1

POINT-TO-POINT COMMUNICATION USING MIGRATING PORTS

Ian T. Foster, David R. Kohr, Jr., Robert Olson,
Steven Tuecke, and Ming Q. Xu

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{foster,kohr,olson,tuecke,mqxu}@mcs.anl.gov

ABSTRACT

We describe and evaluate an implementation of a port-based communication model for task-parallel programs. This model permits tasks to communicate without explicit knowledge of the location or identity of their communication partners, which facilitates modular programming and the development of programs that consist of multiple communicating tasks per processing node. We present the protocols used for point-to-point communication and port migration, expressed in terms of portable abstractions for naming, threading, and asynchronous communication provided by the Nexus runtime system. Experimental results on an IBM SP allow us to quantify the performance costs of the functionality associated with port-based communication. We conclude that task-to-task communication via ports can be achieved efficiently even using standard system software, but that improvements in the integration of threading and communication mechanisms would improve performance significantly.

1 INTRODUCTION

Compilers for data-parallel languages such as Fortran D [9] and Vienna Fortran [3] have proven capable of synthesizing efficient single program multiple data (SPMD) code for a variety of regularly structured problems. However, there exist many problems whose solution on a multicomputer may be more efficient if one can employ *task parallelism*, that is, simultaneous execution of logically distinct tasks on the same or different processing nodes. Classes of problems for which task parallelism is particularly appropriate include pipelines for signal and image processing, task farms, and multidisciplinary simulations. For such problems, it is often desirable for the programmer to retain control

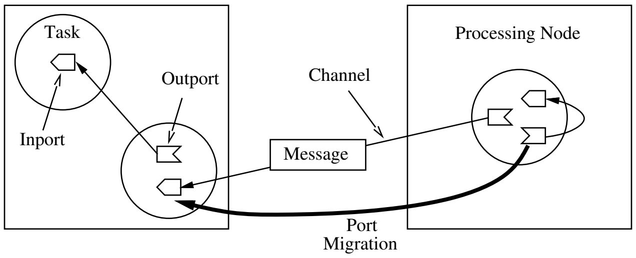


Figure 1 Collection of tasks communicating via ports.

over the definition and creation of tasks, the placement of tasks on nodes, and communication between tasks.

Communication in a task-parallel computation can be specified in a variety of ways. In the *message passing* approach, programmers use send and receive operations to transfer data from a sending task to a designated recipient task. This approach is supported by libraries such as MPI [7] and PVM [12]. It is relatively simple to implement and can be mapped efficiently to a wide range of multicomputer architectures. However, it is not in general a satisfactory solution for a high-level task-parallel programming language. Because message passing does not hide details about the physical location of recipients, development of modular programs becomes difficult. Also message passing does not provide direct support for point-to-point communications between specific tasks when several such tasks are mapped to a single node. Finally, the user is burdened with the management of message "type tags", which organize messages by logical function.

An alternative approach is for tasks to communicate by operating on abstract data types called *ports*. A sending task writes messages to an *outport*, while a receiving task reads messages from an *inport* (see Figure 1). A connection over which messages may pass from an outport to an inport is called a *channel*. This scheme provides direct support for task-to-task communication: we can communicate with a specific task by sending or receiving on a port associated with that task. Another advantage is that a task can send or receive on a port without needing to know the location or identity of the corresponding receiver or sender task. Finally, multiple channels can be created between tasks to group messages by logical function.

Because ports decouple the identities of the sender and receiver of a communication, it is possible to transfer the ability to communicate through a port from one task to another. In other words, we can *migrate* a port from one task to another task—possibly located on a different node—with complete transparency to the task using the channel's other port. Several situations motivate this port migration mechanism; some are illustrated in Figure 2. In (a), an entire task, including its ports, migrates to another node to balance the computational load. In (b), tasks from a user program call tasks from a parallel library to operate on a distributed array; after transposing the array to perform the operation, the library tasks use migrated ports to return the array to the user tasks and restore the original data distribution. In (c), the set of communicating tasks shrinks and grows over time. Some task `A` may not know the identity or location of another task `B`, but `A` may still communicate with `B` if an appropriate port migrates to `A`.

