Higher-Order Concurrency in PVM*

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

Abstract

Message-passing systems are typically used to achieve high-performance parallel computing. In this paper we examine how advanced concurrent programming can be achieved using existing message-passing systems. In particular, we look at Reppy's extension to Hoare's basic CSP model, called *higher-order concurrency*, where communication events (such as send, receive and non-deterministic choice) 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. This allows the construction of high-level concurrent features without explicit support for them in the model itself. We see how first-class events can be implemented in PVM, showing that it forms the base of a powerful concurrent-programming language. We also provide garbage-collection of processes, which simplifies selective communication and dealing with end-cases. The implementation promotes more widespread use of the flexibility present in the higher-order-concurrency model, and allows multiple processors to be exploited in such programs.

1 Introduction

Substantial interest in *concurrent-programming languages* has recently developed. This has resulted in a variety of systems, such as Concurrent ML (based on higher-order concurrency) and occam (based on CSP). In this paper we focus on higher-order concurrency [13], an extension of Hoare's basic CSP model [10, 11].

Briefly, CSP provides message passing by basic send and receive primitives. They are synchronous in that the send/receive operations wait for matching receive/sends to continue. The destination or source passed to one of these primitives is specified by a *channel*. A

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

channel is a uni-directional connection between two processes. They effectively correspond to the *port* abstraction, that is, one of several message queues on a particular process, but are much more convenient to program with. The non-deterministic choice operation does one out of a list of sends and receives, whichever has a matching partner first.

Reppy [13] proposed an extension to the basic CSP model called *higher-order concurrency*. One way to view this extension is that synchronous-communication operations, which we call *events*, can be more than just a send, a receive, or a choice of sends and receives. Rather, an event could consist of doing several subevents in sequence. The idea is that you can then build up arbitrarily complex events by combining primitive events (sends and receives) with both non-deterministic choice and sequences.

1.1 First-Class Events

In the higher-order concurrency model, events are first-class objects, that is, events can be dynamically created, stored in variables, and passed to/returned from functions, as if they were more conventional objects like integers or booleans. Instead of writing a function to perform some communication (e.g., implement a protocol), one can write a function that returns an event describing how to do this communication. Events can be executed when desired via the `sync` operation. The advantage of having an "intermediate form" (events) is that the event returned by the function can now be manipulated (i.e., combined with other events) before it is applied. If the function carried out the communication directly, there is no way in general to manipulate it without changing the code.

A simple example is communicating with a distributed database that may replicate data (for example, DNS servers). A program may wish to query a number of database servers for the same information, and use the result that comes first (this corresponds to a non-deterministic choice). Presumably, the distributed database comes with a client function that allows a program to make a query to a server and get a response. If the function returns an event, we can simply call the function once for each server we want to query, use the `choose` combinator[1] (making another event `e`), and call `sync(e)`. On the other hand, if the function sends the request and then waits for a reply (instead of immediately returning an event explaining how to do so), then we would have to modify the routine to support several transactions at once, resulting in messy, difficult-to-understand/debug code, especially for complex protocols.

One may question whether complex protocols can actually be described using a small number of event combinators. Indeed, the higher-order concurrency model provides very few features, but they are provided in an extremely flexible (first-class) way. In his PhD thesis [13], Reppy showed that the following forms of communication could be constructed, using just synchronous send and receive events and various event combinators (of course,

[1] A *combinator*, from functional-language terminology, is a function that takes values of a particular type and "combines" them into a new value of that type. For example, the compose operation ($\circ$) is a combinator on functions.

the solutions also involve creating threads). In other words, he was able to construct event values *from scratch* that represent the following advanced concurrent features:

- Buffered channels (asynchronous communication)
- Multicast channels and thus multicast operations
- Condition variables (write-once variables)
- Ada-style rendezvous, plus allowing nested transactions
- Locks and semaphores
- Multilisp futures

Since these constructs are abstracted into events, they can be further manipulated. For example, we can non-deterministically choose between multiple Ada entry calls. We could also obtain lock A or lock B, whichever is available first; this is particularly useful for resolving deadlock. This is clearly not possible with other concurrent languages unless there is built-in support for these specific operations, but it is a fortunate side effect of higher-order concurrency.

