A Communication Library to Support Concurrent Programming Courses[*]

Steve Carr, Changpeng Fang, Tim Jozwowski, Jean Mayo and Ching-Kuang Shene
Department of Computer Science
Michigan Technological University
Houghton, MI 49931
Email: {carr, cfang, trjozwow, jmayo, shene}@mtu.edu

Abstract

A number of communication libraries have been written to support concurrent programming. For a variety of reasons, these libraries generally are not well-suited for use in undergraduate courses. We have written a communication library uniquely tailored to an academic environment. The library provides two levels of communication abstraction (topology and channel) and supports communication among threads, processes on the same machine, and processes on different machines, via a unified interface. The routines facilitate controlled message loss along channels and can be integrated with an existing graphical tool that supports visualization of the communication that occurs. An editor has been developed for automatic code generation for arbitrary topologies via a graphical interface. All these tools run over Solaris, Linux, and Windows.

1 Motivation

Concurrent programming is increasingly fundamental to undergraduate Computer Science education [1]. Correspondingly, courses dedicated to, or containing a component in, this area are moving ever earlier into the undergraduate curriculum. Yet this remains a very challenging subject to teach. Aside from the difficulty of the material, available tools generally are not tailored to an academic environment.

In our experience, a significant hurdle to student understanding, especially among lower-level students, is the complexity and diversity of communication interfaces. Students likely learn separate interfaces for synchronized communication among threads (lightweight processes), processes (heavyweight processes) on the same machine, and processes on different machines. Another difficulty faced in teaching networked communication in particular is the introduction of message loss in some controlled fashion.

In order to address these issues, we have developed a library to support communication among threads, processes on the same machine, or processes on different machines, via a unified interface. These routines implement an abstraction of the primary overarching characteristics of IPC (interprocess communication). They facilitate the study of concurrent application design and can serve as a starting point for study of the implementation of IPC within a particular paradigm, threads, processes on the same machine, or processes on different machines. The library abstracts the passing of messages at two levels: topology (the highest level) and channel. Additionally, the routines provide a mechanism for introducing message loss in a controlled fashion. These routines can be integrated into an existing visualization system that depicts the communication that takes place. A topology editor has been developed that facilitates automated generation of code for arbitrary topologies using a graphical interface. The routines, visualization system, and topology editor run over Solaris, Linux, and Windows.

2 Related Work

Arnow developed the XDP message passing library for teaching distributed programming [2]. The goals of this library were more narrow than our goals in developing the tools described in this paper. The XDP library abstracts away some of the complexity of the BSD socket interface, in order to reduce the course time required to cover a network programming interface while requiring that students still address fundamental problems such as buffering, race conditions, synchronization, and reliability. The library does not attempt to provide multiple

[*]This work supported in part by the National Science Foundation under grants DUE-9752244 and DUE-9952509. The fourth author was also supported by National Science Foundation CAREER award CCR-9984862.

levels of abstraction, controlled message loss, or integrated visualization support.

Other, commercially used message passing libraries are available, e.g. PVM and MPI. MPI is perhaps the most widely used, and structuring our message passing library around the MPI interface (adding support for visualization and maintenance of vector time over the MPI primitives) was considered. However, at the time our development began, publicly available implementations of MPI, like XDP, required that the same code be executed for each process comprising an application. This made it unsuitable for distributed or threaded application development. Additionally, we did not want to make installation and maintenance of MPI or PVM a requirement for the use of our system.

McDonald and Kazemi have extended the PVM and MPI message passing environment to support *virtual process topologies* [10]. Several core functions have also been developed to enable a parallel program to request use of a standard process topology, to spawn and instantiate all tasks participating in a topology, and to specify transmission, reception, and synchronization in terms of logical communication patterns, eliminating, for example, the need for students to compute process identifiers. They also provide a graphical interface for specifying, verifying, and viewing topologies. Hence, their tools are similar to what is achieved by our topology classes and editor. Our tools additionally provide controlled message loss and execution visualization.

3 Communication Library

Communication is abstracted at two levels: channel and topology. The two abstractions are described, in turn, below.

Channel The goal of the channel classes is to provide an abstraction of communication that ties closely to that encountered in the literature. Three channel types have been implemented. The first class is a synchronous one-to-one channel. Along this channel, both send and receive are blocking [5]. Addressing in this class, as in all the classes, among threads is by PThreads thread identifier[1] and among processes is by integer identifier. Process identifiers are either assigned implicitly when application processes are started by a control process, described later, or can be assigned explicitly by the user when a channel is created. No attempt is made to prevent deadlock caused by application communication patterns, and the routines will block indefinitely. Message loss cannot be introduced artificially into synchronous channels.