In this paper, we describe techniques that we have developed to realize port-based communication and port migration for task parallel programs. These techniques address such crucial issues as the following: message delivery and buffering, the relationship between communication and task scheduling, and a correct and efficient protocol for port migration. We have implemented these mechanisms in a runtime system for the task-parallel language Fortran M (FM), which provides high-level constructs for port-based communication [1]. We present performance results that enable us to quantify the effectiveness of our techniques.

The rest of this paper is organized as follows. In Section 2, we elaborate on the semantics of port-based communication. Section 3 points out the difficulties of implementing these semantics in a non-shared-memory environment with multitasked nodes. Section 4 sketches our design for point-to-point communications and discusses the performance of the FM implementation. In Section 5, we describe our protocol for performing port migration in the most general instances, and discuss its performance. Section 6 presents techniques for reducing the overhead of migration and shows when they are applicable. Finally,

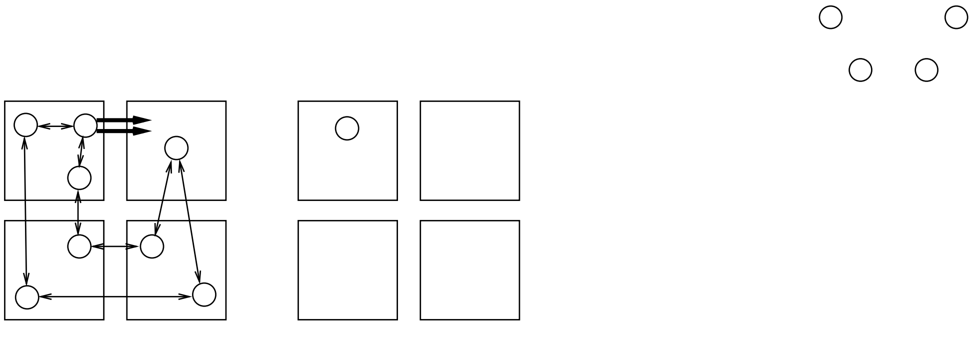


Figure 2 Situations that motivate a mechanism for port migration.

we compare our work with related projects, evaluate the effectiveness of our techniques, and map out directions for future work.

2 SEMANTICS OF PORTS

We provide a brief, somewhat simplified review of the semantics of the port operations considered in this paper, focusing on those properties with significant implications for an implementation.

- A computation comprises a set of tasks, each with a private address space and the ability to communicate with other tasks only via shared channels.
- A message source (outport) can be connected to a sink (inport) to form a channel. Channels can be created and destroyed dynamically, and a port may be part of different channels at different times. Ports may be part of at most one channel at a time, however.
- The send operation is nonblocking: a sending task may continue execution as soon as it is safe to modify the variables whose values were being transmitted, whether or not the message has been received yet. In contrast, a receiver always blocks until a message has arrived.
- Channels guarantee message ordering: messages will always be read from the inport in the order in which they were written to the outport, even if one or both of the ports migrates.
- A task may transmit a port value in a message, without affecting any channel to which the port is connected. The sending task yields to the recipient ownership of the capability for transmitting messages over the channel—that is, the port migrates from the sending task to the recipient.

An important consequence of the single-writer, single-reader property of channels is that FM can guarantee deterministic execution [1]. The same property can often be exploited to optimize certain port operations (see Section 6).

3 IMPLEMENTATION ISSUES

The port-based communication semantics described in the preceding section imposes several major requirements upon an implementation:

- The asynchronous nature of the send operation implies a need for buffering.
- Because multiple inports may be located on a single node, it must be possible to select an incoming message based on its destination port.

