Distributed Process Networks in Java *

Thomas M. Parks parks@cs.colgate.edu
David Roberts droberts@colgatealumni.org
David Millman dmillman@colgatealumni.org

Computer Science Department
Colgate University
Hamilton, NY 13346

Abstract

Kahn defined a formal model for networks of processes that communicate through channels carrying streams of data tokens. His mathematical results show the necessary conditions for an implementation to be determinate, that is, for the results of the computation to be identical whether the processes are executed sequentially, concurrently, or in parallel. In our Java implementation channels enforce blocking reads and each process has its own thread in order to ensure determinacy and avoid deadlock. The network connections required to maintain the communication channels between processes executing on separate servers are automatically established when parts of the program graph are distributed to other servers by Java Object Serialization. Our Java implementation of process networks is suitable for execution on a single computer, a cluster of servers on a high-speed LAN, or geographically dispersed servers on the Internet.

1 Introduction

Kahn defined a formal model for networks of concurrent processes that communicate through first-in first-out channels carrying streams of data elements [10, 11]. Processes produce data elements and send them along a communication channel where they are stored until the destination process consumes them. Communication channels are the *only* method processes may use to exchange information. For each channel there is a single process that produces data and a single process that consumes that data. Multiple producers or multiple consumers connected to the same channel are not allowed. [1] Figure 1 shows a simple process network with three processes connected by two channels. An example of a process network that computes Fibonacci numbers is shown in Figure 2.

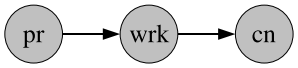

Figure 1: A simple process network with a `Producer`, `Worker`, and `Consumer` arranged in a pipeline.

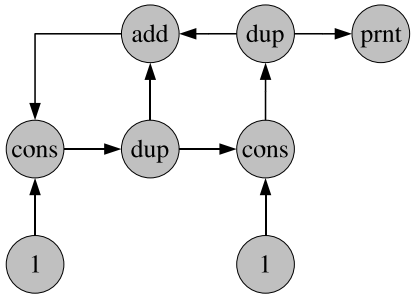

Figure 2: A process network that computes Fibonacci numbers, which are recursively defined by the equation $f_n = f_{n-1} + f_{n-2}$. The `Cons` process inserts an element at the head of a stream. The `Duplicate` process creates multiple copies of a stream. The `Print` process prints the elements of a stream. The `Add` process adds two streams element-wise.

Kahn requires that execution of a process be suspended when it attempts to get data from an empty input channel. A process may not, for example, examine an input to test for the presence or absence of data. He showed that requiring processes to block when attempting to read from empty channels allows processes to be represented as continuous functions over a complete partial order (the set of streams of data elements with a prefix order). A pro-

*This is an expanded version of a paper of the same title that was presented at the International Workshop on Java for Parallel and Distributed Computing held in conjunction with the International Parallel and Distributed Processing Symposium in Nice, France, April 2003.

[1] Kahn allows several consuming processes to share a channel with each consumer receiving a copy of the same data sequence. Instead, we allow only a single consuming process for each stream but have processes, such as `Duplicate`, that explicitly duplicate streams.

gram graph can be represented as a collection of equations that have a unique minimum solution that corresponds to the history of all data elements produced on all streams. Thus systems that obey Kahn's model are *determinate*: the history of data elements produced on the communication channels is uniquely determined by the equations representing the program graph and does not depend on the execution order [10]. This implies that so long as blocking reads are enforced, the results of a computation are unique and correct whether the program is executed on a computer with a single processor, a computer with multiple processors, or many computers distributed across a network.

Individual processes can be defined in a *host language*, such as C++ or Java, with the semantics of the network serving as a *coordination language*. Care must be taken when writing code in the host language. It may be tempting to use shared variables or other mechanisms in the host language to circumvent the communication channels, but this may violate Kahn's model and result in a nondeterminate system. With a little discipline it is not difficult to write determinate programs.

