This paper is a revision of Technical Report No. UCB/EECS-2009-7
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-7.html
January 18, 2009

Disciplined Message Passing *

Edward A. Lee
UC Berkeley
eal@eecs.berkeley.edu

May 24, 2009

Abstract

This paper addresses the question of whether message passing provides an adequate programming model to address current needs in programming multicore processors. It studies the pitfalls of message passing as a concurrency model, and argues that programmers need more structure than what is provided by today's popular message passing libraries. Collective operations and design patterns can help a great deal, but as the use of concurrency in programming increases, application programmers will have increasing difficulty identifying and combining these into complex operations. Moreover, some challenges, such as ensuring data determinacy and managing deadlock and buffer memory, are extremely subtle, and require considerable expertise to implement correctly. This paper illustrates this point by giving a few problematic examples. I argue that application programmers should not have to deal with many of these challenges, but with today's message passing libraries, they have no choice. The solution is to provide infrastructure-level support implementing more disciplined concurrent models of computation (MoCs). I show how process networks and dataflow models can provide excellent implementations of the requisite mechanisms, thus enabling application programmers to focus on the functionality of the application rather than on avoiding the pitfalls of concurrent programming.

1 Introduction

Today, a great deal of attention is going into concurrent programming models because of the rise of multicore machines. The question I address in this paper is whether message passing libraries, as currently designed, provide an adequate programming model to address this need. The conclusion is that although these libraries offer improvements over some of the alternatives, they have a long way to go to really solve the problems.

*This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

A program is said to be **concurrent** if different parts of the program *conceptually* execute simultaneously. A program is said to be **parallel** if different parts of the program *physically* execute simultaneously on distinct hardware (such as on multicore machines or on server farms). Non-concurrent programs can typically be executed in parallel (witness multi-issue instruction streams), and concurrent programs can be executed without parallelism. Thus, the two concepts, though related, are distinct.

Programs are typically executed in parallel to improve performance. Hence, one may develop a concurrent program to facilitate exploiting parallel hardware. However, there are other reasons for developing concurrent programs. One reason is I/O. Portions of a program may stall to wait for inputs from the environment, for example, or to wait for the right time to produce some output. Another reason is to create the illusion of parallel hardware. Time-sharing operating systems, for example, were originally created to give each user the illusion of having their own machine. Of course, these two reasons are intertwined, since interaction with such users is about I/O and its timing.

Techniques for developing concurrent programs divide into two families, **message passing** and **shared memory** [29]. The term "shared memory" is used in the literature to refer to both a hardware architecture technique and a programming model. In this paper, I am concerned only with the programming model, so in an attempt to avoid confusion, I will use the term "threads" to refer to the programming model. Threads are sequential procedures that share variables and data structures. They can be implemented on hardware that does not physically share memory (and conversely, message passing can be implemented on hardware that does physically share memory). Java, for example, includes threads in its programming model, and can be implemented on a variety of hardware.

Many programmers are familiar with threads and appreciate the ease with which they exploit parallel underlying hardware. It is possible, but not easy, to construct reliable and correct multithreaded programs. See for example Lea [30] for an excellent "how to" guide to using threads in Java. Recently, Java acquired an extensive library of concurrent data structures and mechanisms based on threads [31]. Libraries like OpenMP [10] provide support for commonly used multithreaded patterns such as parallel loop constructs.

Nonetheless, threads have a number of difficulties that make it questionable to expose them to programmers as a way to build concurrent programs [42, 46, 34, 23]. Nontrivial multithreaded programs are astonishingly difficult to understand, and can yield unexpected behaviors, nondeterminate behaviors, deadlock, and livelock. Problems can lurk in multithreaded programs through years of even intensive use of the programs. These problems do not yield easily to regression testing techniques because the number of interleavings that a thread scheduler can deliver is vast. They also do not yield easily to static analysis techniques because of decidability problems and other more pragmatic concerns.

In message passing, rather than sharing variables and data structures, sequential procedures invoke library mechanisms for sending messages to one another. Two popular such libraries are MPI [22, 40] and PVM [17].

There has been considerable debate about the merits of message passing schemes (or their cousins, event-based mechanisms) vs. threads. Some argue that message passing is a bad idea [5]. Some argue the contrary [42, 48, 50]. Gorlatch [20] argues

against the direct use of send-receive primitives in message-passing libraries, advocating instead the use of collective operations (like MPI's broadcast, gather, and scatter). He draws an analogy between send-receive and goto, invoking Dijkstra's famous indictment of goto [12]. Collective operations, he argues, are analogous to structured programming. These points are valid, as far as they go.

The collective operations of MPI can be viewed as infrastructure supporting design patterns. There are many more design patterns of interest, however, than those supported directly by MPI [39]. Hence, I would even go further, arguing that no fixed set of collective operations can be adequate. The examples I use below do not in fact fit the patterns of any of the MPI collective operations. What is needed is a notion of higher-order components [37], where patterns of send-receive structures are codified as combinators that can be reused in various scenarios. These higher-order components can be coded in executable pattern languages such as Ptalon [9]. However, even this does not solve many of the problems I identify in this paper. It provides only part of the solution.

Agha [2] makes a similar indictment against send-receive primitives, also drawing analogies with the infamous goto. His conclusions are more similar to mine, however, in that he advocates the use of "interaction patterns," which relate closely to what I call concurrent models of computation. He advocates a particular concurrent MoC that he calls "actors." In my research group, we have been using the term "actor-oriented design" for a broader class of message-passing concurrent MoCs than Agha's, but a class where many of the core principles are the same.

Some researchers equate message passing and event-based schemes (see for example [5]). The term "events" is generally used to refer to the kind of processing that goes on in GUI design. As defined by Ousterhout [42], event-driven programming has one execution stream (no "CPU concurrency"), registered callbacks, an event loop that waits for events and invokes handlers, and no preemption of handlers. However, message passing libraries like MPI are not (to my knowledge) used in this way. They are used in highly parallel settings for scientific computing. So while we could consider events a special case of message passing, we should consider them to be a rather particular special case, best suited for GUI design. In this sense, I agree with Adya et al., who say "the popular term 'event-driven' conflates several distinct concepts." [1].

In this paper, I consider a broad class of message-passing systems, where the key operating principle is concurrency without shared memory. For concurrent processes to communicate, they send messages rather than directly accessing shared data structures. GUI-style events have this character, but also much more specific structure (callbacks, handlers, no preemption) that are not intrinsic to message passing. My broad interpretation here is consistent with that of Lauer and Needham [29].

The debate between threads and message passing is quite old. Lauer and Needham in 1978 showed that "message-oriented systems" and "procedure-oriented systems" are duals [29]. "The principal conclusion is that neither model is inherently preferable, and the main consideration for choosing between them is the nature of the machine architecture upon which the system is being built, not the application which the system will ultimately support." This conclusion is less valid today, however, because machine architectures have evolved to support both quite efficiently. The duality of these models, however, remains important.

