hall, 1986.

[3] Brian N. Bershad, Edward D. Lazowska, Henry M. Levy, and David B. Wagner. An open environment for building parallel programming systems. Technical Report 88-01-03, Department of Computer Science, University of Washington, January 1988.

[4] Ralph Butler and Ewing Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

[5] Barbara M. Chapman, Piyush Mehrotra, John Van Rosendale, and Hans P. Zima. A software architecture of multidisciplinary applications: Integrating task and data parallelism. ICASE Report 94-18, Institute for Computer Applications in Science and Engineering, Hampton, VA, March 1994.

[6] T. C. K. Chou and J. A. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, SE-8(4), July 1982.

[7] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In 4th *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[8] Edward W. Felton and Dylan McNamee. Improving the performance of message-passing applications by multithreading. In *Scalable High Performance Computing Conference*, pages 84–89, April 1992.

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

[10] 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.

[11] Dirk Grunwald. A users guide to AWESIME: An object oriented parallel programming and simulation system. Technical Report CU-CS-552-91, Department of Computer Science, University of Colorado at Boulder, November 1991.

[12] Matthew Haines and Wim Böhm. An evaluation of software multithreading in a conventional distributed memory multiprocessor. In *IEEE Symposium on Parallel and Distributed Processing*, pages 106–113, December 1993.

[13] Matthew Haines and Wim Böhm. On the design of distributed memory Sisal. *Journal of Programming Languages*, 1:209–240, 1993.

[14] High Performance Fortran Forum. *High Performance Fortran Language Specification*, version 1.0 edition, May 1993.

[15] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

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

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

[18] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[19] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[20] Jeff Kramer, Jeff Magee, Morris Sloman, Naranker Dulay, S.C. Cheung, Stephen Crane, and Kevin Twindle. An introduction to distributed programming in REX. In *Proceedings of ESPRIT-91*, pages 207–222, Brussels, November 1991.

[21] 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.

[22] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX*, pages 29–41, San Diego, CA, January 1993.

[23] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A machine independent interface for lightweight threads. Technical Report CIT-CC-93/53, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1993.

[24] nCUBE, Beaverton, OR. *nCUBE/2 Technical Overview, PROGRAMMING*, 1990.

[25] David M. Nicol and Philip Heidelberger. Optimistic parallel simulation of continuous time markov chains using uniformization. *Journal of Parallel and Distributed Computing*, 18(4):395–410, August 1993.

[26] Matthew Rosing and Joel Saltz. Low latency messages on distributed memory multiprocessors. Technical Report ICASE Report No. 93-30, Institute for Computer Applications in Science and Engineering, NASA LaRC, Hampton, Virginia, June 1993.

[27] Carl Schmidtmann, Michael Tao, and Steven Watt. Design and implmentation of a multithreaded Xlib. In *Winter USENIX*, pages 193–203, San Diego, CA, January 1993.

[28] H. Schwetman. *CSIM Reference Manual (Revision 9)*. Microelectronics and Computer Technology Corperation, 9430 Research Blvd, Austin, TX, 1986.

[29] Sun Microsystems, Inc. *Lightweight Process Library*, sun release 4.1 edition, January 1990.

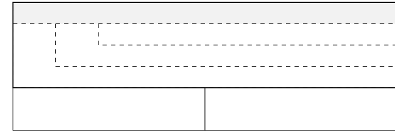
[30] Vaidy Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[31] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communications and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[32] W. E. Weihl. Remote procedure call. In Sape Mullender, editor, *Distributed systems*, chapter 4, pages 65–86. ACM Press, 1989.

[33] Mark Weiser, Alan Demers, and Carl Hauser. The portable common runtime approach to interoperability. *ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989.

[34] Hans P. Zima and Barbara M. Chapman. Compiling for distributed memory systems. *Proceedings of the IEEE, Special Section on Languages and Compilers for Parallel Machines (To appear 1993)*, 1993. Also: Technical Report ACPC/TR 92-16, Austrian Center for Parallel Computation.