Because process networks expose parallelism and make communication explicit, they are well suited to a variety of signal processing and scientific computation applications. The process network model, or a special case of process networks such as dataflow [12], is used in applications such as embedded signal processing [4], sonar beam forming [1], and image processing for geographical information systems [15].

1.1 Related Work

We have previously implemented process networks in C++ [13] and Java [14], as have others [1, 9, 15, 16]. Some have even used CORBA to implement distributed process networks [2]. Our current implementation has several advantages over these other implementations. Unlike [1, 9, 13, 14] our implementation is designed for distributed computing. Our program graphs can run not only on multi-processor servers, as with these other implementations of process networks, but also on a cluster of servers on a local network or even on a loose collection of servers on the Internet. The implementation described in [15, 16] does support distributed computing, however all communication passes through a central server. Even processes running on the same server must communicate with each other indirectly through the central server. In our implementation all communication between processes, whether running on the same server or on different servers, is direct and does not pass through the central server.

The implementation described in [2] is most similar to ours. One important difference is that dynamic reconfiguration, in which the program graph is modified during execution, are managed from a central console in their system. In our system the program can be self-modifying, so reconfigurations occur locally rather than centrally. Because we use Java rather than CORBA, we can easily distribute executable implementations of classes to remote compute servers. Our implementation also takes advantage of features of Java Object Serialization to automate the establishment of network connections that are necessary for communication between processes running on different servers.

Before we go into detail about our implementation of process networks, we first review the formal underpinnings of this model of computation. This will establish sufficient conditions for our implementation to produce determinate systems.

2 Formal Properties of Process Networks

In Kahn's mathematical representation of process networks [10] communication channels are associated with streams (sequences of data elements), possibly infinite in length, and processes are continuous functions operating on streams. This allows us to describe a process network by a set of equations.

2.1 Streams

A stream is a finite or infinite sequence of data elements: $X = [x_1, x_2, x_3, \ldots]$. The empty stream is represented by the symbol $\perp$. Let S be the set of all such streams. Consider a *prefix ordering* of streams, where the stream X *precedes* the stream Y (written $X \sqsubseteq Y$) if X is a prefix of (or is equal to) Y. It could also be said that Y *extends* X. For example, the stream $X = [0]$ is a prefix of the stream $Y = [0, 2]$, which is in turn a prefix of $Z = [0, 2, 1, 3, 4, \ldots]$. The empty stream $\perp$ is a prefix of all streams.

The prefix relation $\sqsubseteq$ is reflexive ($X \sqsubseteq X$), antisymmetric (if $X \sqsubseteq Y$ and $Y \sqsubseteq X$ then $X = Y$), and transitive (if $X \sqsubseteq Y$ and $Y \sqsubseteq Z$ then $X \sqsubseteq Z$). Thus, the prefix relation $\sqsubseteq$ defines a partial order over the set S. Any (possibly infinite) increasing chain of streams $\vec{X} = (X_1, X_2, X_3, \ldots)$ with $X_1 \sqsubseteq X_2 \sqsubseteq X_3 \sqsubseteq \cdots$ has a least upper bound $\sqcup\vec{X}$ such that $X_i \sqsubseteq \sqcup\vec{X}$ for all $X_i \in \vec{X}$. The least upper bound can be interpreted as a limit:

$$\sqcup\vec{X} = \lim_{i \to \infty} X_i \tag{1}$$

Thus, the prefix relation $\sqsubseteq$ defines a *complete partial order* over the set S because every increasing chain has a least upper bound that is also in S.

2.2 Processes

A process is a functional mapping from an input stream to an output stream $f : S \to S$. A function f maps an increasing chain $\vec{X} = X_1 \sqsubseteq X_2 \sqsubseteq X_3 \sqsubseteq \cdots$ into another set of streams $\vec{Y} = Y_1, Y_2, Y_3, \ldots$ where $Y_i = f(X_i)$ for all $X_i \in \vec{X}$. In general, the resulting set of streams $\vec{Y}$ may or may not be an increasing chain.