- Because multiple tasks may execute on a single node, it is desirable that a task awaiting a message not consume processing resources. The task should be rescheduled for execution when the message arrives.
- Because inports and outports may reside on different nodes, a channel is an inherently distributed data structure that must be maintained in a consistent state when its constituent ports migrate.

It is difficult to meet these requirements in an implementation based on the SPMD model supported by many existing message-passing libraries. One reason is the need to distinguish between messages intended for different members of a dynamically varying collection of ports. Another is the inherently *synchronous* nature of the SPMD model: in general, both the sending and receiving node must be aware that a communication is to take place. Yet in task-parallel programs in which multiple tasks are executing on each node, communications are often asynchronous and unpredictable.

Our approach is to rely instead on a model that includes abstractions for both a global name space and asynchronous invocation of operations on objects in that name space. More specifically, we structure our implementation of port mechanisms in terms of three abstractions provided by the Nexus [5] runtime library:

- *Global pointers,* which furnish a global name space, even in distributed-memory execution environments;
- *Lightweight threads,* which facilitate mapping of multiple tasks to a single node and efficient switching between tasks; and
- *Remote Service Requests* (RSRs), which asynchronously invoke a designated handler function on an object referenced by a global pointer. The object may be nonlocal, in which case a message is sent that causes the handler to execute on the object's node. The request may specify arbitrary additional arguments for the handler.

We have found that use of the Nexus abstractions greatly simplifies the structure of the FM runtime: an early prototype based on the SPMD model was significantly more complex. In addition, the portability of Nexus enhances the portability of our runtime system. A potential disadvantage of the approach is that the Nexus abstractions may introduce unnecessary overhead. This issue is addressed in the next section.

4 POINT-TO-POINT COMMUNICATIONS

In this section, we describe our implementation of message-passing operations over ports. We also present measurements of communications latency and

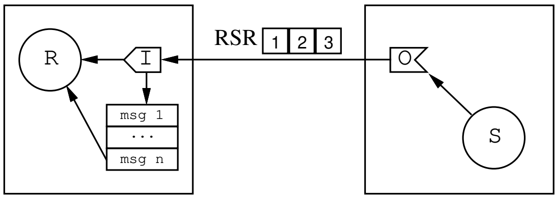


Figure 3 Using Nexus abstractions to implement port communication. Task `S` uses an RSR to send a message from outport `O` to inport `I`, from which task `R` will receive it.

bandwidth, both for our implementation and for the underlying software layers on which it is based. We analyze the overheads imposed by our approach, placing particular emphasis on tradeoffs between functionality and performance.

4.1 Implementation Sketch

Our runtime system uses Nexus abstractions to implement point-to-point communication over a channel as follows (refer to Figure 3):

- FM tasks are represented by threads. Local variables within a task hold references to its ports.
- Ports are data structures that contain global pointer references to other ports. An inport also contains both a message queue and state fields that indicate whether a task is blocked waiting to receive from it.
- The distributed data structures representing channels are constructed by means of global pointers: the inport has a global pointer to the outport (for migration), and the outport has a global pointer to the inport (for communication and migration).
- Sending a message over a channel corresponds to sending an RSR to the inport by using the global pointer contained in the outport. The RSR incorporates the message contents as its additional arguments, and invokes a "send" handler that either buffers or delivers the message, as appropriate. If no receiving task is waiting on the inport, the message is buffered in the queue for later receipt. Otherwise, the message data is copied directly to memory locations previously specified by the waiting receiver, and the receiver is scheduled for execution.

4.2 Communications Performance

Compared with traditional message-passing systems such as MPI and PVM, the port-based communication mechanisms of FM provide substantial additional functionality, such as location transparency, the ability to discriminate among several tasks on a single node, and determinism. However, this added functionality introduces additional costs.

To enhance portability, the FM runtime is designed as a layered system, with each higher layer of software providing communications primitives with increased functionality. Therefore, it is possible to determine the approximate cost of each major piece of functionality by measuring the performance achieved by each successive layer, and then comparing the layers with each other. This is the methodology we employ for determining the costs incurred by our implementation of port mechanisms.