With typical concurrent-programming languages, the power of the system can be measured by how many features it provides. On the other hand, if a concurrent-programming language provides first-class event abstraction, it does not have to "guess" what concurrency structures the user might want. If the language does not support a needed construct, users can build it themselves. Users do not have to make do with the provided forms of communication; instead, they can build up their own forms from existing ones, and use them as if they were built-in. Assuming they do this in the spirit of higher-order concurrency, they can then make new constructs from others that they have built.

One other important feature of higher-order concurrency is garbage collection of threads. The system monitors channels, and marks them as *dead* if there are no threads currently "at the other end" of the channel, that is, communications on this channel will necessarily block forever. Whenever a thread executes an event that only involves dead channels, it can be proven that the thread will block forever, and hence it can be discarded. This greatly simplifies non-deterministic communication and end cases; for example, the programmer does not need to worry about sending "kill" messages to so-called "daemon" threads that implement the above constructs.

1.2 Goals

Reppy implemented the higher-order-concurrency model in the Concurrent ML (CML) system [12, 14]. CML is implemented in the sequential language Standard ML (SML) via user-level threads. There are two major disadvantages to implementing higher-order concurrency in this way.

First of all, SML is not in common use except for programming-language research. This has the consequence that CML is not wide-spread, even though it has an incredible amount of flexibility that makes it a top-ranking concurrent language. It would be advantageous to bring the concurrency features to a more popular language such as C.

Second, there are no possibilities for exploiting a parallel computer, even though CML programs often have a high level of concurrency. If we can use system-level threads or operating-system processes, then we can put parts of the program on separate processors. SML has little support for this, whereas most of the development in this area has been done in C.

Although the ideas in higher-order concurrency look potentially difficult to implement, we show that with a proper protocol we can implement them efficiently on top of any message-passing system with only a few crucial features. In particular, we describe an efficient implementation on top of PVM (Parallel Virtual Machine) [8] using a base language of C. This proves that existing message-passing systems are powerful enough to build high-level concurrent-programming languages. In addition, the PVM implementation uses the concept of *tasks*, which are typically UNIX processes. Hence, programs can use several processors in an arbitrary heterogeneous system.

The implementation attempts to bring higher-order concurrency to a larger community. In this way, we hope that more people will realize the useful generality of the approach, and that the ideas will become more widespread. We feel that garbage collection of processes and first-class events and channels offer a powerful and easy-to-use abstraction for concurrent programming that does not significantly harm performance. Results discussed in Section 3.7 confirm this.

1.3 Related Work

As far as we know, no other work has been done on implementing higher-order concurrency in a language other than ML. Hence, the ideas of using parallel computers and bringing higher-order concurrency to a wider audience are entirely new. The advantage of first-class events, namely the ability to create new abstract forms of communication, has remained mainly hidden.

Another new idea is using message-passing systems as bases for concurrent-programming languages. This too has received little attention; typically, entirely new runtime systems are created. A few projects wisely choose to use existing message-passing systems, increasing portability and removing the extra time to reinvent the wheel.

Glenda [15] is a C-Linda implementation on top of PVM. Linda employs a partially shared-memory view, and so is entirely different from our work. As a result, Glenda offers a fair amount of overhead, whereas our implementation is more lightweight, especially in the worst case.

Compositional C++ (CC++) [4] and *Fortran-M* [5] are two interrelated projects both headed by Foster. They are designed to be inter-operable and are both based on top of the message-passing system *Nexus* [7]. CC++ is based completely on shared-memory concurrency (there are no message-passing primitives, for example), making it different from our work. Fortran-M uses typed channels, which is probably the only similarity to the higher-order concurrency model; there is no event type, channels cannot be passed along channels, and processes are not garbage-collected.

1.4 Outline

The rest of the paper is organized as follows. Section 2 briefly describes the reason we chose PVM, and notes the difficulty with using MPI. In Section 3, we show how higher-order concurrency can be implemented on top of PVM. We conclude in Section 4.

2 Choosing a Message-Passing System

The two most popular message-passing languages are *PVM* (Parallel Virtual Machine) [8] and *MPI* (Message Passing Interface) [9]. We now examine the two and show why we chose PVM.

PVM is a message-passing system designed for heterogeneous computing, based on C or Fortran. It supports many workstations and parallel computers. One of its useful features is that the entire system is presented in a uniform view. Whether we are using a completely concurrent system (that is, one processor) or a mix of parallelism and concurrency, we see a collection of processes that can communicate with each other.