A function f is defined to be *continuous* if and only if for any increasing chain

$$f(\lim_{i\to\infty} X_i) = \lim_{i\to\infty} f(X_i) \tag{2}$$

Continuity can also be interpreted in terms of least upper bounds:

$$f(\sqcup\vec{X}) = \sqcup\vec{Y} \tag{3}$$

This implies that $\vec{Y}$ is itself an increasing chain when f is continuous. Clearly then, the composition of two continuous functions f and g is itself continuous:

$$f(g(\lim_{i\to\infty} X_i)) = \lim_{i\to\infty} f(g(X_i)) \tag{4}$$

Functions that map an increasing chain into another increasing chain are said to be *monotonic*:

$$X \sqsubseteq Y \implies f(X) \sqsubseteq f(Y) \tag{5}$$

All continuous functions are monotonic[12], and the composition of two monotonic functions f and g is itself monotonic:

$$X \sqsubseteq Y \implies f(g(X)) \sqsubseteq f(g(Y)) \tag{6}$$

The following functions are examples of continuous mappings:

first(U) Returns the first element of the stream U. By definition, first$(\perp) = \perp$.

rest(U) Returns the stream U with the first element removed. By definition, rest$(\perp) = \perp$.

cons(x,U) Inserts a new element x at the beginning of the stream U. By definition, cons$(\perp,U) = \perp$, and cons$(x,\perp) = [x]$.

All of our discussion thus far can be generalized from streams in the set S to p-tuples of streams in the set S^p: $\vec{X} = (X_1, X_2, \ldots, X_p) \in S^p$. Then $\perp^p \in S^p$ is a set of empty streams. The prefix ordering for elements of S^p is then defined such that $\vec{X} \sqsubseteq \vec{Y}$ if and only if $X_i \sqsubseteq Y_i$ for all $X_i \in \vec{X}$. The prefix relation $\sqsubseteq$ defines a complete partial order over the set S^p. We can also have functions that map a set of input streams to a set of output streams $f : S^p \to S^q$. The definitions of continuous and monotonic functions for S^p are analogous to those given above for S.

Using the mathematical representation of communication channels and processes, we can describe a process network as a set of equations with functions operating on sets of streams. If fact, we can compose all of the functions in a process network into a single function f that is continuous (and monotonic) and combine all of the streams in the network into a single p-tuple $\vec{X}$.

With a continuous function over a complete partial order there is a unique minimal (in a prefix sense) solution to the fixed point equation:

$$\vec{X} = f(\vec{X}) \tag{7}$$

We can use an iterative procedure to solve for the least fixed point. Initially we let all streams be empty, $\vec{X}_0 = \perp$. Then we extend the inputs using the previous output of the function, $\vec{X}_1 = f(\vec{X}_0) = f(\perp)$. We continue this procedure, repeatedly extending the inputs, $\vec{X}_{j+1} = f(\vec{X}_j)$. Because f is monotonic (as a consequence of being continuous) the sequence $\vec{X}_0 \sqsubseteq \vec{X}_1 \sqsubseteq \vec{X}_2 \sqsubseteq \ldots \sqsubseteq \vec{X}_j \sqsubseteq \ldots$ forms an increasing chain. Because S^p is a complete partial order, this increasing chain has a unique least upper bound. This least upper bound is a solution to the fixed point equation.

Kahn's blocking read semantics provide sufficient conditions for the function representing a process to be continuous. When the functions are continuous mappings over a complete partial order, there is a unique least fixed point that corresponds to the histories of data elements produced on the communication channels [10]. Because the least fixed point is unique, the channel histories are determined by the definitions of the processes and the network describing the communication between them. The number of data elements produced, and their values, are determined by the definition of the system and are not affected by the choice of execution order. We call such systems *determinate*.

