First Class Communication in MPI

Erik Demaine
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
eddemaine@uwaterloo.ca

Abstract

In this paper we compare three concurrent-programming languages based on message-passing, Concurrent ML (CML), Occam, and MPI. The main advantage of the CML extension of Standard ML (SML) is that communication events are first-class just like normal program variables (e.g., integers), that is, they can be created at run-time, assigned to variables, and passed to and returned from functions. In addition, it provides dynamic process and channel creation. Occam, first designed for Transputers, is based on a static model of process and channel creation. We examine how these limitations enforce severe restrictions on communication events, and how they affect the flexibility of Occam programs. The MPI (Message Passing Interface) standard provides a common way to access message-passing in C and Fortran. Although MPI was designed for parallel and distributed computation, it can also be viewed as a general concurrent-programming language. In particular, most Occam features and several important facilities of CML can be implemented in MPI. For example, MPI-2 supports dynamic process and channel creation, and less general first-class communication events.

We propose an extension to MPI which provides the CML `choose`, `wrap`, *and* `guard` *combinators. This would make MPI a strong base for the flexible concurrency available in CML. Assuming that the modifications are incorporated into the standard and its implementations, higher-order concurrency and its advantages will become more widespread.*

1. Introduction

Recently a large amount of interest in *concurrent programming languages* has developed. This has resulted in a variety of systems that provide concurrency under several models. Common to these models is the notion of *processes* that conceptually run in parallel, each with its own *thread of execution* (including state and code). The models can be classified by how processes are created, how they communicate, and how communication devices are created.

Processes can either be *statically created*, *statically determined*, or *dynamically created*. Static determination, where the number of processes is bounded before program execution, is much less general than dynamic process creation, where processes can be generated on-the-fly. A further restriction is static process creation, where all processes are allocated prior to program execution. This is in contrast with static determination which allows processes to be dynamically created, although the pattern must be restricted by the program text. It is possible to assume a bound on the number of processes, and potentially use less. However, programming in this way is very difficult, resembling programming dynamic data structures in a language where dynamic memory allocation is not provided [10].

We will focus on concurrent languages that use the *message-passing* or *distributed-memory* paradigm. This means that processes have their own address space, and communicate by sending *messages* to one another. This is in contrast to *shared-memory* systems, where processes share a global address space for communication. The reason we emphasize distributed-memory languages is that they are usually more efficient, even on shared-memory multiprocessors [7].

For one process to send a message to another, the destination (and likely the source) must somehow be specified. Using the process *name* (for example, a host identifier along with a UNIX pid) is the simplest way to do this. A more general approach is to assign a series of *ports* to each process (typically identified using a single integer), and send messages to individual ports. This makes it easier for the receiver to distinguish pending messages.

One more level of abstraction, called *channels*, is also very useful. A channel is a uni-directional connection between two processes. A process at the receiving end of several channels corresponds to the port notion above. Channels can be used to abstract which process is "at the other end." As with processes, channels may be statically determined or dynamically created.

The send and receive primitives can be supported in a variety of ways. We shall focus on *synchronous* operations, where the sender (receiver) waits for a matching receive (send) to continue. Synchronous message-passing originated from *CSP* (Communicating Sequential Processes) [5]. Although synchronous message-passing can be implemented using only asynchronous message-passing via busy waiting, the reverse can also be done via buffering. Hence, we find that synchronous communication is a good base for concurrent-programming languages. We will call synchronous communications *events*.

The following section discusses the first-class events of Concurrent ML. In Section 3, we examine the facilities of Occam and see why they are restrictive. Section 4 describes the MPI standard and the first-class communication requests that it provides. We see how Occam programs can be converted into MPI-2 programs in Section 5. Section 6 proposes additions to MPI that provide features similar to CML. Finally, we conclude in Section 7.

2. Concurrent ML

Reppy [9] has developed a powerful concurrent language called *Concurrent ML* (CML) based on the New Jersey implementation [1] of the Standard ML (SML) [8] language. One of SML's key features is that functions are *first-class* just like normal program variables (e.g., integers or booleans) in that they can be created at runtime, assigned to variables, and passed to/returned from functions. Following this idea, CML defines processes, channels, and events all to be first-class. Furthermore, they are garbage-collected whenever they are no longer required.

Processes are dynamically created using the `spawn` function. Given a `unit` $\rightarrow$ `unit` function (the code), it returns an identifier representing a new process that executes the given code. By (dynamically) creating a channel in the scope of the process's code, one can achieve parent-child communication. This can be done with the `channel` function, which takes no arguments. All CML channels have a type, so that only values of that type may be transmitted through a channel.