In the next section we design a system built on top of PVM that provides several of the features in the higher-order-concurrency model. In fact, we do not need many facilities of PVM. We shall use the following features, most of which are typically available in message-passing systems:

1. Buffered send (`pvm_send`): send data to another process in the background, and upon arrival it waits in a buffer.

2. Blocking receive (`pvm_recv`): wait until data arrives and receive it.

3. Dynamic process spawning (`pvm_spawn`): create a new process (with separate code) that can communicate with any other process.

4. Explicit buffers: at the user's request, buffer ids can be obtained and created. For example, one could receive a message of unknown type or form and send it to some other process. This allows a sort of type-unchecked polymorphism.

We assume that each process has a unique integral id, and this is all that is needed to communicate with that process. Hence, destinations can be passed via messages. Furthermore, we assume that the application starts with a single process.

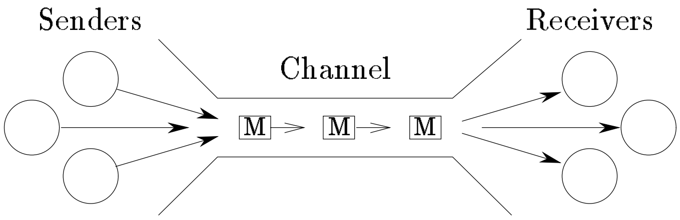

Figure 1: *A many-to-many channel. Attempting to send on a channel will block until one of the receivers attempts to receive a message from the channel, and vice versa. When there are multiple willing senders and/or receivers, the behavior is non-deterministic, although we draw it as a queue here.*

Unfortunately, we cannot use MPI in its current form. The various versions of MPI-1 (e.g., MPI 1.1 [3] and the almost-final MPI 1.2) do not support process spawning. The current draft of MPI-2 describes dynamic process spawning in the following way: when a process spawns a new one, an *intercommunicator* is defined. For the spawn operation to be local to the parent (that is, other processes do not have to contribute), the newly spawned process can only talk to its parent. In other words, destinations cannot be passed via messages as with PVM.

Foster, Kesselman, and Snir [6] have proposed a generalization of MPI communicators which are essentially ports. An important point is that references to ports can be passed to other ports, in other words, destinations can be passed via messages. Assuming this is accepted into the MPI-2 standard, this would allow an implementation for MPI similar to the one presented below. Without it, however, it is impossible to implement channels.

3 Implementing Higher-Order Concurrency in PVM

We now describe an implementation of first-class events and process garbage-collection on top of PVM. It promotes more wide-spread use of these powerful and flexible concurrency features. In addition, it allows multiple processors to be exploited.

3.1 Channels

The first construct we want to implement is channels. The higher-order-concurrency model allows *many-to-many* channels, that is, those with an arbitrary number of senders and receivers (Figure 1). However, such channels are difficult to implement in a distributed system, and cannot be implemented efficiently. For this reason, we will assume that channels are either one-to-many or many-to-one. From these we can build many-to-many channels by adding another process in between a many-to-one and a one-to-many channel. Using first-class events, these channels can be viewed as if they were built-in.

It is easier to implement one-to-many and many-to-one channels because they each have a unique sender or receiver, called the *owner*. The other processes connected to the channel are called *members*. Hence, a channel consists of the owner process id along with a unique identifier, so that data communicated on one channel is not mixed up with data on another channel. In practice, the identifier does not need to be globally unique. Instead, members can store two identifiers for each channel they belong to, unique to the member and the owner, respectively.

Channels will look different in the PVM implementation, because the language (C) does not allow shared-address spaces like CML. In other words, we cannot create channels in a context shared by multiple processes. Instead, we must define primitive operations to create and manipulate channels.

Initially, there is only one process and hence no channels. For each spawned process, we automatically create parent-to-child and child-to-parent channels. This is the only special construction, which provides the initial channels that we can build from. Any process can create a channel (either many-to-one or one-to-many) that it owns. Finally, processes can communicate channels by sending them along other channels. Receiving a channel automatically makes you a member[2].