3 Process Networks in Java

In creating a Java implementation of process networks, one of our goals has been to follow existing conventions in the Java API so that programmers will find our implementation easy to use correctly. There are some constraints of Kahn's process network model that we have chosen not to strictly enforce, so we depend on ease-of-use to promote the discipline required to create programs that conform to these constraints. It would not be impossible to enforce these restrictions, such as having only a single producer and a single consumer process for each stream, but this would incur some run-time overhead. Alternatively, a visual front end could be used for programming. This front end would generate Java code to implement the process network graph that the programmer had drawn. The responsibility for consistency checking could be given to

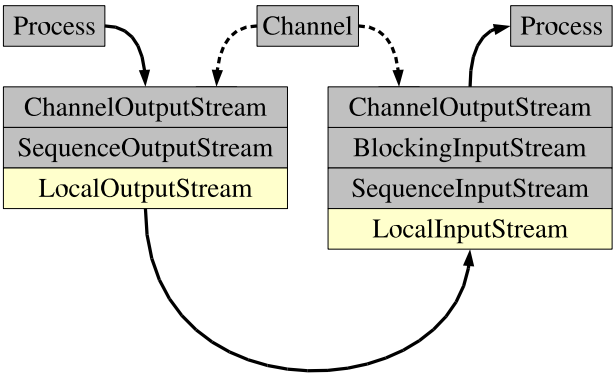

Figure 3: Classes used at each layer to implement the communication channels used by processes. The class at the lowest layer can be replaced to provide communication between processes on the same server or different servers.

this visual front end, relieving the run-time system of this burden.

3.1 Channels

The connections between processes that are denoted mathematically by streams are embodied operationally as first-in first-out queues called *channels*. Figure 3 shows some of the classes in our implementation of channels. A `Channel` object represents a connection between `Process` objects. The `getOutputStream` and `getInputStream` methods provide indirect access to the `Channel` through Java streams. Because the `write` and `read` methods of Java streams operate on bytes or arrays of bytes, the individual bytes passing through a `Channel` correspond naturally to the data elements of the mathematical representation of streams. Thus all `Process` objects communicate through streams of bytes by using the `write` and `read` methods of `ChannelOutputStream` and `ChannelInputStream`.

In order to preserve the mathematical semantics of streams, read operations on channels *must* block if no data is available [11]. Unfortunately, `java.io.InputStream` allows non-blocking read operations. When reading an array of bytes, the operation may complete early, returning fewer bytes than were requested. Our `BlockingInputStream` class enforces blocking reads.

The actual transfer of data across a channel takes place at the lowest layer in the diagram shown in Figure 3. Our `LocalOutputStream` and `LocalInputStream` classes extend the `java.io.PipedOutputStream` and `java.io.PipedInputStream` classes to provide communication through a shared buffer in memory. We have other input and output classes that use network sockets and, as we discuss later, it may be necessary to switch the underlying communication mechanism at the lowest layer while the program executes. We accomplish this switching with the `SequenceOutputStream` and `SequenceInputStream` classes. These classes encapsulate a sequence of streams and can safely switch their underlying Java stream, ensuring that all bytes are delivered in order and preserving the first-in first-out queue semantics for channels.

Even though all communication between processes takes the form of streams of bytes, a process may send more complex data types across a channel by layering a `java.io.DataOutputStream` or `java.io.ObjectOutputStream` over a `ChannelOutputStream`. Methods of these classes such as `writeInt`, `writeDouble`, and `writeObject` serialize these more complex types, converting them into a sequence of bytes that are written to the underlying `ChannelOutputStream`. A receiving process layers a `java.io.DataInputStream` or `java.io.ObjectInputStream` over the corresponding `ChannelInputStream` in order to correctly interpret the incoming stream of bytes.

Because higher level formatting of the byte stream is carried out within a process, we can retain the model of processes communicating only through streams of bytes. Some processes, such as `Cons` and `Duplicate` simply process bytes and need not be aware of any structure within a byte stream. This allows us to have a single, type-independent implementation of such processes.