Events are of the type `event`; they represent communications that could be done, not ones that are in progress. Synchronous send and receive events can be created using a `transmit` and `receive`, respectively. `sync` takes an event, fully executes it, and returns the event's result. Thus, `send` and `recv` operations are simply `sync` $\circ$ `transmit` and `sync` $\circ$ `receive`, respectively.

One way to combine events is using `wrap`, which "wraps" an event and a function of the event's result type, creating a new event. Note that the `event` type is parameterized by its return value's type. For example, `receive` returns a value of the type of its channel, and `transmit` returns a `unit`. When a wrapped event is given to `sync`, the event is `sync`'d and then the function is applied to the result. A wrapped event is considered "committed" when the event completes. This means that once the event completes, the function must be fully evaluated, even if it blocks.

Another event combinator is `guard` which takes a function of `unit` returning an event, and creates an event representing that function. When `sync` is applied to a `guard` event, the function is called and the result is given to `sync`. A guarded event is considered "committed" only after the function's result is `sync`'d. Thus, by mixing `wrap` and `guard` we can create an arbitrary *commit point* for the manipulation of several events.

The `choose` operator returns an event based on a list of events (all returning the same type). Evaluating it (with `sync`) involves a non-deterministic choice of the events. The first one that commits will be fully evaluated, and its result will be returned. These non-deterministic choices are useful for handling multiple channels, since it is not known which events will occur first, and choosing the wrong one may cause delay and possible deadlocks.

A problem with using `guards` with `choose` is that evaluating the function will cause the entire `choose` to block if there are blocking operations (such as `send` above) in it. To solve this problem, we can create a process to evaluate the blocking operation. For example, suppose that we want to request something through the `req` channel and get a reply from `reply`, or get an abort message from `abort`, causing the request/reply to be canceled. Then we could use the following CML code:

```
fun g () = spawn (fn () ⇒
  (send (req, x); receive reply))
sync (choose [guard g, receive abort])
```

CML has enormous flexibility to create new forms of communication. For example, we can create a general request-reply as follows:

```
fun reqreply (c1, req, c2) =
  wrap (transmit c1 req, fn () ⇒ recv c2)
```

Note that this function returns an event, and does not do any actual communication. This way the result can be used as part of a `choose`, or get `wrap`ped, etc. The generality obtained using first-class events is an important part of CML.

A summary of the covered CML event-creation functions is given in Figure 1. For details on more CML functions, see [9].

```
receive : 'a chan -> 'a event                (* receive-message event *)
transmit : ('a chan * 'a) -> unit event      (* send-message event *)
sync : 'a event -> 'a                        (* evaluate an event *)
wrap : ('a event * ('a -> 'b)) -> 'b event   (* modify event result *)
guard : (unit -> 'a event) -> 'a event       (* delay, forced by sync *)
choose : 'a event list -> 'a event           (* non-deterministic choice *)
```

Figure 1. Types of basic CML event-creation functions.

3. Occam

Occam [6] is a concurrent programming language based on EPL (Experimental Programming Language) [2] and CSP (Communicating Sequential Processes) [5]. It was originally designed as a high-level assembly language for Transputer systems, providing concurrency via language primitives. We will discuss Occam version 2.

To write a sequential piece of the program, statements (including assignments, channel I/O, etc.) cannot simply be listed in sequence. Instead, any sequential code must be prefixed by the `seq` construct and indented two spaces. Similarly, to make statements execute concurrently, `par` can be used in place of `seq`. Hence, `par` acts as the process creation routine; since the code is static, Occam enforces static process determination. The reason for this is that the language was partly designed for embedded systems, where programs must have bounds on resources such as memory and processes. This also means that recursion and dynamic memory allocation are not supported.

The `alt` construct provides non-deterministic selection, similar to a second-class version of CML `choose`. Clauses in an `alt` can either be prefixed by a channel input operation (`?`), a channel input operation guarded by a boolean expression, or a boolean expression with a `skip` operation (meaning that if the expression is true, the `alt` will automatically succeed). A guarded prefix (channel input or `skip`) is only used in the selection if the corresponding boolean expression evaluates to true. Again, since the code is static, the clause list's members must be exhaustively specified, and can only be removed using a boolean expression. Furthermore, `alt` does not allow output operations even though they are blocking. These limitations do not exist in CML, since the `choose` function takes an arbitrary `event list`.

First-class typed channels are supported by the `?` and `!` input/output symbols, and are created using the `chan` datatype (hence they are statically determined). The ends of channels are implicitly specified in the code, as with CML. They should be declared before a `par`; exactly one subprocess can read from the channel, and exactly one can write to it.