This method is as general as possible. For example, we can create a channel from parent and grandparent to child as follows: the child creates a many-to-one channel with itself as the receiver, and sends it across the child-to-parent channel. The parent is then connected to the child; the descriptor which it receives tells how to send a message to the child via the just-created channel. Note that the owner of the channel (the child in this case) is not the only one who can send the channel to other processes. The parent can now send what it received from the child to the grandparent (which it is directly connected to). The grandparent can then communicate directly to the child.

Another concern is the deletion of channels. In CML, this is done automatically when the processes leave the scope in which the channel is declared. No such option is available in C. Instead, we must create a "delete" operation which discards channels. For channel members, we notify the owner about the deletion; we also tell the owner about any new processes that we send the channel to, so that it can keep track if there are any members left. For channel owners, we must only conceptually delete the channel and keep it around until every member discards its piece. This way, we can tell members that the owner is no longer accepting communications whenever they try.

On the member side, if a "no longer accepting" signal is sent, or if a send fails because of a PvmNoTask error, then the channel will be marked "dead." Communicating on such a channel will never complete. Similarly, the owner considers a channel to be dead during times when the number of members for the channel is zero. If a process `syncs` on an event whose channels are all dead, then that process is "garbage collected" as in CML, that is,

[2]This is even the case if the receiver is the owner of the channel. This way there is never more than one "copy" of the owner's side of the channel, which would cause obvious problems (the owner is no longer "unique").

`pvm_exit` and `exit` are called.

We describe in more detail how the membership is maintained in Appendix A.

3.2 First-Class Events

Next we show how to implement first-class events, using the spawn and channel operations described in the previous section.

Let us first describe the event combinators of interest. As we have indicated, `choose` takes a list of events and returns a new event representing the non-deterministic choice between them. `wrap` takes an event e and a function f with one argument, and returns a new event that is exactly like e, except that (if it completes) its result is f applied to the result of e. `wrapAbort` takes an event e and a function f with no arguments, and returns a new event that is exactly like e, except that if it does not complete, f is called. f should not contain blocking operations, or deadlock may occur; normally, it spawns a process to do the work. Finally, `guard` takes a function f with no arguments and returns a new event which, when `sync`'d, acts like the event returned by f, for the duration of that `sync` operation; it is effectively a delayed evaluation of an event. Together, `wrap`, `wrapAbort`, and `guard` can be used to build arbitrary *sequence* events.

Basically, an event is a tree, the root of which describes the highest-level event. For example, a `choose` node would have n children, where n is the length of the list.

The functions passed to `wrap`, `wrapAbort`, and `guard` take an extra `void *` parameter, which is given to the three combinators. This is done since functions in C (unlike SML) are not first-class, and do not have their own context. Conceptually, the extra parameter is a context; its actual meaning is anything conceived by the user. In addition, a deleter function is given to the three functions, so that the extra parameter can be deallocated when the corresponding event is.

We would like to allow the construction of a buffered-channel process, for example, where the type of the data being sent is unknown. To do this, we define both the return value of a `receive` event and the second parameter to the `transmit` function to be PVM buffers. `pack (x, f)` returns a buffer suitable for sending, where `x` is a pointer to some data, and `f` is a function that takes this pointer and packs it into the default PVM buffer using other pack functions. The code for `pack` looks roughly like: create a new PVM buffer b (`pvm_initsend`), set it to be the default buffer (automatic), call `f(x)`, and return b. Similarly, there is a function `unpack (buf, x, f)` which unpacks data stored in a buffer `buf` and places it in `x`, using a type-specific function `f`. The package provides packing and unpacking functions for simple C datatypes and channels.

The most difficult part in the implementation is the `sync` operation. In this paper we will only discuss the implementation for many-to-one channels, where members and owners correspond to senders and receivers, respectively. The one-to-many version is almost identical.

First of all, guard functions are evaluated, and (at least conceptually) they are replaced with the returned event. We then take note of any `transmit` and `receive` events in the tree. The basic idea is that (a) at least one of these events must complete before the `sync` operation completes, and (b) more importantly, once one of these events completes, the rest of the processing of the entire event is determined. Part (b) essentially says that the completion of a `transmit` or `receive` commits the entire path from that node to the root; in other words, this `sync` operation will cause no other `transmit` or `receive` operations to complete.