We measured latency and bandwidth using a simple synthetic "ping-pong" benchmark in which both short (one byte) and long (one megabyte) messages were exchanged between two different nodes of Argonne's IBM SP multicomputer. This machine is equipped with an SP2 network switch and SP1 nodes running version 3.2.5 of the AIX operating system. The FM runtime was layered atop a version of Nexus that used IBM's MPL message-passing library for communications and IBM's DCE Posix threads implementation for multithreading. All code was compiled with the most effective optimization options of the `xlf` and `xlc` Fortran and C compilers.

Table 1 contains the latency measurements for each layer of software. Because a detailed performance analysis of the first three layers has been provided elsewhere [5], here we discuss only the functionality provided by these layers. The lowest layer, the MPL message-passing library, provides transport of data between the memories of different nodes. The next layer is the RSR mechanism of Nexus, which provides the ability to invoke a handler function asynchronously when a message is received. At the next highest layer, Nexus threads on different nodes exchange messages through the RSR mechanism. The use of threads makes it possible for multiple tasks to run on each node.

For the highest layer, we measured FM tasks exchanging messages through ports. At this layer, the full semantics of ports becomes available, including message buffering, deterministic execution, in-order message arrival, and transparency of the locations of senders and receivers. We believe that the increase in latency to 25 μsec. more than the previous layer is chiefly due to the cost of acquiring locks for exclusive access to port data structures and to management of inport components such as the message queue.

The data in Table 1 indicate that, compared with the underlying message-passing library on the SP, our implementation of port-based communications has a latency about 2.5 times greater, with an absolute increase of about

Software Layer	Latency (μsec.)	Bandwidth (Megabytes/sec.)
Native message passing (MPL)	66.8	32.2
Handler-to-handler (Nexus)	97.0	15.4
Thread-to-thread (Pthreads)	148.7	15.4
Task-to-task (FM)	173.6	15.4
Interrupt receive (RCVNCALL)	219.8	31.0

Table 1 Point-to-point communication performance for each software layer.

100 μsec. To put this cost into perspective, we also measured the performance of IBM's `RCVNCALL` message driven processing facility. When a message arrives at a node, `RCVNCALL` interrupts the node and invokes a handler function defined by the programmer . Hence this facility provides functionality similar to the Nexus handler-to-handler layer, with the addition of pre-emption, yet it increases latency to 3.3 times, which is appreciably more than the entire additional cost imposed by our implementation of port-based communication.

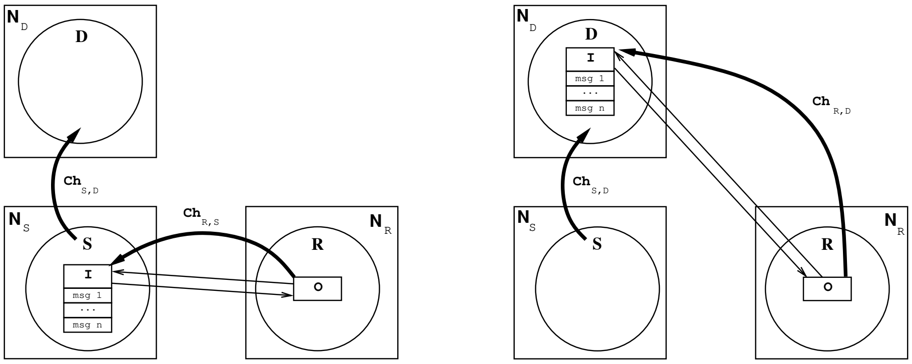
Table 1 also contains bandwidth measurements for each layer. The data clearly show that the thread-to-thread and task-to-task layers add negligible costs in bandwidth, while the Nexus handler-to-handler layer adds a significant per-byte cost that reduces bandwidth to about half that of MPL. In other work [5], it was shown that most of this bandwidth reduction was due to the fact that, for large messages, Nexus does not post receive buffers in advance. By switching to a multiple-message protocol for large messages, in which the length is sent in advance of the data so that a receive of the appropriate size can be posted ahead on the destination node, the bandwidth achieved by Nexus approached that of the underlying message-passing library. We believe that a similar technique can be used to reduce greatly the gap in bandwidth between our runtime system and the underlying MPL layer.