Although message passing may provide a better programmer's model than threads, it is far from perfect. I will show that message passing can be improved considerably using known techniques, by providing programmers with more structure (and more constraints) than what is found today in message-passing libraries. This can maintain the flavor of message passing libraries, and can be used like libraries with legacy programming languages. This contrasts with other solutions that require programmers to adopt new languages that support concurrency, such as Erlang.

Message passing suffers from three key difficulties that I will elaborate on. First, it lacks sufficient discipline to yield nontrivial programs that are determinate and/or easily understood. Programmers will be surprised by unexpected behavior. Of course, there are trivially simple applications of message passing that are easily understood, and more complicated patterns that have become well understood. But with the drive for ever greater concurrency in order to exploit multicore machines, fewer programmers will constrain themselves to these simple forms and well-understood patterns.

Second, message passing programs are difficult to make modular. The code for each process must be written with explicit knowledge of which processes will provide it with input data and which processes will use its output data, and with explicit knowledge of how those data that are provided are used (for example, the order in which messages are read, and whether the read is blocking). The code for a process cannot usually be reused in another scenario without rewriting. Some libraries, notably MPI with its notion of groups, have improved the modularity of such programs, but I will show that the improvements fall far short of truly modular design.

Third, message passing interacts in very subtle ways with scheduling, and message passing libraries give little or no control over scheduling. Programmers have to become quite expert in the subtleties, and then are often forced to play operating-system-specific tricks to avoid buffer overflows and deadlock.

These difficulties can all be overcome by using more disciplined programming models. By this, I mean that programs should be built to conform with a *concurrent model of computation* (MoC) that emphasizes understandable and predictable behavior (over flexibility). The problem with overly flexible models is that every programmer is forced to discover and circumvent the pitfalls, some of which can be extremely subtle. Consider for example the question of whether a program requires fairness, and if it does, how to achieve that fairness. Consider also determinacy. Some programs require nondeterminate behavior to achieve their objectives, but programmers do not want to be surprised by nondeterminism. Unfortunately, it is common for programmers to use intrinsically nondeterminate means to accomplish determinate ends. But is the result really determinate? How can you know? Testing will not answer that question. Consider also boundedness. How can a programmer of a message-passing system be sure that buffers will not overflow? Ensuring this by using a synchronous send is draconian, changes the semantics of programs, and often makes it very difficult to avoid deadlock. Requiring the programmer to set fixed buffer sizes is also asking quite a bit, since good choices (or even correct choices) may require holistic analysis of the application.

A well-chosen MoC (and its implementation) will provide programmers with easily understood and used mechanisms that will do the job most of the time. Let's give the programmers these mechanisms rather than giving them the tools to re-invent them. Unfortunately, what we mostly do today is the latter.

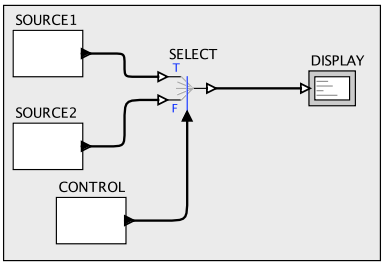

Figure 1: A simple five-process program, where messages from two SOURCE processes are merged based on messages from the CONTROL process and displayed by the DISPLAY process.

2 Motivating Example

In this section, I consider first a simple scenario that we can use to illustrate some of the weaknesses of design approaches based on message passing libraries. In this example, we use the popular library MPI [40], but similar arguments apply as well to other libraries that programmers use. I begin with a simple, small-scale version of the example, developing an MPI implementation and analyzing its effectiveness. I then show that the example represents a pattern with very real, practical applications, that these applications immediately scale up to large amounts of concurrency, and that the MPI solution that I constructed is not satisfactory. In subsequent sections, I will show that a satisfactory solution requires going beyond the built-in capabilities of MPI, and argue that in fact it requires mechanisms that would not naturally be included in a message passing library, treading instead on the turf of the operating system. The principles such a library appear to be at odds with finding a satisfactory solution.

2.1 The Simple Version

The scenario is illustrated graphically in figure 1. There are five processes, four of them depicted by boxes and one (SELECT) depicted by a suggestive icon. The two SOURCE processes send messages to the SELECT process using MPI statements like this one:

```
MPI_Send(&value, count, MPI_INT, SELECT, ...);
```

This statement sends an array of `count` elements in the variable `value`, each of which has type `MPI_INT`, to a process whose rank (ID) is `SELECT`. The remaining arguments are not important for our purposes here.

Using a similar MPI_Send statement, the CONTROL process sends a sequence of boolean values (or, in C, zero and non-zero), that tell the SELECT process whether it should receive data from SOURCE1 or SOURCE2. The SELECT process looks like this:

```
while (1) {
  MPI_Recv(&control, 1, MPI_INT, CONTROL, ...);
  if (control) {
    MPI_Recv(&selected, 1, MPI_INT, SOURCE1, ...);
  } else {
    MPI_Recv(&selected, 1, MPI_INT, SOURCE2, ...);
  }
  MPI_Send(&selected, 1, MPI_INT, DISPLAY, ...);
}
```

The DISPLAY process reads data sent by SELECT using a similar MPI_Recv statement.

Assume this program is to process a large amount of data. That is, the SOURCE and CONTROL processes provide an unbounded number of messages. It is not hard to imagine applications that fit this pattern. The pattern is easily elaborated, for example by replacing the DISPLAY process with a process that does data analysis. The pattern is simple: multiple sources of data are merged in some way for analysis.

There are a number of major problems with this program, as described. First, the MPI standard [40] does not give a definitive semantics to the MPI_Send procedure. The procedure is described as a "blocking send," which means simply that it does not return until the memory storing the value to be sent can be safely overwritten. In the above `MPI_Send` statement, that is the memory pointed to by `value`. The MPI standard allows implementations to either copy the data into a "system buffer" for later delivery to the receiver, or to rendezvous with the receiving process and return only after the receiver has begun receiving the data.

There are enormous practical and semantic differences between these two choices. Suppose that an MPI implementation uses a system buffer. Then how can we keep that system buffer from overflowing? In practice, if the two SOURCE processes and the CONTROL process send unbounded streams, the system buffer will rapidly overflow and the program will fail when it runs out of memory. Explicitly allocating buffers using `MPI_Buffer_attach` doesn't help because, first, MPI does not limit the buffering to the specified buffers, and second, the result is still an error when the buffers overflow. Yet it is easy to see by inspection that this program can be run with bounded buffers. We need somehow to exercise control over the process scheduling.

One way to exercise such control is to force MPI to use a rendezvous style of communication by using `MPI_Ssend` instead of `MPI_Send`. `MPI_Ssend` does not return until the receiving process has at least begun to receive the message (it has reached a matching `MPI_Recv` statement). However, this is semantically very different from the asynchronous message send. In MPI, we are forced to modify the process code based on how the outputs from the process are being used. This is poor programming practice. We cannot re-use the same process code for the SOURCE components in another setting where rendezvous is not desired. The resulting code is not modular.[1]

[1] Interestingly, one of the design goals of MPI was modularity, and it has been argued to be more modular than PVM [21]. MPI includes the notion of a process *group*, and sources and destinations are given as a rank within a group, rather than an absolute ID. This provides a partial form of modularity, in that groups of processes can be designed together and reused. In fact, this mechanism can help quite a bit when scaling

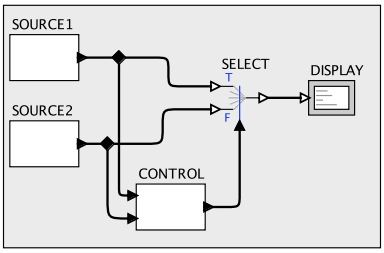

Figure 2: A variant of the program in figure 1 where the CONTROL process analyzes the data from the SOURCE processes before deciding how to merge them.