Timers can be declared using the `timer` declaration. Accessing a timer is similar to reading from a channel; for example, `clock ? t` will place the current time according to `clock` into the integer variable `t`. The `after` operator will return true whenever two given times are in chronological order. Using some extra syntax, `clock ? after t1 plus t2` will wait until the time according to `clock` is `t2` units past `t1`. This can be used as a timeout in an `alt` statement.

To summarize, a simple process that multiplexes the data on two channels into a final channel is given by the following Occam code. When it times out (sitting idle for 100 units), it sends a "ping" message to the output channel.

```
-- "chan of byte c1,c2,c3" occurs earlier
timer clock:
int t:
byte x:
while true
  clock ? t -- get current time
  alt
    c1 ? x
      c3 ! x -- feed c1 data into c3
    c2 ? x
      c3 ! x -- similarly for c2
    clock ? after t plus 100
      c3 ! 'P' -- write 'P' on timeout
      clock ? t
```

Note that if the process on the other end of `c3` does not perform a read on the channel, then the above process will block, causing writers of `c1` and `c2` to block. In CML, this can be easily solved using the following code (assuming that arrival order is not important):

```
fun out (x) =
  spawn (fn () => send c3 x)
fun loop () =
  sync (choose [wrap (receive c1, out),
                wrap (receive c2, out)])
  loop ())
```

However, since dynamic process spawning is not supported in Occam, it is impossible to achieve this.

4. MPI

MPI (Message Passing Interface) [3, 4] has become a de facto standard for message passing in C and Fortran. The main advantage of MPI is portability. It has been implemented (both commercially and in public domain) on a variety of parallel computers and networks of workstations.

MPI provides a variety of features, including one-to-one (asynchronous and synchronous send and receive) and one-to-many (multicasting, synchronization, and reduction) communication. In addition, it provides a significant amount of support for process groups and communicators (defined below). As we shall see, these can be used to implement channels.

MPI 1.1 only provides a static process-creation model. This severely limits the applications suitable for MPI. In addition, MPI 1.1 does not provide a timeout facility. For these and other reasons, MPI-2 is being developed. Although not finalized, mechanisms for dynamic process creation have been designed.

We shall now discuss some basic send and receive operations that MPI 1.1 supports. In later sections, we will see how some Occam and CML features can be implemented in MPI. Although MPI is normally only considered a message-passing library, we will think of it as a concurrent programming language.

The `MPI_Ssend` and `MPI_Recv` functions provide synchronous send and receive operations, respectively. In CML terminology, these have explicit `sync`'s, and so correspond to the `send` and `recv` of Section 2. On the other hand, `MPI_Ssend_init` and `MPI_Recv_init` do not perform any communication; instead, they return first-class *communication requests*. These correspond to CML `events` returned by `transmit` and `receive`.

The requests created by `MPI_Ssend_init` and `MPI_Recv_init` are called *inactive*. They can become *active* using the `MPI_Start` routine. This will initiate the communication, but will not block the process. For this blocking to occur, `MPI_Wait` must be called, which pauses until the request becomes inactive again (in which case status information and received data can be retrieved). Thus, the equivalent of CML's `sync` can be accomplished by an `MPI_Start` followed by `MPI_Wait`.

We can already see one limitation of MPI's first-class requests. Although they are handles and may be passed into functions, returned as values, and assigned to variables, they cannot be simulataneously used more than once. This means that `MPI_Start` cannot be called twice on the same request handle unless `MPI_Wait` is called in between. This can be a problem if we want to replicate requests, for example, guarding one request by two different functions, and then non-deterministically choosing one of them. It is possible that the two copies of the same event would be `MPI_Start`ed, which is illegal.

MPI provides a second-class operation `MPI_Waitany` similar to CML's `choose`. That is, given an arbitrary list of active requests, it waits for one to become inactive and returns it. This can be used in conjunction with `MPI_Startall` which activates a list of requests. Thus, `MPI_Waitany` involves an implicit `sync` (in CML terminology), so that `MPI_Waitany` does not return a request representing non-deterministic choice as CML's `choose` does.

All MPI communication routines (including `MPI_Ssend_init`) require a *communicator* object. This consists of two significant parts: a *context* and a *group*. Each communicator has a context in the sense that a send cannot match a receive unless they use the same communicator. In addition, an arbitrary set of processes (called a group) is associated with each communicator. This means that only these processes have access to the communicator. An interval of natural numbers starting at zero is assigned to processes to yield unique identifiers. The MPI system begins with a default communicator called `MPI_COMM_WORLD`, whose group is all processes.

In the next section we will see how communicators can be used to implement channels.