The protocol we adopt is sender-initiated. For each `transmit` event in the tree, we send a control message to the destination of the channel stating that we may send data along that channel if there is a matching `receive`. If there is such a `receive`, a reply will be sent to say that the destination is interested; this can only be done for one message at a time. Assuming that the sender did not commit to doing something else, it will then send a final commit message, and send the data itself. If the sender does commit to something, all other sends will be aborted (via control messages).

3.3 Detailed Protocol

There are four operations, `possible`, `complete`, `fullyComplete`, and `abort`, which manipulate the internals of events. They are used throughout the implementation of the `sync` operation. We summarize first, showing their use; explicit details can be found in Appendix B.

`possible (e)` marks `e` as a "possible" event, and recursively calls itself on any subevents. This is called with the event being `sync`'d as initialization. One crucial point is that for `transmit` operations, a "data available" message is sent to the channel's owner. `complete (e)` does the actual sends and receives when `e` was created by `transmit` or `receive`, calls wrapped functions, etc. `fullyComplete (e)` does this, and recursively does it for the parent event when the entire event is viewed as a tree. Finally, `abort (e)` aborts an event, as well as recursively aborting any subevents.

Once `possible` is called, the execution of `sync (e)` consists mainly of a loop. Each iteration of the loop begins with checking if any other processes have asked for data via data requests; if this is the case and we aren't committed to any data, then we send a "commit send" message, `fullyComplete` the corresponding send event, and leave the loop. This is the first end-case. Next, if there is a channel that we are interested in, and we aren't committed to any data, then we send a "request data" message, and commit to this data. Finally, we block to receive a control message, and then interpret it as follows (see Figure 2).

For "request data" messages, where a remote process asks for data of ours, we check if the requested data is that which we are *currently* trying to send along that channel. If not, we can ignore the message, since we have already sent an "abort send" message. If so, we add the request to the list of data requests. Assuming we aren't already committed to receiving some data, we will catch this addition in the next iteration of the loop, in which case a "commit send" message is sent. If we eventually leave the loop and the data-request list is non-empty, we will send "abort send" messages to the requesting processes, and delete

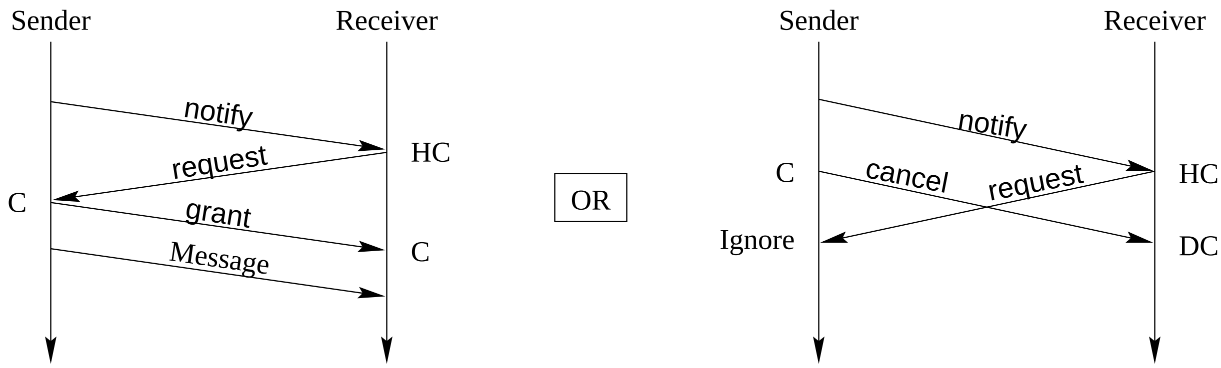

Figure 2: *The successful and unsuccessful scenarios in the described protocol.* notify*,* request*, and* grant *denote "data available," "request data," and "commit send" messages.*

the list.

"Commit send" messages state that the sender has committed to sending the data that we are committed to trying to receive. In this case, we remove the corresponding message information from the channel's queue, we `fullyComplete` the corresponding receive event, and leave the loop. This is the second and final end-case.

Alternatively, we may receive an "abort send" message from a sender, which implies that the send was aborted and so the data is no longer a possibility. If we are currently committed to receive that data, we break the commitment (in the next iteration of the loop, we may take up a new commitment). In any case, we remove a description of the data from the channel's queue.