To underscore this point, consider the variant shown in figure 2. This figure has added one more visual notation, a black diamond, which means simply that the messages from the SOURCE processes should now go to both the SELECT and CONTROL processes. One way to accomplish this in MPI is to modify both SOURCE processes to issue two `MPI_Send` statements instead of one, as follows,

```
MPI_Send(&value, count, MPI_INT, SELECT, ...);
MPI_Send(&value, count, MPI_INT, CONTROL, ...);
```

This approach is unfortunate, of course, because the code for the SOURCE processes is being modified based on how the messages are used, which is not modular. An alternative would be to create two additional processes whose task it is to route copies of the messages to both destinations, but this will result in considerable overhead.

The CONTROL process might be written as shown in figure 3. Here, the process first reads once from each input channel. Depending on the values read from the input, it chooses to send a *true* or a *false* control message to the SELECT process (the `someCondition()` procedure makes the choice). This instructs the SELECT process to forward one of the two data values. Once the message is sent to the SELECT process, it then reads from whichever input stream it instructed the SELECT to forward data from. Thus, each time through the loop, it compares the two data values available to the SELECT process, chooses one, an instructs the SELECT to forward that one.

Note that as written now, if synchronous message passing is used (either because the MPI implementation realizes `MPI_Send` synchronously or because we use `MPI_Ssend` instead of `MPI_Send` in the code above), then the system will deadlock immediately. The two SOURCE processes will attempt to rendezvous with the SELECT, which is attempting to rendezvous with CONTROL, which is attempting to rendezvous with the SOURCE processes.

We can fix this by reversing the order of the two sends in the SOURCE processes,

---

up programs, as for example in moving from figure 2 to figure 4, discussed below. But within a group, each process must still specify in its source code which processes it receives from and which processes it sends to. The fact that this is specified in the source code for the process is what makes this non-modular. Moreover, changing any communication across groups also requires modifying source code in the process definitions. This is a rather weak form of modularity.

```
MPI_Recv(&data1, 1, MPI_INT, SOURCE1, ...);
MPI_Recv(&data2, 1, MPI_INT, SOURCE2, ...);
while (1) {
  if (someCondition(data1, data2)) {
    MPI_Send(&trueValue, 1, MPI_INT, SELECT, ...);
    MPI_Recv(&data1, 1, MPI_INT, SOURCE1, ...);
  } else {
    MPI_Send(&falseValue, 1, MPI_INT, SELECT, ...);
    MPI_Recv(&data2, 1, MPI_INT, SOURCE2, ...);
  }
}
```

Figure 3: The CONTROL process definition for figure 2.

```
MPI_Send(&value, count, MPI_INT, CONTROL, ...);
MPI_Send(&value, count, MPI_INT, SELECT, ...);
```

Again, we need to modify the process code depending on how it is used. Worse, the modification is very subtle and brittle. Any small change in the usage pattern, and the program will again fail with a deadlock. Reasoning about the cause of the deadlock quickly gets difficult.

No such deadlock would occur if asynchronous message passing were used, which is probably what we really want. But without some way to control process scheduling, we cannot prevent buffer overflow. And even if we were given a way to control process scheduling, what policy should we follow? Fair? Data-driven? Demand-driven? I will show that all three fail, leaving the programmer with quite a conundrum. Yet policies that work well exist. We should not require programmers to reinvent them for every program.

This example can, of course, be made to work. But the programmer is suddenly forced to do some rather sophisticated reasoning about the concurrency and memory management properties of the system. Yet this is a rather simple application. The result is code that is highly non-modular, where every process has to be customized for the particular way its input messages are provided and its output messages are used.

2.2 Discussion of the Example

Consider a practical problem. Suppose that we have a record of time-stamped stock or commodity transactions in markets around the world, sorted by time stamp. These would certainly be large data files. Suppose further that we wish to merge these in time-stamp order in order to perform analysis of global dynamics of the markets. If we had only two such files, from two markets, then the program in figure 2 would do the job. The CONTROL process in figure 3 will work if the `someCondition()` procedure returned `true` if the time stamp of `data1` is less than the time stamp of `data2`.

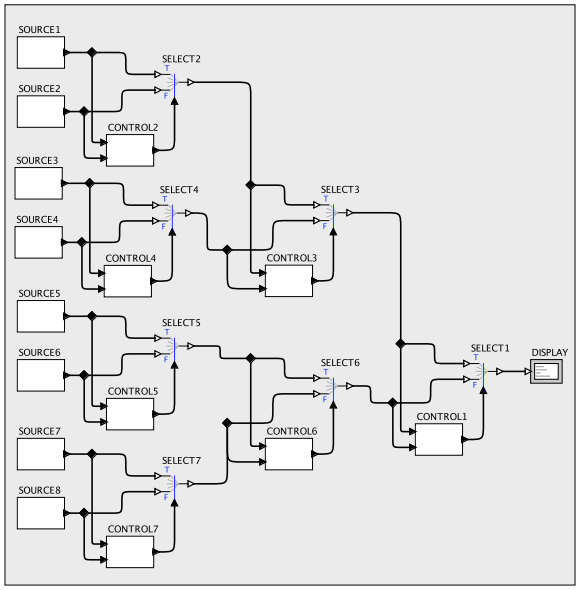

Figure 4: A variant of the program in figure 2 that merges 8 sources rather than two.

There are more than two markets in the world, however. We can easily scale up this application using a tree structure, as suggested in figure 4, which assumes eight markets. A full-scale application, of course, would be much larger.

If we choose the solution of using a synchronous send, then we will have effectively sacrificed most of the available concurrency in the model. This would not yield good performance on a multicore machine or server farm.

Von Behren, Condit, and Brewer argue in [5] that message-passing, which they equate with event-based programming, is a bad idea for high-concurrency servers. They argue instead for threads. This application seems to fit the high-concurrency server class, but it also seems to fit what they describe as "dynamic fan-in" patterns. Such patterns, they say, are the only ones they considered that are "less graceful with threads [than with message passing]." Thus, even advocates of threads over message passing would seem to agree that this example is well-suited to message passing. Yet message passing, at least as realized in MPI, does not handle it well.

3 Concurrent Models of Computation

A message passing library like MPI realizes a concurrent model of computation. The question is whether it is the right MoC. I will argue that it is not. We need more structure to grow complex programs in a concurrent and parallel world.

Concurrent MoCs that are more constrained can result in programs that are more easily understood. Consider an old, but rather elegant, example: Unix pipes (and their

extension, named pipes). Unix pipes provide a very simple mechanism for concurrently executing interacting programs. The mechanism couples memory management and process scheduling to yield provably bounded buffer memory and determinate composition. Given the same input data, every execution of programs communicating via pipes yields the same data results.