3.2 Processes

In our implementation of process networks, each process executes in its own Java thread using our `Process` interface. This makes it possible to exploit the parallelism available in the program graph. To implement a new type of process, one simply defines a new class that implements the `Process` interface and provides an implementation of the `run` method. Because many processes are iterative with one-time initialization and cleanup code that should be executed as the process starts or stops, we provide `IterativeProcess` as an abstract base class for developing new process types. This class provides several protected methods (`onStart`, `step`, and `onStop`) with useful default implementations in addition to a default implementation of the `run` method, as shown in Figure 4.

The default implementation of `onStart` does nothing, so a new class should provide an implementation only if one-time initialization that is not appropriate for the constructor is needed. The provided implementation of `run`

```
public void run()
{ try
  { onStart();
    if(iterations > 0)
      while(iterations-- > 0) step();
    else while(true) step();
  }
  catch(IOException ignored) {}
  finally { onStop(); }
}
```

Figure 4: The `run` method of `IterativeProcess`. The `onStart` method is invoked once as execution begins. The `step` method is executed until an iteration limit is reached (if such a limit is specified) or until an exception occurs. The `onStop` method is invoked once as execution ends.

```
protected void step() throws IOException
{ int b = input.read();
  for(int i = 0; i < outputs.length; i++)
    outputs[i].write(b);
}
```

Figure 5: The `step` method of `Duplicate`.

calls this method first before beginning the main work of the process.

The default implementation of `step` does nothing, so a new class should provide an implementation that performs one step of the process's work. The provided implementation of `run` repeatedly invokes `step` for the specified number of iterations (or indefinitely if no iteration limit was specified in the constructor).

The default implementation of `onStop` closes all of the input and output streams associated with the process. If a process terminates, either because the iteration limit was reached or because `onStart` or `step` encountered an exception, then all of that process's input and output streams are closed.

Often a new process type can be implemented by simply extending `IterativeProcess` and implementing the `step` method. For example, the `step` method for a `Duplicate` process that creates multiple copies of a stream is shown in Figure 5.

We have also defined a `CompositeProcess` class that can express hierarchy in the program graph. Because requiring the `step` methods of the component processes to be executed in sequence could lead to deadlock, we retain a separate thread for each process within a `CompositeProcess` to avoid introducing deadlock through composition. Figure 6 shows code using a `CompositeProcess` that constructs the process network

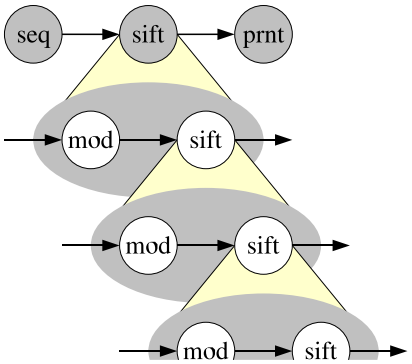

Figure 7: A self-modifying process network that computes the sequence of prime numbers using the Sieve of Eratosthenes. `Modulo` processes are inserted to remove multiples of each newly encountered prime.

for computing Fibonacci numbers.

3.3 Reconfiguration

Processes and the channels that connect them may be added to or removed from a process network not only during the initial construction of the graph but also during execution. Care must be taken to preserve any unconsumed data residing in the channels at the time that reconfiguration takes place. In order to maintain determinism it is important that reconfiguration be initiated by processes and not some external agent.

An example of a self-reconfiguring process network treated by Kahn and MacQueen [11] is shown in Figure 7. This process network computes prime numbers using the Sieve of Eratosthenes. The sequence of integers beginning with 2 is operated on by the `Sift` process and the resulting sequence of prime numbers is printed. The `Sift` process can be defined recursively or iteratively. In a recursive definition, as shown in Figure 7, the `Sift` process replaces itself with a `Modulo` process, which filters out multiples of a newly encountered prime, and a new `Sift` process. In an iterative definition, as shown in Figure 8, the `Sift` process repeatedly inserts new `Modulo` processes ahead of itself but remains in the program graph.