Finally, there is a "data available" message, which is sent in the `possible` phase to tell of potential sends. Here we simply add the data description to the tail of the queue for the channel. This may be picked up in the next iteration of the loop if that particular channel is currently of interest.

3.4 Possible Improvements

In the previous section we described a protocol for implementing the features of higher-order concurrency in a message-passing system such as PVM, without changing any of the internals. The basic point is that it can be done, without any busy waiting or other serious inefficiencies. We now look at improving the protocol to reduce the number of messages required. In certain cases, we can modify the negotiation process to use an optimal number of messages.

One optimization we can make in general. Instead of sending or receiving actual data as separate phases (in `complete`), we can put the data in the "commit send" message. This reduces the number of messages by one, so that three messages are required in a typical case to negotiate a synchronous message-passing, which is close to the optimal value of two.

Unfortunately, this is not possible (in general) with one-to-many channels.

Another optimization takes place when there is only one send and no receive operations in the tree. In this case, the process is committed to doing the send. The receiver can detect this if the process provides a flag in the "data available" control messages. In fact, the "commit send" message is no longer needed; instead, the data is embedded into the "data available" message. Hence, only two messages (the optimal number) are required in this case.

We have recently developed other protocols that use the optimal number of message cycles per synchronous communication. We have also developed deadlock-free versions; the protocol described here is subject to deadlock for cyclic communication patterns (which are uncommon in concurrent programs).

A completely different form of efficiency is the overhead of multitasking. *PVML* [16] is a rewrite of PVM to support lightweight processes (threads). Unfortunately, it enforces that only one UNIX process can be used. At the time of implementation, this was the only available option, and we were forced to choose between lightweight concurrency (PVML) or exploiting multiple processors (PVM). *TPVM* [2] supports both in a transparent way; in the future, we hope to exploit this.

3.5 Multiple References

Suppose that the user creates a choose event e out of events e_1 and e_2, and then discards e_1 and e_2. e depends on e_1 and e_2, but the user should still be able to use e until s/he discards it. One way to implement this is to have the `choose` combinator duplicate e_1 and e_2, and use the copies with e. When e is deallocated, so are the copies. This will be inefficient, since dynamic allocation is slow.

To solve this, we created a general *multiple references* package. The idea is to maintain the number of references to each object, and only truly delete the object when this number becomes zero. The number of references to an event e is incremented each time a new event is created with e as a subevent. Similarly, it is decremented each time an event including e as a subevent is truly deleted, or when the user discards e.

Hence, the amount of dynamic allocation is minimized. We use this idea for events, channels, and buffers.

We implement this idea by maintaining a hash table whose (key, value) pairs correspond to (pointer, number of references) pairs. (Note that this works because we never use pointers that refer to the middle of an object.) The table is maintained within the package so that it is transparent to the user. However, we expect that this facility has other applications, providing a simple, efficient, explicit form of garbage collection for desired objects (dynamically allocated structures).

3.6 C++

C++ templates and function overloading simplify the described interface somewhat — these advantages are discussed below. We cannot, however, achieve type-checked message-passing (without preprocessing the source files). The main reason for this is that programs (executable files) define processes, whereas in CML functions define processes. Using threads only, it may be possible to achieve type-checking (as well as more easily-defined channels, etc.), but this removes the possibility of using a distributed-memory parallel system, which is one of the main foci of this work.

With C++ destructors, we can automatically delete channels and events. This slightly decreases the code size, but doesn't affect development time much. It is also likely that braces would be required to properly place the destructors; thus, explicit destructors would still be implicitly in the code, and the programmer would have to think about them.

We can overload functions `genpack` and `genunpack` to describe how to pack and unpack various datatypes, respectively. This way we wouldn't need a function parameter to the `pack` and `unpack` routines. Instead, they would be templated functions.

We can also overload a `discard` function, which can be used as a default for the extra parameter in the `wrap`, `wrapAbort`, and `guard` events. It would still be useful, however, to allow specifying a particular discard function in the call (this can be done with optional arguments). For example, pointers may be allocated with `malloc` or `new`, which would affect the choice of a discarding function.

Overall, we find that C++ does not simplify the interface significantly. Thus, we currently only provide a C interface. We have also developed a Java version of the package [1].