5. Implementing Occam in MPI

In this section we discuss how most of Occam's features can be implemented using MPI. We need equivalents for the following constructs: `par`, `alt`, channels (`?` and `!`), and timers.

Channels can be viewed as communicators with two-process groups. These can be created by calling `MPI_Comm_create` with the group returned by `MPI_Group_incl`, which takes a list of processes in another communicator (such as `MPI_COMM_WORLD`) to put into a group. The sender of the channel communicates with process number 1 of the group to the channel, whereas the receiver "talks" to process number 0. Since communicators (and groups) are first-class and dynamically created, so are these channels. Hence, MPI communicators are generalizations of channels.

The `par` construct can be implemented in MPI-2 using the `MPI_Spawn_multiple` routine, assuming that the processes are compiled as separate programs. This routine creates new processes with their own `MPI_COMM_WORLD`, and returns an *intercommunicator* allowing communication between the two groups of processes. Using this, the two `MPI_COMM_WORLD`'s can be merged using `MPI_Intercomm_merge`. Once this is complete, any necessary channels can be set up.

When `?` and `!` appear as statements, they can be converted to `MPI_Recv` and `MPI_Ssend` calls, respectively. In an `alt` construct, `?` would first be converted to its `MPI_Recv_init` equivalent; this request would then be added to a list. If the input is guarded by a boolean expression, then it is only added to the list when the expression evaluates to true. Finally, in the case of a `skip` instruction guarded by a boolean expression, nothing is done if the expression evaluates to false, and the corresponding code is executed if true is found. Assuming that the latter case does not occur, the list of requests is given to MPI_Waitany (after MPI_Startall is called).

There is a slight complication once MPI_Waitany completes. All of the requests except the chosen one must be reset to inactive via MPI_Cancel. Unfortunately, some of these requests might have already completed. For this reason, a small protocol must be implemented to determine if a received message was accepted. Once a message is sent, a boolean value is received; if it is false, the message is re-sent. The process executing MPI_Waitany sends a "true" value back to the sender of the chosen received message, and failed-to-cancel receives cause a "false" value to be sent.

A more efficient protocol can be implemented so that MPI_Cancel is not required. Essentially messages are sent to potential receivers, and then receivers reply with a flag of whether or not they care to receive. Work is currently underway to implement this protocol to provide Occam-like features in MPI.

At the moment, MPI provides no support for timeouts, making it difficult to handle timers. However, we can view a timer as a channel connecting to a "clock process." This process can accept requests of the form "notify me when the clock reaches time t" (which can be canceled) or "what time is it?" on a channel. All requests come from the process that created the timer (via a timer declaration which converts to an MPI_Spawn). The requests would be handled using MPI_Wtime (and busy waiting), which returns the current time relative to some time in the past (representing the value of a timer). This result can simply be passed back through another channel. When a notification request is made and a response is sent, but a cancel request is made, then the pending MPI_Ssend_init/MPI_Start request should be stopped using MPI_Cancel. (To avoid race conditions, all cancels should actually be acknowledged by sending an arbitrary time. Cancelling then becomes a request-reply operation to provide necessary synchronization.)

Hence, all Occam programs can be converted into MPI-2 programs. Although the timers require busy waiting, this problem can be alleviated once MPI supports timeouts (a feature that will most likely be included in MPI-2).

6. Extending MPI Requests

As we saw in Section 3, Occam code has a restrictive format, making it difficult to extend the language to support first-class events. On the other hand, MPI already has a limited form of first-class events (called requests). They are limited since there are no first-class wrap, guard, and choose combinators. In addition, requests cannot be activated multiple times via MPI_Start.

MPI is a popular and powerful system. It provides a variety of features, such as collective communication, which are not available in CML. For this reason, we set out to remove the limitations of MPI communication requests. To do this, we propose additions to the MPI standard which provide the most important features of CML, namely, full-fledged events. Appendix A lists the described additions in the format of the MPI 1.1 document [3].

The first routine is MPI_Dup_req, which makes a replicate of the given communication request, so that both the original and duplicate can be MPI_Started. This would allow requests to be used in several places, by copying before MPI_Starting, thereby making requests completely first-class. Note that whenever a request is used in creating a high-level construct, the request is automatically copied for internal use.

Next we define a function MPI_Choice_init, which is similar to MPI_Waitany except that it is first-class and allows only one operation to complete, similar to CML's choose. That is, it creates a request representing a nondeterministic choice of the given list of inactive requests. If any active requests are given, an error is generated. MPI_Starting one of these requests corresponds to starting all of the subrequests, except that once one completes, the others are cancelled. At most one can complete (this could be implemented, for example, using a protocol similar to that of Section 5). The status returned from MPI_Wait should be the status of the single subrequest that completed.