But Unix pipes are too constraining. The example in figure 1 cannot be realized with pipes alone because a process has only one standard input. A programmer can step down a level of abstraction and resort to sockets, but this, in effect, is just choosing to use a (rather old) message passing library. Sockets do not integrate process scheduling like pipes, and the programmer will face all the same problems we illustrated with MPI in the last section.

The question becomes simply: can we achieve the conceptual simplicity of pipes with the expressiveness of message passing? I claim that the answer is yes, and that in fact there are several well-understood mechanisms that could be provided to programmers. Each of them can be realized with message passing libraries, in principle, but doing so correctly would require a great deal of sophistication from programmers. More importantly, it is extremely inefficient to require programmers to reinvent these techniques every time they write a program. Few of them will get it right. Let's codify what we know and provide suitable programming frameworks.

In this section, I will describe a few disciplined concurrent MoCs, namely Kahn process networks (KPN), extended process networks (EPN), and dataflow models (specifically, synchronous dataflow or SDF). These vary in the expressiveness, determinacy, and understandability, but each provides much more structure than what is found in message-passing libraries. I will illustrate some of the (considerable) subtleties in implementing these well, in an effort to convince the reader definitively that we cannot continue to demand that programmers reinvent these techniques each time they write a program. There are several more concurrent MoCs that I will not discuss in this paper, including rendezvous-based models (CSP [25], Reo [3]), Synchronous/Reactive (SR) [6], Discrete Events (DE) [32], and time-driven models (e.g. Giotto [24]). These are interesting and potentially very useful, but the point of this paper is not to catalog the available disciplined concurrent MoCs, but rather to convince the reader that we need them.

3.1 Kahn Process Networks

Concurrent processes that communicate via streams of messages have been around for some time. Unix pipes, mentioned above, are a particularly constrained form of this pattern. A more general form with many of the same attractive properties is known as Kahn Process Networks (KPN) [26, 27, 37]. KPN provides an elegant solution to the problems considered above, and to many others. Moreover, KPN can be generalized to support nondeterminism in a disciplined way (as discussed below), and be specialized to yield to extensive static analysis and optimization (also discussed below).

In our realization of KPN in Ptolemy II [14], there is a level of indirection between the process code and the message passing infrastructure. In particular, the SOURCE processes of figure 1 are written using statements like this:

```
output.send(data);
```

where `output` is an *output port*. The notion of an output port permits us to write process code that makes no reference to the processes that will receive its messages, nor even to how many processes will receive its messages. The connections between processes are declared separately, so the code is much more modular. A similar level of indirection is used in nesC to improve modularity [15]. A process definition can be easily reused in multiple contexts. The code does not change based on how many recipients there are for the message, and it makes no reference to buffer sizes. Yet a correct implementation of KPN can guarantee that every process composition will execute with bounded buffers if there exists a bounded buffer execution.

Processes that receive messages have statements like

```
data = input.get();
```

where `input` is an input port. Again, there is no direct reference to the source of the messages. Following Kahn-MacQueen semantics [27], this call blocks until the required input data is available.

Using ports with get() and send() methods provides for much more modular programming than direct MPI calls. As a result, process definitions can be effectively reused in different scenarios. In fact, a current usage of KPN is to construct scientific workflows [38], where programs are constructed graphically as in figure 1 from mostly predefined components in a library. Each component defines a reusable process (these processes do data analysis, for example, or more interestingly, wrap database, grid, or web services).

The reader may object that this extra modularity must come at the cost of performance. Parks compares the performance of MPI with a Java implementation of KPN, and concludes that the overhead is modest with primitive data types [44]. For non-primitive data types, he attributes much of the performance penalty to serialization and deserialization of Java objects.

The semantics of KPN is elegant and simple. At its root, it assumes that buffers are unbounded. I will show below that this does not result in overflowing memory, as the above MPI example did. Semantically unbounded does not imply that the implementation must be unbounded, unless the program itself requires it.

As elaborated below, KPN programs are determinate. All executions yield the same sequence of messages on each connection between processes. It is very difficult to make any such assertion about an MPI program unless the programmer exercises enormous restraint. The MPI API is both too rich, providing tempting capabilities to deviate from the constraints, and too impoverished, providing little control over process scheduling. The latter problem of course can be fixed by enriching MPI. This is the standard approach in libraries when limitations are discovered. Enrich the API to provide more capability. But as I will show below, this leaves the programmer with the extremely challenging task of correctly implementing something that several Ph.D dissertations have been written about.

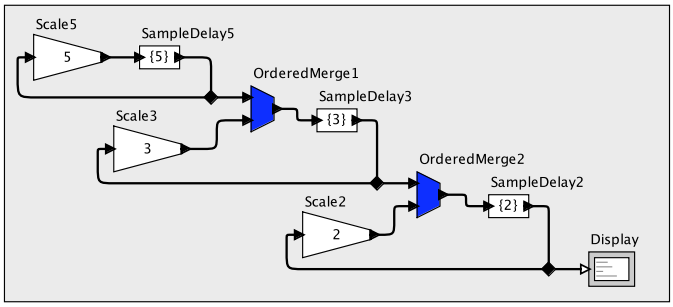

Figure 5: A message-passing model for which it is difficult to determine appropriate buffer sizes.

3.2 Buffer Management in KPN

Consider first buffer management. KPN semantics depend on unbounded buffers. But any reasonable implementation of either figures 1 or 2 would use a finite (and small) amount of memory for buffering messages. How can that be achieved with KPN semantics?

One answer is to require that the programmer specify buffer sizes for each connection between processes, and to implement the `send` method of output ports to block when buffers are full. This is done in SEDA, for example [49]. The problem here is that the programmer may have considerable difficulty determining appropriate buffer sizes. Buffers that are too small can result in an *artificial deadlock* (a deadlock that is due only to the bound on the buffers). Buffers that are too large waste resources.

Consider for example the model shown in figure 5. The output is an ordered sequence of integers of the form $2^n 3^m 5^k$, where n, m and k are non-negative integers. These are known as the Hamming numbers, and this program for computing them was studied by Dijkstra [13] and Kahn and MacQueen [27].

To understand how this program works, we need to understand what each of the processes does. The ones labeled `ScaleN` simply read an integer-valued input message, multiply it by *N*, and send the product to the output port. They repeat this sequence forever, as long as there are input messages. The `SampleDelayN` processes first produce an output message with value N, then repeat forever the following sequence: read the input message and send it to the output. As before, the black diamonds direct a single message sequence to multiple destinations (over separate, unbounded buffers).