[1] *Pthreads* refers to thread implementations that adhere to the POSIX standard P1003.1c.

```
char msg[]="False pearls before real swine";
channel1 = new AsynOnetoOneChannel(1,
                       myID,dropSome(rand()));
channel1.send((void *)msg,sizeof(msg));
```

(a) Sender

```
char msg[MSGLEN];
channel0 = new AsynOnetoOneChannel(0,myID,0.5);
channel0.recv((void *)msg,sizeof(msg));
```

(b) Receiver

Figure 1: Message Transmission along Asynchronous Channel

The second class implements an asynchronous one-to-one channel. Along these channels, sends are non-blocking; two receive primitives are provided, one blocking and one non-blocking [5]. Message loss can be introduced along any asynchronous one-to-one channel between processes. The loss can be specified either as a value between zero and one or via an integer function, with a single integer input, at the time the channel is created. When the loss is specified as a value between zero and one, messages will be dropped, immediately prior to the point at which they would be sent along a totally reliable channel, by the communication layer according to a uniform distribution. When loss is specified as a function, the function is evaluated at this same point immediately prior to the send operation. If the return value from the user-supplied function is greater than or equal to one, the message is sent, otherwise the message is dropped. (Hence, the loss function is directional.) Figure 1 depicts code that creates a channel between the processes with identifiers zero and one; process zero then sends a message to process one. Messages sent by process zero are dropped according to the function `dropSome()` (which is assumed to be defined elsewhere). Half the messages sent by process one (to process zero) are dropped according to a uniform distribution.

Implementation of this code using a BSD socket interface would require approximately twice as many calls as the two (constructor and send) required here. This estimate neglects the additional, non-trivial, complexity required to support process addressing via integer identifier and to drop messages in a controlled fashion.

The final class is a many-to-many channel, and is currently available only for threads. This class essentially implements a bounded buffer.

Each class has a method that allows a channel to be queried for data available to read. The return has a

```
theGrid = new Grid(ROWS,COLS,myID);

if ((myID % COLS) != 3)
   theGrid.send(RIGHT,(void *)&myID,sizeof(myID));
if ((myID % COLS) != 0)
   theGrid.send(LEFT,(void *)&myID,sizeof(myID));
if (myID >= COLS)
   theGrid.send(UP,(void *)&myID,sizeof(myID));
if (myID < (ROWS-1)*COLS)
   theGrid.send(DOWN,(void *)&myID,sizeof(myID));
```

Figure 2: Exchange of ID Among Grid Neighbors

value of one when data is available, but has no effect on the channel itself. If no message is available, the return value is zero.

Vector time is maintained within user applications. Vector time is used to determine the happened-before relation [8] among events that occur within a distributed computation [9]. Users can query, and increment the local component of, the current local vector time within an application.

Topology One-to-one channels can be joined into topologies. The primary function of the topology class is to facilitate creation of multiple channels via instantiation of a single class. Several standard topologies, derived from the topology class, are also provided including: fully-connected, star, linear array, ring, grid, and torus. Message sends and receives are restricted to directly connected nodes within the topology. For example, the center node of a star network is the only node able to send to, and receive from, outer nodes; outer nodes can only send to, and receive from, the center node. Topologies are built with reliable asynchronous channels. Figure 2 depicts exchange of identifiers among all neighbors in a grid topology. Note that macros `RIGHT,LEFT,UP,DOWN` are defined within our system. Similar macros are defined as appropriate to a given topology.

Within the topology class, and each standard topology, the methods `Send()`, `Receive()`, `Broadcast()`, `Scatter()`, `Gather()`, and `Reduce()` are provided. The `Send` (`Receive`) routine sends (receives) a message to (from) a specified process. `Broadcast` effects a broadcast, to all processes, of a specified message from a specified source process. `Scatter` partitions an input block of data, from a specified source process, into a number of pieces equal to the number of processes and sends a unique piece to each of the application processes. `Gather` complements `Scatter` and collects data from application processes to the process with a specified identifier. A `Reduce` method collects data from each application process and stores the result in a specified location. The user specified function is then run to reduce the collected data. `Scatter`, `Gather`, and `Reduce` currently operate on one-dimensional arrays (consecutive storage). Support for operations on some non-consecutive storage in two-dimensional arrays (e.g., submatrix) is under development.