3.7 Results

We shall now present some results of our implementation on a single (IBM RISC/6000) processor. Basically, we want to evaluate the cost of event abstraction. To do this, we implemented four versions of a simple ping-pong benchmark to evaluate the time to send a message.

- Abstract: during each cycle, two events are created and `sync`'d.
- Partial: the same events are `sync`'d each cycle.
- Structured: the event structure is exploited to avoid all dynamic allocation.
- PVM: non-event-oriented asynchronous version.

In the best case, we would expect the first three benchmarks to run twice as slow as the PVM benchmark, since the former involve synchronous communication.

	Indirect		Direct	
Benchmark	Abs.	Rel.	Abs.	Rel.
Abstract	8.4029	4.7207	6.2711	3.6624
Partial	8.2087	4.6116	6.1405	3.5861
Structured	7.8974	4.4367	5.9473	3.4733
PVM	1.7800	1.0000	1.7123	1.0000

Table 1: *Absolute and relative timings for the four ping-pong benchmarks. Absolute times (abs.) are measured in milliseconds. Relative times (rel.) are the absolute times divided by the PVM benchmark.*

We ran each test several times using both indirect (process-daemon-process) and direct (process-process) routing via PVM, which uses TCP/IP communication. The results are given in Table 1.

Currently, the protocol is implemented so that four messages are required for each synchronous communication. Hence, the relative time of 4.72 for indirect routing is reasonable. Nearly half overhead is caused by dynamic allocation, as illustrated by the structured benchmark of 4.44.

We observe a more complex behavior for direct routing, because of the organization of the protocol. The second message is sent from the destination to the source, and the other three are sent from the source to the destination. Hence, the last two could be coalesced. This is somehow done automatically when there is a direct TCP/IP connection between the communicating processes. This explains the difference of about one between the direct- and indirect-routing relative times.

We conclude that event abstraction is cheap relative to PVM's process-to-process communication performance. It would be worthwhile to test how much this changes if we look at PVML's or TPVM's thread-to-thread communication performance, which should be much better.

4 Conclusion

This paper has presented an implementation of many higher-order-concurrency ideas in PVM. In particular, we achieved our main goal of providing a concurrent-programming language with first-class events. This allows one to build up complex concurrency constructs from the most basic operations in a flexible way, a powerful feature that is not yet well known or commonly available. The implementation also supports garbage collection of processes, greatly simplifying selective communication and end-cases.

There are several advantages to our implementation which come from the use of PVM as a base. First, it is portable. Second, many languages can be potentially used; with conventional computing environments, C is the "universal glue" between languages. For example, Fortran code could be used for computation phases of programs. Last and most important is the possibility of exploiting multiple processors to obtain performance benefits with the program's concurrency.

Nevertheless, there is room for improvement. First, threads and processes cannot be supported simultaneously (a limitation of PVM). Second, our interface is somewhat tedious to use directly. For example, creating a new kind of process involves creating and compiling an additional file. A simple macro utility may significantly improve ease-of-use.

Acknowledgment

We wish to thank David Taylor and Thomas Kunz for valuable discussions on the work and comments on the paper.

References

[1] Erik D. Demaine. Higher-order concurrency in Java. In *Proceedings of the Parallel Programming and Java Conference (WoTUG20)*. IOS Press (Netherlands), April 1997.

[2] Adam Ferrari and V. S. Sunderam. TPVM: A threads-based interface and subsystem for PVM. Technical Report CSTR-940802, Dept. of Mathematics and Computer Science, Emory University, Atlanta, GA 30322, August 1994.

[3] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994.

[4] Ian Foster. Compositional C++. In *Designing and Building Parallel Programs*, chapter 5. Addison-Wesley Inc., 1995.

[5] Ian Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):21–35, 1995.

[6] Ian Foster, Carl Kesselman, and Marc Snir. Generalized communicators in the Message Passing Interface. In *Proceedings of the 2nd MPI Developers Conference*, pages 42–49, Notre Dame, Indiana, July 1996. IEEE Computer Society Press.

[7] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus task-parallel runtime system. In *Proceedings of the 1st International Workshop on Parallel Processing*, India, 1994.

[8] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, MA, 1994.

[9] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI.* MIT Press, Cambridge, MA, 1994.

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

[11] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[12] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, Toronto, June 1991.