We will now define equivalents of CML wrap and guard, namely MPI_Wrap and MPI_Guard. Errors are generated whenever active requests are specified or returned. Since functions are not first-class in C or Fortran 77, we allow an extra user-specified parameter to be passed to the wrap or guard function. The value is copied into the generated request, and is used whenever the function is called.

MPI_Starting a wrapped request corresponds to starting the subrequest. Obtaining the status object (via MPI_Wait) will call the associated function (which takes a status object and returns another).

MPI_Starting a guarded request corresponds to evaluating the function (which must return a request), and starting the returned request. MPI_Wait simply returns the status object obtained when MPI_Wait is applied to the previously returned request.

As the final request constructor, we define MPI_Timeout. This will finish once the specified amount of time (which must be positive) elapses (from when it was MPI_Started).

Finally, we must discuss what a "status object" represents. In MPI 1.1, it is a record with two fields, MPI_TAG and MPI_SOURCE, which contain the message's tag (a value which must be matched on receipt) and the source process's id, respectively (assuming that the operation was a receive). To provide more flexibility on the contents of this structure, we propose that a new field, `MPI_USER_INFO`, be added, which is simply a pointer. This allows the user to add data to the status object by wrapping; for example, a flag could be set to reflect whether the executed communication was a send, receive, or timeout. (Since status objects are not opaque, the `MPI_USER_INFO` field can be modified before a status object is returned by a function which is wrapped around a request.)

7. Conclusion

In this paper we have compared three concurrent programming languages, CML (Concurrent ML), Occam, and MPI (Message Passing Interface). CML has great flexibility because of its first-class events. Occam, although useful, provides no advantages over CML or MPI; in fact, Occam programs can be translated into CML or MPI programs. MPI has a large number of message-passing facilities, such as communicators and collective communication, and has the ability to be a competing concurrent language.

MPI has a restricted form of events called communication requests. These are (almost) first-class, but only allow basic sends and receives. We saw how requests can be extended to support many of the first-class events supported in CML, including the `choose`, `wrap`, and `guard` combinators. The addition of these features to the MPI standard would make MPI a more powerful concurrent language.

Looking at the other features of CML as possibilities for the MPI standard is one option for future work. For example, CML's synchronous I/O facilities may provide a good base for parallel I/O support (which is not yet provided by MPI).

A. Proposed Extensions to MPI

MPI_DUP_REQ(request, newrequest)

IN	request	communication request (handle)
OUT	newrequest	duplicate communication request (handle)

MPI_WAITONE_INIT(count, array_of_requests, request)

IN	count	list length (integer)
IN	array_of_requests	array of communication requests (array of handles)
OUT	request	communication request (handle)

MPI_WRAP(request, function, newrequest, datatype, parm)

IN	request	communication request (handle)
IN	function	user-defined function (function)
IN	datatype	type of parm (handle)
IN	parm	second parameter for function (choice)
OUT	newrequest	wrapped communication request (handle)

MPI_GUARD(function, datatype, parm, request)

IN	function	user-defined function (function)
IN	datatype	type of parm (handle)
IN	parm	parameter for function (choice)
OUT	request	guarded communication request (handle)

MPI_TIMEOUT(seconds, request)

IN	seconds	time to pause (double precision floating point)
OUT	request	timeout communication request (handle)

Acknowledgement

This work was supported by the Natural Science and Engineering Research Council (NSERC).

References

[1] A. W. Appel and D. B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 301–324. Springer-Verlag, September 1987.

[2] A. P. Black, N. C. Hutchinson, B. C. McCord, and R. K. Raj. EPL programmer's guide. Technical report, University of Washington, Seattle, WA, June 1984.

[3] M. P. I. Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994. For more information on MPI, see `http://www.mcs.anl.gov/mpi/`.

[4] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI.* MIT Press, Cambridge, MA, 1994.

[5] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[6] S.-T. M. Limited. *occam 2 Reference Manual.* Prentice Hall International Ltd., 1988.

[7] C. Lin and L. Synder. A comparison of programming models for shared memory multiprocessors. In *1990 International Conference on Parallel Processing*, volume 2, pages 163–170, 1990.

[8] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML.* MIT Press, Cambridge, MA, 1991.

[9] J. H. Reppy. Concurrent programming with events — The concurrent ML manual. Technical report, Dept. of Computer Science, Cornell University, Ithaca, NY, November 1990.

[10] J. H. Reppy. *Higher-order concurrency*. PhD thesis, Dept. of Computer Science, Cornell University, June 1992.