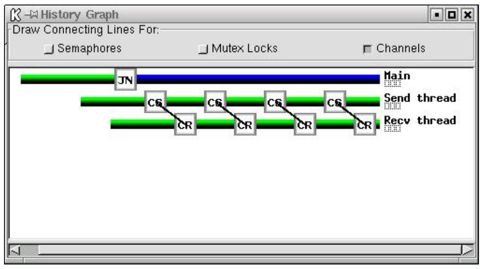

Figure 3: History Window

Implementation Applications comprised of multiple processes are spawned via a central control process. Specification of the user programs and the machines on which they should execute takes place either via the command line or from user-specified configuration files. The control process also spawns the visualization process, when requested. Visualization data is passed along TCP channels between the user processes and the visualization process. When channels (or topologies) are used, a TCP connection between the user processes is created to effect a channel. Hence, the control process is not involved in communications between user processes.

4 Topology Editor

A topology editor has been created to facilitate rapid development of complex topologies via a graphical interface. The editor allows creation of connections among single nodes or among topologies. The editor output is a file containing specification of a class derived from the topology class. (The derived class name can optionally be specified by the user.) The interface for this class is equivalent to that for the topology class. This file can be included by the user in her code to easily create the constructed topology. The derived class supports broadcast, scatter, gather, and reduce functions for each custom topology.

5 Visualization

A visualization system has been developed for visualizing synchronization among the threads of an executing application [3]. This system has been extended to depict the communication that occurs among threads or among processes. This visualization is linked to use of the channel class, and hence is available when the channel, topology, and specific topology classes are used.

A *History Graph* window (see figure 3) depicts the sends and receives that occur within each process or thread, and connects corresponding send and receive operations between threads or processes. Clicking on any channel in this window opens the *Channel* window. This window displays all recent activity along this channel, including channel type, messages in the channel, messages received since the window was opened, and current status, either *Sending Message* or *Receive Message*.

6 Experience

The channel classes closely follow the abstractions of communication found in the literature and they are easily incorporated into existing assignments. Their use provides the additional advantage of allowing students to visualize the communication that takes place.

```
with1 = new SynOneToOneChannel(1,0);
msg.done = 0;

do {
   msg.value = max(mySet);
   sentValue=msg.value
   with1.send((void *)msg,sizeof(msg));
   remove(mySet,msg.value);
   with1.recv((void *)msg,sizeof(msg));
   add(mySet,msg.value);
} while (sentValue > msg.value);

msg.done=1;
with1.send((void *)msg,sizeof(msg));
```

(a) Solution for Process Zero

```
with0 = new SynOneToOneChannel(0,1);

do {
   with0.recv((void *)msg,sizeof(msg));
   if (msg.done == 0){
      add(mySet,msg.value);
      msg.value = min(mySet);
      with0.send((void *)msg,sizeof(msg));
      remove(mySet,msg.value);
   } while (msg.done==0);
```

(b) Solution for Process One

Figure 4: Set Partition

We present a sequence of exercises that result from our experience with teaching network programming over the past several years in an upper-level undergraduate course. These exercises were recently developed to serve as the first set of network programming exercises. Initially, our first network programming assignment was more complex. We typically required an application that contained multiple clients and a server for all client types, similar to that described in [4]. While they report large-scale success, we have found that, while many students are able to complete the assignment, a significant number of students have difficulty. We hope that completion of these exercises will lead to greater success in the more complex client server exercise.

This set of exercises was designed to demonstrate three fundamental aspects of concurrent application design: (1) the use of synchronous versus asynchronous communication, (2) deterministic versus non-deterministic communication, and (3) the use of a client-server architecture versus a fully distributed one. The basis of the exercises is Soundararajan's CSP [7] implementation [11] of a set partitioning problem [6]. The problem is to partition a set of integers into two sets according to the element values. Each process (heavyweight or lightweight) initially has half of the set. One process P_0 will end up with the lower half of the elements and the other process P_1 ends up with the upper half of the elements.

The first exercise requires that the problem be solved using synchronous message passing. P_0 sends the maximum value $max(S_0)$ from its set S_0 to P_1 and removes $max(S_0)$ from S_0. P_0 then waits to receive the minimum value $min(S_1)$ from the set S_1 of P_1. This continues until P_0 receives a value that is greater than or equal to the one it sent.

Upon receiving a value from P_0, P_1 adds the received value to S_1. P_1 then sends $min(S_1)$ to P_0 and removes $min(S_1)$ from its set. This continues until P_0 notifies P_1 that the set is partitioned. A solution is given in figure 4. This exercise illustrates development of a simple application protocol. (P_0 and P_1 must agree on the format of messages, and agree on a sentinel message that lets P_1 know that no further messages will be sent.) It also serves to familiarize students with the (synchronous) message passing interface.