The processes labeled `OrderedMergeM` are the only nontrivial processes in this program. These have a logic similar to the two-input `CONTROL` process in figure 3. Given a sequence of numerically increasing messages on each input port, they produce a numerically increasing merge of the two input sequences on their output ports. You can now understand how this program generates the Hamming numbers. The loop at the upper left produces the sequence 5, 25, 125, $\cdots$, 5^n, $\cdots$. That sequence is merged into the the second loop, which as a result produces the sequence 3, 5, 9, 15, 25, $\cdots$,

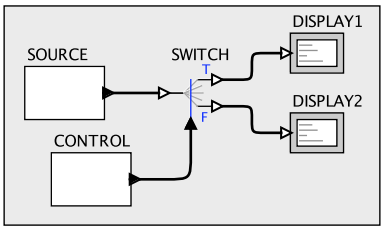

Figure 6: An illustration that demand-driven (lazy) execution does not solve the boundedness problem in general.

$5^n 3^m, \cdots$. That sequence is merged into the final loop, which produces the Hamming numbers, in numerically increasing order, without repetitions.

It is a simple exercise to prove that any finite bound on buffer sizes for the interprocess communications in figure 5 will cause the program to fail to produce all the Hamming numbers. What will happen when a buffer fills up and blocks an upstream process on a send operation? Does the program deadlock? These questions are subtle.

I claim that the execution platform should support the application designer by executing concurrent programs with bounded use of buffers when it is possible to do so. It should not be up to the application designer to figure out how to do that (by, for example, guessing at the buffer bounds and enforcing them). And the application designer should not be forced to do without buffering or implement the buffering as part of the application, as would be required in MPI, where the only real option for applications like those in figures 1 and 2 is to use synchronous communication or resort to explicit buffer management.

But executing programs with bounded buffers is not as easy as it might seem, as elaborated in the next three subsections.

3.3 Demand-Driven Execution

Looking at figures 1 and 2, it is tempting to note that a bounded execution would result from demand-driven execution, also known as lazy evaluation. Under such a policy, upstream processes are stalled until their outputs are required. This can be implemented using MPI, with some effort, by designing processes that mediate the communication. Although this delivers bounded execution for those particular examples, it does not solve the problem in general.

Consider the example shown in figure 6. Here, a process labeled `SWITCH` directs a stream of messages to two `DISPLAY` processes, depending on the messages from the `CONTROL` process. Suppose that the `CONTROL` process happens to always issue messages with value `true`. Then how should the system respond to demands from `DISPLAY2`? Depending on the implementation, the result could be deadlock or buffer overflow. The only way to prevent this result would be for the `DISPLAY` processes to have knowledge of the signal from `CONTROL` to regulate their issuance of demands. This would be extremely awkward.

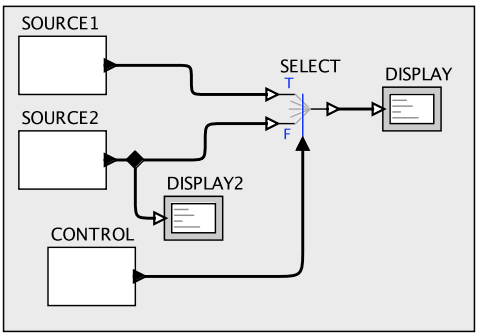

Figure 7: An illustration that output demand can trigger unbounded memory usage.

It is tempting to observe that for this example, data-driven execution would prevent buffer overflow. In data-driven execution, downstream processes would be scheduled to execute only when they have unprocessed input data, which would seem to solve the problem in this case. But this example could be naturally combined with the market data example of section 2.2. For example, after merging market data sorted by time stamp, we may want to then separate the data by security type. In that case, we have a mixed system, where neither data-driven nor demand-driven execution works.

Unless there is only exactly one consumer of data in the program, the question of when to generate "demand" can be quite subtle. Consider the example shown in figure 7. This is identical to figure 1 except for one additional `DISPLAY` process monitoring the output of `SOURCE2`. Again suppose that the `CONTROL`. process happens to always issue messages with value `true`. Unlike figure 1, there is an observer of the outputs of `SOURCE2`. Should `SOURCE2` generate an unbounded sequence of output messages? If so, it will overflow the buffers on the communication link to `SELECT`.

The example in figure 6 will execute with bounded memory if executed under a data-driven policy rather than a demand-driven policy. The example in figure 7 uses unbounded memory under both policies, and yet there is a correct bounded execution of this program. As programs grow in complexity, they are bound to include both of these patterns, so both demand and data driven scheduling policies fail.

Consider another example, shown in figure 8. This program has a feedback loop. If the process labeled `IO_PROCESS` reads its input using a blocking read before producing an output, then it will block immediately, and never be able to continue. The feedback loop is deadlocked, but the overall program is not, as long as the CONTROL process is able to produce outputs. If the `CONTROL` process is able to produce an unbounded number of control messages, then it will continue to execute until the communication buffers overflow.

Should this program be declared defective? If so, how can it be analyzed to determine that it is defective? What if `IO_PROCESS` produces an output before reading an input, and `CONTROL` generates an infinite sequence of `true`-valued messages? How will demand-driven scheduling deal with this situation? How about data-driven execution?

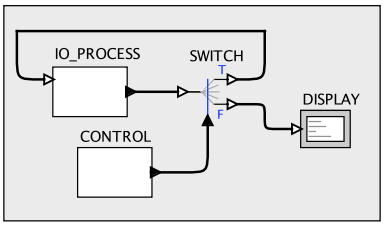

Figure 8: A program that has an infinite execution, but no bounded infinite execution. The CONTROL process is able to send messages to the SWITCH, but the SWITCH is unable to receive them because it is deadlocked with the IO_PROCESS. If the CONTROL process can execute forever, sending control messages, then memory will overflow.

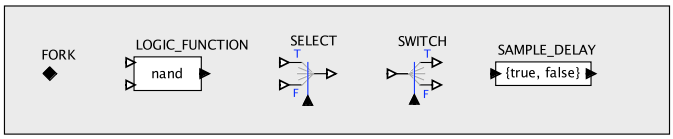

Figure 9: Five processes that, together with a boolean message data type, when connected in a KPN, provide a Turing-complete model of computation.

3.4 Undecidability of Buffer Bounds

For many message passing programs, a programmer can determine required buffer sizes without much difficulty. So given an implementation of bounded buffers with blocking reads and writes, it would not be hard to achieve bounded execution. Perhaps the Hamming numbers example and figure 8 of the previous section are anomalous. However, programming methodologies have never succeeded by assuming that programmers won't exercise their capabilities, sticking only to simple patterns. Moreover, the complexity and scale of message passing programs is bound to increase considerably with the drive towards multicore. What used to be anomalous may become commonplace. If buffer management is hard for some programs, then it is important for the software infrastructure to provide support for it.

The fact is that the buffer bounds question is fundamentally problematic. It can be shown that the question of whether a KPN program can execute forever with bounded buffers is undecidable [8]. This can be proven by showing that the five process definitions in figure 9 connected in a KPN, with only boolean message types, are sufficient to build a universal Turing machine. As a direct consequence, a number of questions about KPN programs become undecidable.

The processes in figure 9 are defined as follows:

- The `FORK` process reads one input stream and replicates it to multiple recipients. Although this can be implemented as a process, it is sufficiently fundamental that

it is better implemented as part of underlying KPN infrastructure.