5 PORT MIGRATION

This section describes our runtime system's protocol for migrating ports between tasks. We also present data for the protocol's performance on a synthetic benchmark and discuss factors that contribute to its execution time.

5.1 Migration Protocol

Port migration must preserve any invariant properties of channel data structures needed to guarantee proper semantics for point-to-point communication,



a. Before migration

b. After migration

Figure 4 Channel data structure before and after migration. Thick curved lines denote channels; thin straight lines denote references via global pointers.

such as message ordering along channels and exclusive ownership of a port by at most one task. Figure 4 illustrates how the distributed data structure representing a channel changes as its inport migrates from one node to another. In this example, source task `S` on node $\mathtt{N}_{\mathrm{S}}$ transfers inport `I` to destination task `D` on node $\mathtt{N}_{\mathrm{D}}$ over channel $\mathtt{Ch}_{\mathrm{S,D}}$. Remote task `R` owns an outport `O`, and `I` and `O` together form a channel $\mathtt{Ch}_{\mathrm{R,S}}$.

In general, a protocol for migrating a port `P` must do the following:

- Create a new data structure for `P` at the destination node.
- Update global pointer references of any ports connected to `P` by a channel.
- Forward any messages queued at `P` to the new data structure, taking care that message order is correct when migration completes.
- Transfer ownership of `P` from the sending task to the receiver.

Performance considerations demand that a port migration protocol exploit parallelism where feasible and minimize message passing and synchronization. In designing the protocol described here, we struck a balance between performance and complexity: we parallelize steps that typically dominate execution time, but adopt simple solutions for cases that are expected to occur rarely. For example, we reduce the complexity inherent in simultaneously migrating both ends of a channel by serializing the two migrations.

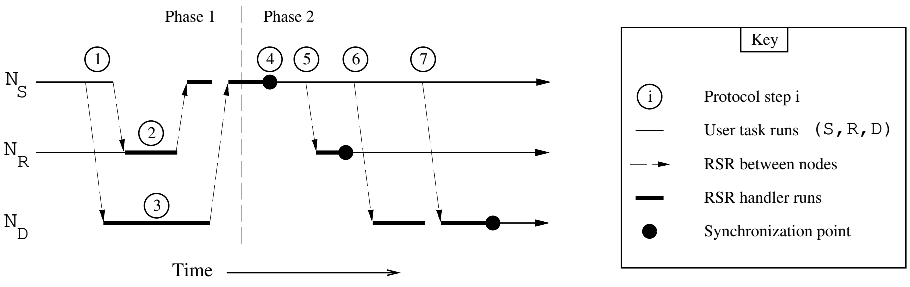


Figure 5 Timeline of execution of migration protocol for example of Figure 4. Timelines for each node are stacked vertically to show partial ordering of events.

We illustrate the operation of the protocol by describing the major steps that occur during its execution. Figure 5 contains a graphical depiction. Our example is the same as that of Figure 4:

1. `S` sends an RSR to node $\texttt{N}_\mathrm{D}$ requesting creation of a new data structure for `I`. `S` also sends an RSR to $\texttt{N}_\mathrm{R}$ to "lock down" `O` so that it will not migrate or send further messages to `I`.
2. `O` becomes "locked down", and an acknowledgment message is sent to `S`.
3. A new data structure for `I` is created on $\texttt{N}_\mathrm{D}$, and a reply message containing a global pointer to the data structure is sent to `S`.
4. `S` waits until the messages sent in Steps 2 and 3 have arrived.
5. `S` sends an RSR to $\texttt{N}_\mathrm{R}$ to release the lock on `O` and update `O`'s pointer to `I`.
6. `S` forwards all messages queued at `I` to the new data structure on $\texttt{N}_\mathrm{D}$.
7. `S` sends to $\texttt{N}_\mathrm{D}$ an RSR which transfers ownership of `I` to `D`. If `D` had been blocked waiting for `I`, it is now unblocked.