Appendix A: The Chant Interface

The Chant interface can be viewed as an extension to the POSIX pthread interface, where we have created a new thread object called a *chanter*, representing a global thread capable of communication and synchronization with other global threads in the system. In addition to the pthreads routines that deal with attributes, user-local data, mutex variables, condition variables, and scheduling (which can all be applied to the pthread base of a global thread), the chant interface consists of the following routines (also depicted with ANSI prototypes in Figure 14):

- `pthread_chanter_t` is a new datatype that defines a global thread within the system, composed of a processing element identifier, a process identifier, and a local thread identifier, which is the base class of the thread type from the underlying thread package (in this case, a `pthread_t`).
- `pthread_chanter_create` creates a global thread within a specified processing element (pe) and process, which may be `LOCAL`.
- `pthread_chanter_join` blocks the calling thread until the specified global thread exits.
- `pthread_chanter_detach` informs the system that the storage for the specified global thread is to be reclaimed when the thread exits.
- `pthread_chanter_exit` terminates the calling thread, making the specified value available to any threads joining with the calling thread.
- `pthread_chanter_yield` gives up the processing element to the next ready thread, as determined by the (possibly global) scheduler.
- `pthread_chanter_self` returns the `pthread_chanter_t` structure for the calling thread.
- `pthread_chanter_pthread` returns the local thread identifier of the speficied global thread, which can then be used for any of the local thread operations provided by the underlying thread package. This allows any global thread to behave as a local thread with respect to the underlying thread package, thus avoiding the need to provide full lightweight thread capabilities at the Chant interface.
- `pthread_chanter_pe` returns the processing element identifier of the specified thread, which can be used to test if two threads occupy the same processing element, perhaps having access to common shared memory.
- `pthread_chanter_process` returns the process identifier of the specified thread, which can be used to test if two threads occupy the same process, and hence exist in the same address space.
- `pthread_chanter_equal` compares two global thread identifiers to see if they refer to the same thread. This functionality allows the global thread representation to be hidden from the user interface.
- `pthread_chanter_cancel` causes the specified global thread to exit as if it had called the `pthread_chanter_exit` routine.
- `pthread_chanter_send` sends the data pointed to by `buf` to the specified global thread. This is a locally-blocking routine, and returns when the data being sent (`buf`) can be modified.
- `pthread_chanter_recv` posts a receive for a message from the specified global thread, informing the system where the message is to be placed. This is a blocking routine, which returns only when the data is located in the specified buffer location.
- `pthread_chanter_irecv` posts a receive for a message from the specified global thread, informing the system where the message is to be placed. This is a non-blocking routine, which returns immediately, and returning a handle by which the message can be later checked for completion using the `pthread_chanter_msgtest` or `pthread_chanter_msgwait` routines. Although neither of the receive routines actually blocks the processing element, this routine will return immediate control to the calling thread rather than some other ready thread.
- `pthread_chanter_msgtest` checks for the completion of an immediate receive operation using the handle returned by the `pthread_chanter_irecv` routine, and returns a true or false value.
- `pthread_chanter_msgwait` waits for the completion of an immediate receive operation using the handle returned by the `pthread_chanter_irecv` routine.

```
typedef struct pthread_chanter {
        int             pe;             // processing element id
        int             process;        // kernel entity (process) id
        pthread_t       thread;         // thread id
} pthread_chanter_t;

int pthread_chanter_create (pthread_chanter_t *thread,
        const pthread_attr_t *attr, void * (*start_routine) (void*),
        void *arg, int pe, int process);

int pthread_chanter_join (const pthread_chanter_t *thread, void **status);

int pthread_chanter_detach (const pthread_chanter_t *thread);

void pthread_chanter_exit (void *value_ptr);

void pthread_chanter_yield (void);

pthread_chanter_t *pthread_chanter_self (void);

pthread_t pthread_chanter_pthread (const pthread_chanter_t *thread);

int pthread_chanter_pe (const pthread_chanter_t *thread);

int pthread_chanter_process (const pthread_chanter_t *thread);

int pthread_chanter_equal (const pthread_chanter_t *t1,
        const pthread_chanter_t *t2);

int pthread_chanter_cancel (const pthread_chanter_t *thread);

int pthread_chanter_send (int type, char *buf, int count,
        const pthread_chanter_t *thread);

int pthread_chanter_recv (int type, char *buf, int count,
        pthread_chanter_t *thread);

int pthread_chanter_irecv (int *handle, int type, char *buf, int count,
        pthread_chanter_t *thread);

int pthread_chanter_msgtest (int handle);

int pthread_chanter_msgwait (int handle);
```

Figure 14: Chant interface based on an extension of pthreads