- The `LOGIC_FUNCTION` process reads a boolean-valued message from each input port, performs a memoryless logic function such as AND, NAND, OR, or NOR on the data, and produces a boolean-valued message on the output with the result (NAND alone is sufficient, since the others can all be implemented with NAND).
- The `SELECT` process is the same one that we saw in figures 1 and 2.
- The `SWITCH` process performs the converse operation. It reads a boolean control input from the bottom port, then reads a data input from the left input port. If the control is `true`, then it produces the data value on the upper output port, labeled `T`. If the control is false, it produces the data value on the lower output port, labeled `F`. It then repeats this forever.
- Finally, the `SAMPLE_DELAY` process is the same one we encountered in figure 5. It produces an initial output, which has value `true` or `false` in this case, and then forever copies its input to the output.

We can build a universal Turing machine by interconnecting these processes according to KPN semantics. As a direct consequence, the question of whether the program deadlocks (halts) is undecidable, and the question of whether the program can execute with bounded memory is also undecidable. Since given these process definitions, the only (significant) memory is in the buffers, the question of whether a program can execute forever with bounded buffers is undecidable.

3.5 Executing KPN with Bounded Buffers

Fortunately, this particular undecidable problem is solvable. Parks gives a scheduling policy that executes every Kahn process network that can execute in bounded memory in bounded memory [43]. Parks' policy is quite simple. Begin with arbitrary bounds on all buffers, and block processes that attempt to write to a full buffer. Execute until the program deadlocks. If the program does not deadlock, then we have succeeded in executing with bounded buffers. If it does deadlock, then if there is any process that is blocked on a write to a buffer, increase the capacity of all buffers and continue.[2] If there is no process blocked on a write, then the program intrinsically deadlocks, and we can halt execution.

There is no contradiction here with undecidability. Parks' algorithm does not deliver the answer to the boundedness question in finite time. It does, however, ensure boundedness if the answer is that the program can execute with bounded memory. It does so for all programs, whereas fair, data-driven, or demand-driven policies always fail for some programs.

Note that implementing this policy under MPI would be quite difficult. First, a programmer would have to implement a bounded buffer with blocking reads and writes.

[2]There are more efficient alternatives than increasing the capacity of all buffers, but the proof is simple with this strategy.

Each buffer would be implemented by a process with synchronous send and receive. Second, the programmer would have to implement deadlock detection, which would be easier, though far from trivial, with shared data structures. This is simply too much to ask of application programmers. To be useful, it must be provided as part of the execution platform.

The correctness of this policy has not been entirely without controversy. Basten, et al., argue that the policy prevents convergence to the least fixed point of the Kahn semantics for some programs [4, 16]. This claim is somewhat tricky, since we need a notion of convergence to infinite sequences. Nonetheless, the claim is valid, and we can see that this it true with the examples in figures 1 and 2. Under the Kahn semantics, if the `SOURCE1`, `SOURCE2`, and `CONTROL` processes are able to produce an unbounded number of outputs, then the Kahn semantics is that all connections between processes carry infinite sequences. Those infinite sequences are the least fixed point. Basten et al. give a scheduling policy that is bounded (when a bounded policy exists) for a class of Kahn networks that they call *effective*. In an effective Kahn network, every message sent is eventually received. For non-effective programs, they provide no guarantee, and in fact allow for a fallback to Parks' algorithm, which will provide bounded execution if this is possible.

Application programmers should not have to understand these subtleties. It is almost certainly not central to whatever they are trying to accomplish.

3.6 Determinism

A program is said to be *determinate* if given the same inputs, it produces the same outputs on every execution. Although this definition seems straightforward, it is not. What do we mean by inputs and outputs? Is the time at which an input is provided part of the input? The time at which an output is provided? For some programs, in the designer's mind, these are definitely important. Any discussion of determinacy must make clear what is considered part of the input and output.

In classical Turing-Church computation, the input and output are each a fixed body of data that can be represented as a finite sequence of bits. For KPN, the question is a bit more complex.

A *closed* KPN is a network of processes where every sequence of messages is generated by a process in the network. An *open* KPN is one where at least one sequence of messages is provided from outside the KPN. An open KPN can always be modeled by a closed KPN if we consider the externally provided sequence of messages to be produced by a process that has no input message sequences. Thus, we need to only consider closed KPNs.

In a KPN, it is easy to define the inputs and outputs of an individual process. The inputs are input message sequences, delivered in the figures in this paper by input ports, and the outputs are output message sequences. Two sequences are identical if they contain the same messages in the same order. The time at which a message is sent or read is irrelevant, except that messages are read and written in order. A KPN process is said to be determinate if every output sequence from the process is a function of the input sequences. This must be true for finite or infinite sequences.

If a KPN process has no input sequences, then it is determinate if it produces exactly one possible output sequence on each output port. Since we model an open KPN as containing such processes, those processes are determinate if and only if the input sequences to the open KPN are known.

Two sequences may be related by a prefix relation. A sequence s_1 is said to be a prefix of another sequence s_2 if every message in s_1 is in s_2 in the same order, and any message in s_2 that is not in s_1 occurs after all of these messages. This is written $s_1 \sqsubseteq s_2$. This relation provides a nice mathematical structure exploited by Kahn [26]. Tutorial expositions can be found in [37, 35]. In this paper, we avoid the mathematics and focus on the intuition.

A determinate KPN process with one input sequence and one output sequence is said to be *monotonic* if $s_1 \sqsubseteq s_1'$ implies that $F(s_1) \sqsubseteq F(s_1')$, where F is the function from sequences to sequences implemented by the process. Intuitively, if an input sequence is extended with additional messages, the output sequence may only be extended. The process cannot "change its mind" about outputs previously produced, given additional input messages. The definition of monotonic processes is easily extended to processes with any number of input and output sequences. A process with no input sequences is monotonic if and only if it is determinate. A process with no output sequences is always monotonic.

A monotonic process is further said to be *continuous* if, intuitively, any output message that it produces is produced after only a finite number of input message. That is, it cannot produce outputs only in response to infinite input sequences. It cannot "wait forever" to produce outputs. A determinate process with no input sequences is always continuous. A process with no output sequences is also always continuous. An MPI process that does not probe for input messages, but rather just does blocking reads, is continuous as long as it does not interact with other processes via shared data structures (by using, for example, random number generators or buffer occupancy probes).

Kahn showed that any network of continuous processes is determinate [26]. This is the key advantage of KPN as a concurrent MoC. Although it is highly concurrent, and can result in very effective use of parallel resources, the results of a computation are repeatable. Every execution, regardless of parallelism and scheduling, yields the same results.

Without such a MoC to work with, programmers frequently use nondeterminate mechanisms even when they are constructing determinate programs. This is risky because errors may not show up in testing. It is far preferable to provide programmers with explicitly nondeterminate constructs that they only need to invoke when they actually need nondeterminism.