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

[14] John H. Reppy. Concurrent ML: Design, application, and semantics. In *Programming Concurrency, Simulation and Automated Reasoning*, Lecture Notes in Computer Science, Berlin, 1993. Springer-Verlag.

[15] Benjamin R. Seyfarth, Jerry L. Bickham, and Mangaiarkarasi Arumughum. *Glenda Installation and Use.* University of Southern Mississippi, November 1993. World Wide Web. `ftp://seabass.st.usm.edu/pub/glenda.tar.Z`.

[16] Honbo Zhou and Al Geist. LPVM: A step towards multithread PVM, November 1995. World Wide Web. `http://www.epm.ornl.gov/~zhou/lpvm.ps`.

A Maintaining Membership

When a member discards its channel, it sends a "remove" message to the owner. However, "add" messages are not as simple.

Suppose that members only send "add" messages immediately upon receiving a new channel. Then the following scenario leads to an unfortunate situation: the sole member of the channel sends the channel to another process, and immediately deletes its copy. This will cause a "remove" message to be sent, which will likely arrive before the "add" message. Thus, for a period of time, the owner believes that there are no members. It is possible, then, that the owner decides to delete the channel entirely, even though there will be a member soon.

A similar race condition occurs if only the sender issues "add" messages if we allow arbitrary network delays[3]. Hence, both the sender and the receiver need to send "add"

[3]Note, though, that we will assume, for any pair of processes p and q, the messages from p to q are delivered reliably and in the order that they were sent. This corresponds exactly to the quality-of-service that PVM provides.

messages in such a way that the second-received one is ignored. This will remove any possible race conditions.

The owner cannot simply maintain a list of first-received "add" messages, and remove one from the list when it receives the second "add" message with the same source and destination. This would not work if a process sent a channel to another process twice in a row (simply suppose that the sender's "add" messages both arrived at the owner before the other "add" messages, and we can arrive at race conditions similar to those above).

To solve this, we split "add" messages into sending-side and receiving-side add messages, sent by the sender and receiver of a channel, respectively. Several sending-side (or receiving-side) add messages with the same source and destination may queue up in the list, and are individually removed by receiving-side (or sending-side) add messages with the same source and destination.

There is one final complication. "Add" messages are sent at the packing and unpacking phases. However, not all received messages are unpacked; they may just be treated as buffers and sent to other processes. We enforce that they are eventually unpacked. Once they are unpacked, however, the source and destination will not match with the original sending-side add message sent during packing.

To solve this, we store the original sender and receiver along with each buffer, and use this as the source and destination pair for the receiving-side add message. This is required in any case, because the arguments to `pack` and `unpack` do not include the destination and source processes.

B Event-Manipulation Operations

This appendix gives details on all four of the event-manipulation operations used in the PVM implementation of first-class events. These are `possible`, `complete`, `fullyComplete`, `abort`.

Given a send event, `possible` sends a "data available" message to the destination process, and sets a flag to say that this was done. A receive event causes the channel to added to the list of "interesting channels." `possible` is recursively called on the subevents of `wrap`, `wrapAbort`, and `choose` events. For a `guard` event, the function is evaluated (saving the result in the event structure), and `possible` is called on that result.

`complete`, given a `transmit` event, sends the actual message to the destination, clears the "sent data-available message" flag, and sets the overall result of the `sync` operation to nothing. With a `receive` event, we receive the actual message, and set the overall result to the received data. For a `wrap` event, the function is applied to the overall result, changing the overall result. A `choose` event causes all but the chosen subevent to be `abort`ed. Nothing happens for `guard` or `wrapAbort` events.

`fullyComplete (e)` calls `complete (e)`, and then checks the parent of `e` when the overall event (the one passed to `sync`) is viewed as a tree. If the parent exists, it is given to `fullyComplete`. If the parent is a `choose` event e', it is first noted which subevent is the current event. This is used when `complete` is called on e', where `abort` is called on all events except e.

Finally, `abort`, when given a `transmit` event, sends an "abort send" message if the "sent data-available message" flag is set. For `receive` events, we remove the channel from the list of interesting channels. `aborting` a `wrap` or `choose` event causes recursive `aborts` on the subevents. Similarly, with `guard` events, we recursively `abort` on the result of the function. Most importantly, to `abort` `wrapAbort` events, we recursively `abort` the subevent and then call the associated abort function.