Note that there are two round-trip exchanges of messages between `S` and `R` and between `S` and `D` in steps 1 through 4. We expect that in many cases, these round trips will dominate execution time. Hence in our protocol they are performed in parallel. Moreover, the portion of the protocol executed by `S` can be usefully split into two distinct phases: steps before 4, and steps from 4 onward. After `S` has completed its portion of the first phase (Step 1), it can fill the idle time spent waiting for Steps 2 and 3 to complete with other computation, then continue with the second phase at Step 4.

Communication Protocol	Raw Time (μsec.)	Ratio to Latency	Compared with Conservative
MPL	1,105	11.4	~2% faster
TCP/IP	5,571	3.9	~29% faster

Table 2 Execution times for port migration. Column 3 is Nexus handler-to-handler latency.

5.2 Migration Performance

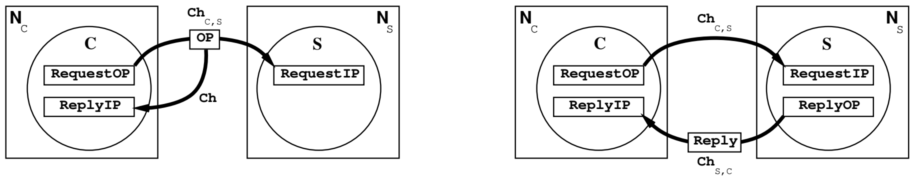
We measured the performance of our protocol using a synthetic "ping-pong" benchmark that repeatedly migrates an inport in a manner identical to that shown in Figure 4. Table 2 shows the average execution time for each migration. The last column compares our protocol with a more conservative implementation that does not parallelize the two round-trip exchanges described above.

If these round trips in fact did dominate execution time, we would expect our protocol to be about twice as fast as the conservative one, and the ratio of migration time to latency to be close to but greater than 2. Yet under MPL, our protocol falls far short of that mark. We surmise that thread switching and poor thread scheduling are among the major factors causing this.

To test our hypothesis, we repeated these measurements, this time using the TCP/IP protocol over the SP's diagnostic Ethernet for communications. Latency was 1,413 μsec., about an order of magnitude higher than for MPL. As the table shows, round-trip latency then dominated execution time, and our parallel protocol gained significantly over the conservative one. The results of this experiment imply that if we can minimize overheads from sources such as threading, our protocol's performance under MPL should improve dramatically.

6 MIGRATION OPTIMIZATIONS

In many instances of port migration, it can be determined *a priori* that the ports affected by the migration will be used in only limited ways. Figure 6 illustrates one example that we commonly encounter in FM programs: A client task `C` sends a message to a server task `S`. The message includes both information about processing to be performed on behalf of `C`, and an outport `ReplyOP` through which `S` sends a reply message containing the results. Because the client needs the results produced by the server, it will not cause inport `ReplyIP` to migrate. Among other examples, this paradigm for task interaction appears when implementing both remote memory accesses of elements of distributed arrays and task farms in which slave tasks request work from a master task, as depicted in Figure 7.



a. Request phase

b. Reply phase

Figure 6 Request and reply phases of client-server interaction.

In Figure 6, because `ReplyIP` does not migrate, it has no need for the global pointer to `ReplyOP` used by the general port migration protocol. Therefore migration of `ReplyOP` can become as simple as a single RSR from $\mathtt{N}_{\mathrm{C}}$ to $\mathtt{N}_{\mathrm{S}}$. This RSR causes a new data structure for `ReplyOP` to be created and transfers ownership of `ReplyOP` to `S`. We expect that such an optimized migration protocol will be only slightly slower than the point-to-point communication latency and therefore considerably faster than the general migration protocol.