3.7 Extended Process Networks

A useful example of an explicitly nondeterminate construct is a *nondeterminate merge* or *fair merge*. A use of this is shown in figure 10, another variant of figure 1. In this variant, there is no `CONTROL` process, and instead we rely on the `NONDETERMINATE_MERGE`. This process probes both input channels, and when there is a message on either, it sends that message to the output. This could be useful in combination with fair scheduling,

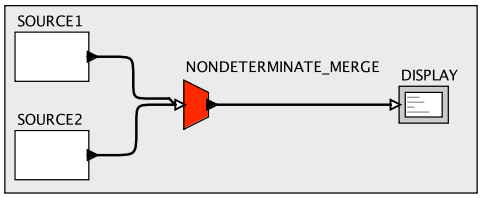

Figure 10: A nondeterminate version of figure 1 using extended process networks.

assuming we don't really care about the order in which messages from the `SOURCE` processes are delivered. There are many applications that have this character.

Brock and Ackerman have shown that any stream merge must be either unfair or nonmonotonic [7]. The `SELECT` process of figure 1 is, in fact, an unfair merge. It may completely ignore the messages from one of the `SOURCE` processes, if told to do so by the `CONTROL` process. `SELECT` is monotonic (and continuous). `NONDETERMINATE_MERGE` is not.

Extending an implementation of KPN to support `NONDETERMINATE_MERGE` is fairly straightforward, and has been done for example in Ptolemy II [14]. A programmer that wishes to build nondeterminism into an application does so explicitly by invoking this process. In contrast, maintaining determinacy with message-passing libraries like MPI requires that programmers know which capabilities to not use when. It is too easy for programmers to inadvertently introduce nondeterminate mechanisms.

3.8 Dataflow

The KPN MoC can be shown to be a generalization of dataflow models [37]. All the same scheduling issues apply to dataflow models; Parks' algorithm and Basten et al.'s extension, for example, are effective for executing dynamic dataflow programs with bounded memory [43, 4, 16].

In dataflow models, the components of a program have discrete, finite actions called *firings*, rather than the (potentially) unbounded executions of processes. A firing consumes some amount of input data from each input stream and produces some amount of data on each output stream. In dataflow, the components are also called *actors*, although they differ considerably from Agha's actors [2], which are processes. I have previously proposed broadening the use of the term "actors" to explicitly encompass both of these models and any concurrency model where interactions between components are via messages mediated by ports rather than via procedure calls [33].

A dataflow actor is defined by giving a firing procedure. An actor will fire repeatedly in response to available input data, and those repeated firings in fact define a process [35]. In fact, any KPN process can be converted into a dataflow actor, at the expense of requiring the programmer to do manual stack management, or "stack ripping" [1]. Specifically, since a firing is a finite discrete action, if the next firing must pick up where the current firing left off, then the programmer has to explicitly keep track of where it left off. The one and only advantage of defining a KPN process

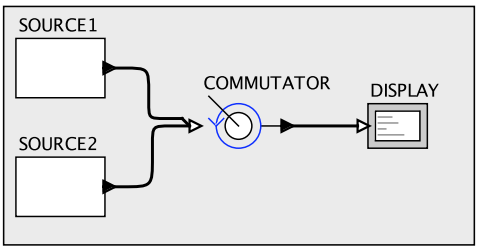

Figure 11: A version of the program in figure 1 that does round-robin scheduling.

over a dataflow actor is that if a process blocks due to unavailable input (or full output buffers), when the input becomes available (or room in the output buffers becomes available) it automatically picks up where it left off. When actors have little or no state, this cost is negligible. For example, the `SWITCH` and `SELECT` that we have been using have dataflow variants that have been called Distributor and Selector by Davis and Keller [11].

In exchange for this cost, however, we can accrue considerable benefits. If the actors all conform, for example, to the *synchronous dataflow* (SDF) model [36], then programs become statically analyzable for deadlock and bounded buffering, and amenable to static load balancing and scheduling.

Suppose, for example, that in figure 1 the application designer specifically designs the `CONTROL` process to issue messages that alternate in value between `true` and `false`. This property could be exposed to a compiler and/or scheduler by specifying an SDF model instead of a KPN model. Such an SDF model is shown in figure 11. Here, the `SELECT` process has been replaced with an SDF actor called `COMMUTATOR`, named after the electromechanical part in generators. This actor requires a single message on each input to fire, and when it fires, it produces those two messages, in order, on the output. As a consequence of this rather simple production and consumption pattern on messages, it is easy to analyze this program for deadlock, buffer sizes, and load balancing.

An SDF program is a network of dataflow actors with fixed production and consumption patterns for input and output messages. That is, for each actor, it is statically declared how many input messages are required on each input port to fire the actor, and how many messages will be produced on each output port when it fires. Such programs can always be statically analyzed for deadlock, buffer sizes, and, if execution loads of the fire procedures are known, for load balancing [45]. The model of computation is not Turing complete, and none of these questions is undecidable.

Of course, analyzability comes at a price in expressiveness. Many researchers have augmented the SDF MoC with more expressive formalisms that preserve decidability. For example, the *heterochronous dataflow* (HDF) MoC defines an actor as a finite state machine, where production and consumption patterns can vary in each state [18]. With careful constraints on when state transitions can be taken, this model remains decidable. These analyzability properties have been heavily exploited in StreamIT [47]

and LabVIEW [28], which provide structured programming constructs for SDF and some HDF patterns. Moreover, SDF and its variants can be combined with KPN or extended PN [19], so we really have lost nothing in expressiveness.

4 Conclusion

There are many other concurrent MoCs that could prove interesting in the long run for parallel and concurrent programming. We have only touched on the possibilities here, merely to emphasize that solutions exist that provide much more structure to programmers than what today's message passing libraries provide, and that this structure has real value.

A good programming model is as much about *constraints* on the programmer as it is about *capabilities* offered to the programmer. Message passing libraries provide rather low-level mechanisms that give the programmer enormous freedom to implement a variety of concurrent operations. But this freedom comes at a price. Programs may have inadvertent nondeterminism, lack of portability, or brittleness, where for example small changes in the operating environment can cause buffers to overflow. Probes of message inputs to determine whether a message is available may seem innocent enough to a programmer, but if a process branches based on the result, the result can easily turn into inadvertent nondeterminism.

While these libraries are too rich, providing too few constraints or too little structure, they are at the same time too impoverished. I have given programming examples where it is easy to see intuitively how they should be executed to avoid overflowing communication buffers, but achieving this disciplined execution using MPI proves to be extremely difficult. Such disciplined execution requires infrastructure-level support.

Gorlatch argues that send-receive are like the goto's of concurrent programming [20]. In this paper, I support this indictment, but I have shown that his solution, the use of collective operations like MPI's gather and scatter is incomplete.[3] Those operations don't cover, for example, the program in figure 4. And even if they were extended to do so, they would not, by themselves, solve the problem of bounding the buffers while maximizing concurrency. To solve those problems, we need a disciplined concurrent model of computation.

Such low-level mechanisms may acceptable when performance is the overriding concern. Programmers may be willing to construct non-portable programs that are very laborious to get right. But such programming practice is not a practical alternative for mainstream programming of mainstream computers, such as the next generation of multicore machines.