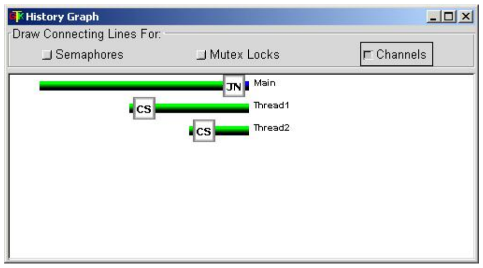

Figure 5: Simultaneous Synchronous Sends - History Window

The second exercise requires a solution similar to that for exercise one, with the exception that, at each step, P_0 and P_1 send their maximum and minimum values, respectively, simultaneously. When P_0 (P_1) receives a value less than (greater than) the one it sent out, the sent value is removed from its set and the received value is added. When P_0 (P_1) receives a value greater than (less than) the one it sent out, the partition is complete and no set modifications are made. We do not specify the use of a particular channel class. Development of a solution requires that students come to the realization that only asynchronous message passing facilitates simultaneous execution of a send by both P_0 and P_1. If synchronous communication is chosen by the student, the problem quickly presents itself within the visualization system, as depicted in figure 5.

The final exercise of this sequence incorporates N processes with portions of the set and requires a centralized solution. Students use a supplied non-deterministic receive function (or can create this function for themselves) that listens for data on any incoming channel. A server collects a set of integers from all clients, computes the set partition, and returns the resulting sets to the clients.

7 Conclusions and Future Work

We have developed a message passing library that provides two levels of abstraction, channel and topology, for the communication that occurs among processes and threads. Tightly integrated visualization support is available, as is support for controlled message loss. A topology editor allows development of custom topologies via a graphical interface. We anticipate that these communication classes and associated tools will support the instruction of concurrent programming by reducing the overhead associated with learning message passing interfaces, by providing a uniform interface for communication both among threads and among processes, and by providing integrated visualization support without the need for instrumenting user programs.

These message passing classes are part of a larger system that provides a class library for threads, thread synchronization, and message passing. The system currently also has support for visualizing the synchronization of threads and the message passing that occurs among threads and processes. We are currently adding support for synchronization of processes (barrier and mutual exclusion), implementing well-known parallel and distributed algorithms, adding support for distributed arrays, and adding additional visualization support specifically for parallel and distributed programming. We believe the tools can be used at any level in which students have the programming sophistication and background sufficient to cover concurrency. We have taught thread and network programming in a course populated predominantly by sophomores and juniors. The system has not been used at a lower level. Comprehensive, detailed information on our work is available at `http://www.cs.mtu.edu/~shene/NSF-3/index.html`.

References

[1] ACM. Computing Curricula 2001 (Steelman Draft, August 1, 2001). `http://www.acm.org/sigs/sigcse/cc2001/steelman/`, 2001.

[2] Arnow, D. M. A simple library for teaching a distributed programming module. In *Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education* (Nashville, Tennessee, Mar.2-4 1995), pp. 82–86.

[3] Bedy, M., Carr, S., Huang, X., and Shene, C.-K. A visualization system for multithreaded programming. In *Proceedings of the 31st Annual SIGCSE Technical Symposium on Computer Science Education* (Austin, TX, March 2000), pp. 1–5.

[4] Bryant, R. E., and O'Hallaron, D. R. Introducing computer systems from a programmer's perspective. In *Proceeding of the Thirty-second SIGCSE Technical Symposium on Computer Sciense Education (SIGCSE-01)* (New York, Feb.21–25 2001), pp. 90–94.

[5] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed Systems Concepts and Design*, third ed. Addison-Wesley, 2001.

[6] Dijkstra, E. W. A correctness proof for networks of communicating sequential processes - a small exercise. EWD-607, 1977.

[7] Hoare, C. Communicating sequential processes. *Commun. ACM 21*, 8 (1978), 666–677.

[8] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[9] Mattern, F. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, M. C. et. al., Ed. Elsevier Science Publishers B. V., 1989, pp. 215–226.

[10] McDonald, C., and Kazemi, K. Teaching parallel algorithms with process topologies. In *Thirty-first SIGCSE Technical Symposium on Computer Science Education* (Austin, Texas, Mar.7-12 2000), pp. 70–74.

[11] Soundararajan, N. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems 6*, 4 (1984), 647–662.