Opportunities for employing such an optimization can in principle be detected by a compiler, if appropriate analyses are performed. Alternatively, the programmer can supply annotations which indicate when to use the optimization. We plan to modify the FM compiler to process such annotations, and will extend our runtime system to include this optimized protocol.

Figure 7 Examples of client-server interaction paradigm.

7 RELATED WORK

Channel-based communication in task-parallel programs was introduced in Communicating Sequential Processes [10], and implemented in occam [11]. Our port-based communication mechanisms are significantly more general, supporting, for example, the dynamic creation of tasks and channels and reconfiguration of channels via port migration. In this respect, an FM channel has similarities to shared logical variables in concurrent logic programming languages [6]. An important difference is FM's guarantee of determinism.

Communication ports in FM are similar in certain respects to those of the Mach operating system, in that the physical location of ports is transparent to tasks, and capabilities to access ports may be passed through ports [4]. Yet Mach permits both reading and writing on the same port, while FM distinguishes between inports and outports. In addition, multiple writers may utilize a Mach port, whereas FM permits only a single writer at a time for each inport.

The port-based communication model described in this paper separates the concepts of communication endpoint and task, hence allowing a message to be sent to a specific task, without knowledge of the identity or location of that task. An alternative approach to intertask communication, adopted in Chant [8], is to augment traditional message-passing libraries with task identifiers. A send operation then specifies not only a destination node but also a destination task on that node.

8 CONCLUSIONS

We have found port-based communication useful for solving a number of interesting parallel programming problems. Our implementation of point-to-point communication and port migration mechanisms proved particularly natural when expressed in terms of the Nexus asynchronous invocation and global name space abstractions. Performance analysis of this implementation indicates that the cost of message delivery to individual threads need not be excessive compared to traditional message passing, but nevertheless could be improved by a tighter integration of threading and communication. Our implementation of general port migration still suffers from relatively large overheads, despite a significant design and tuning effort. Fortunately, we have discovered many instances in which faster special-case migration protocols can be employed instead.

Most of the techniques described in this paper have been incorporated in the Fortran M compilation system developed at Argonne, which is available at the World Wide Web URL `http://www.mcs.anl.gov/fortran-m/`. The parallelized version of port migration will be incorporated shortly. In future work, we will use more substantial task-parallel applications to evaluate port communication and migration performance and to investigate the utility of special-case protocols. We also wish to explore interoperability through port-based communication between FM and other parallel languages such as Compositional C++ [2].

Acknowledgments

This work was supported in part by the National Science Foundation's Center for Research in Parallel Computation under Contract NSF CCR-8809615 and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38. We are grateful to members of the Nexus development team for their assistance with the Nexus software.

REFERENCES

[1] K. M. Chandy and I. Foster. A deterministic notation for cooperating processes. *IEEE Trans. Parallel and Distributed Syst.*, 1995. To appear.

[2] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, 1993.

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

[4] M. Young et al. The duality of memory and communication in Mach. In *Proc. of the 11th Symposium on Operating Systems Principles*, 1987.

[5] I. Foster, C. Kesselman, and S. Tuecke. Portable mechanisms for multithreaded distributed computations. Mathematics and Computer Science Division Preprint P494-0195, Argonne National Laboratory, 1995.

[6] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1989.

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

[8] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: A talking threads package. In *Proc. of Supercomputing 94*, November 1993.

[9] S. Hiranandani, K. Kennedy, and C. Tseng. Preliminary experiences with the Fortran D compiler. In *Proc. of Supercomputing 93*, Portland, November 1993.

[10] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1984.

[11] G. Jones. *Programming in occam.* Prentice Hall International, 1987.

[12] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, 1990.

[13] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A mechanism for integrated communication and computation. In *Proc. 19th Int'l. Symposium on Computer Architecture*, May 1992.