There are many candidates for more disciplined models of computation, many of which my group has implemented in Ptolemy II [14]. In fact, every graphical rendition

[3] Interestingly, Gorlatch's solution and mine can be combined in an interesting way. The SWITCH and SELECT actors that I have used in the examples in this paper are arguably as evil as goto statements in imperative code. In fact, they are being used in these examples to have data flow imply control flow. Used in an unstructured way, they can lead to incomprehensible programs even when a disciplined MoC is being used. The solutions is to combine a disciplined MoC with collective patterns. Such a technique underlies the structured dataflow model of LabVIEW [28] and StreamIt [47]. More extensible is to treat these patterns as combinators or higher-order components [37].

of a message passing program in this paper is a screen image of an executable Ptolemy II program. Most use the PN director, which implements a Kahn process networks model of computation extended with a nondeterminate merge. The others use the SDF director, which implements a synchronous dataflow model of computation. There are many more possibilities, including some that control timing of execution. Most of these models of computation can be combined in a single program (see [18, 14, 19]), providing interesting possibilities for systematically composing design patterns.

Disciplined concurrent models of computation can be implemented on top of threads and/or message passing libraries. In my research group, we have done both. Thus, the argument here is not that these mechanisms should be abandoned. The argument is that they provide inadequate programming models for application programmers. They should be used by infrastructure builders, not application programmers.

A number of interesting open questions remain, leaving considerable research to be done. First, if message passing libraries are to be replaced with higher-level mechanisms, what form should those take? Do we need new programming languages? Languages like Scala [41], with its deep roots in Java and functional languages, could overcome some of the innate resistance to new languages. An alternative is to treat concurrency as a software component problem rather than a programming language problem [33]. This may still require new languages, just as object-oriented component technology did, but those new languages can be merely elaborations of familiar languages, as Java and C++ are elaborations of C. Another interesting open question strikes at the tired old debate of message passing vs. shared memory. Concurrent MoCs with message passing roots do not always handle shared state well. Complex applications are likely to require some shared state. Can mechanisms for managing shared state be cleanly integrated with disciplined concurrent MoCs that are based on message passing? Finally, message passing overhead biases applications to coarse-grain systems, where relatively large data sets are exchanged in messages. Can efficiency be improved enough to be effective at fine granularity? There are many ideas out there, but turning them into useful and understandable programming models has yet to occur.

5 Acknowledgments

Thanks for very helpful comments on early drafts of this paper to Eric Brewer, Christopher Brooks, Dai Bui, Thomas Huning Feng, Jim Larus, Ben Lickly, Slobodan Matic, Bert Rodiers, Stavros Tripakis, and Jia Zou.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *USENIX*, 2002.

[2] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. N. Stefani and J.-B., editors, *Formal Methods for Open Object-based Distributed Systems, IFIP Transactions*. Chapman and Hall, 1997.

[3] F. Arbab. Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[4] T. Basten and J. Hoogerbrugge. Efficient execution of process networks. In *Communicating Process Architectures*, pages 1–14, Bristol, UK, 2001. IOS Press.

[5] R. v. Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *10th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, 2003.

[6] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[7] J. D. Brock and W. B. Ackerman. Scenarios, a model of non-determinate computation. In *Conference on Formal Definition of Programming Concepts*, volume LNCS 107, pages 252–259. Springer-Verlag, 1981.

[8] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Ph.d. thesis, University of California, Berkeley, 1993.

[9] J. A. Cataldo. The power of higher-order composition languages in system design. PhD Thesis Technical Report UCB/EECS-2006-189, EECS Department, University of California, Berkeley, December 18 2006.

[10] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[11] A. L. Davis and R. M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, 1982.

[12] E. W. Dijkstra. Go to statement considered harmful (letter to the editor). *Communications of the ACM*, 11(3):147–148, 1968.

[13] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[14] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.

[15] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[16] M. Geilen and T. Basten. Requirements on the execution of Kahn process networks. In *European Symposium on Programming Languages and Systems*, LNCS, pages 319–334. Springer, April 7-11 2003.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[18] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions On Computer-aided Design Of Integrated Circuits And Systems*, 18(6):742–760, 1999.

[19] A. Goderis, C. Brooks, l. Altintas, and E. A. Lee. Composing different models of computation in Ptolemy II and Kepler. In *International Conference on Computational Science (ICCS)*, to appear, 2007.

[20] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, 2004.

[21] W. Gropp and E. Lusk. Goals guiding design: PVM and MPI. In *Cluster*, Chicago, Illinois, September 23-26 2002.

[22] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing*. MIT Press, second edition edition, 1999.

[23] B. Hayes. Computing in a parallel universe. *American Scientist*, 95:476–480, 2007.

[24] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, Tahoe City, CA, 2001. Springer-Verlag.

[25] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), 1978.

[26] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[27] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing*, pages 993–998. North-Holland Publishing Co., 1977.

[28] J. Kodosky, J. MacCrisken, and G. Rymar. Visual programming using structured data flow. In *IEEE Workshop on Visual Languages*, pages 34–39, Kobe, Japan, 1991. IEEE Computer Society Press.

[29] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Second International Symposium on Operating Systems*, volume 13, pages 3–19, 1978. http://doi.acm.org/10.1145/850657.850658 ACM Press reprinted in Operating Systems Review, 13,2 April 1979, pp. 3-19.

[30] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading MA, 1997.

[31] D. Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, 2005.

[32] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7:25–45, 1999.

[33] E. A. Lee. Model-driven development - from object-oriented design to actor-oriented design. In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop)*, Chicago, September 24 2003.

[34] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[35] E. A. Lee and E. Matsikoudis. The semantics of dataflow with firing. In G. Huet, G. Plotkin, J.-J. Lvy, and Y. Bertot, editors, *From Semantics to Computer Science: Essays in memory of Gilles Kahn*. Cambridge University Press, 2009. Draft version available at http://ptolemy.eecs.berkeley.edu/publications/papers/08/DataflowWithFiring/.

[36] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[37] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[38] B. Ludscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency & Computation: Practice & Experience*, 18(10):1039 – 1065, 2006.

[39] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[40] Message Passing Interface Forum. MPI: A message passing interface standard – version 2.1. Technical report, University of Tennessee, Knoxville, Tennessee, June 23 2008.

[41] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the scala programming language. Technical Report LAMP-REPORT-2006-001, cole Polytechnique Fdrale de Lausanne (EPFL), 2006. Second Edition.

[42] J. K. Ousterhout. Why threads are a bad idea (for most purposes) (invited presentation). In *Usenix Annual Technical Conference*, 1996.

[43] T. M. Parks. *Bounded Scheduling of Process Networks*. Phd thesis, UC Berkeley, 1995.

[44] T. M. Parks. A comparison of MPI and process networks. In *International Parallel and Distributed Processing Symposium, Workshop on Java for Parallel and Distributed Computing*, Denver, CO, April 2005.

[45] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc. (now Taylor and Francis), 2000.

[46] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[47] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, volume LNCS 2304, Grenoble, France, 2002. Springer-Verlag.

[48] R. van Renesse. Goal-oriented programming, or composition using events, or threads considered harmful. In *Eighth ACM SIGOPS European Workshop*, September 1998.

[49] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles (SOSP)*, volume 35(5), pages 230–243, Banff, Canada, 2001. ACM.

[50] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference*, San Antonio, Texas, USA, June 9-14 2003.