When a new `Modulo` process is inserted into the graph, it is assigned the input channel previously belonging to the `Sift` process. Because the rearrangement of channel connections and the activation of the `Modulo` process is controlled by the `Sift` process, determinism is preserved. When activated, the `Modulo` process reads from the channel precisely where the `Sift` process left off; data elements are neither lost nor repeated.

An example of a self-reconfiguring process network in which processes are removed from the program graph is

```
Channel ab, be, cd, df, ed, eg, fg, fh, gb;
ab = new Channel(); be = new Channel(); cd = new Channel();
df = new Channel(); ed = new Channel(); eg = new Channel();
fg = new Channel(); fh = new Channel(); gb = new Channel();

CompositeProcess p = new CompositeProcess();

p.add(new  Constant(1, ab.getOutputStream(), 1));
p.add(new      Cons(ab.getInputStream(), gb.getInputStream(),  be.getOutputStream()));
p.add(new Duplicate(be.getInputStream(), ed.getOutputStream(), eg.getOutputStream()));
p.add(new       Add(eg.getInputStream(), fg.getInputStream(),  gb.getOutputStream()));
p.add(new  Constant(1, cd.getOutputStream(), 1));
p.add(new      Cons(cd.getInputStream(), ed.getInputStream(),  df.getOutputStream()));
p.add(new Duplicate(df.getInputStream(), fh.getOutputStream(), fg.getOutputStream()));
p.add(new     Print(20, fh.getInputStream()));

new Thread(p).start();
```

Figure 6: An example showing the construction of a process network program graph to compute Fibonacci numbers. The visual representation of this program graph is shown in Figure 2.

```
protected void step() throws IOException
{ long prime = inputs[0]).readLong();
  outputs[0].writeLong(prime);
  Channel ch = new Channel();
  OutputStream out = ch.getOutputStream();
  Modulo mod = new Modulo(in, out, prime);
  in = channel.getInputStream();
  inputs[0] = new DataInputStream(in);
  new Thread(mod).start();
}
```

Figure 8: The `step` method of `Sift`. A `Modulo` process is inserted into the program graph to filter out multiples of each prime number read from the input.

shown in Figures 2 A `Cons` process prepends an initial constant at the head of a stream of data. After this a `Cons` process functions as an identity process, copying data from its input to its output. To avoid unnecessary copying of data and improve efficiency, the `Cons` processes remove themselves from the program graph. The resulting, reconfigured process network is shown in Figure 9.

When a process removes itself from the program graph, it is important to preserve all data elements in the channels associated with the process. This preservation is accomplished using the `SequenceInputStream` contained within every `ChannelInputStream`, as shown in Figure 10. When process `b` removes itself from the graph it splices together channels 1 and 2. The `InputStream` associated with the channel 1 is appended to the `SequenceInputStream` from which process `c` is reading. Because process `b` no longer produces new data on the channel 2, process `c` eventually reaches the end of the data stream. The`InputStream` for the channel 1 then becomes active and process `c` continues to read data without interruption.

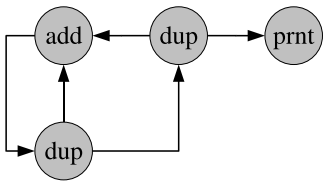

Figure 9: A reconfigured process network from which the `Cons` processes have removed themselves.

3.4 Termination

Signal processing and scientific applications are often intended to operate indefinitely with (conceptually) infinite streams of data. Other applications have a well-defined point at which they terminate. Our implementation of process networks supports both non-terminating and terminating programs.

Any process can have a fixed iteration limit imposed upon it, as seen in Figure 4. To compute all prime numbers less than 100 in the process network of Figure 7, we can impose an iteration limit on the `Sequence` process so that it produces the sequence of integers from 2 to 100 and then stops. To compute the first 100 prime numbers, which is a somewhat different task, we can impose an iteration limit on the `Print` process so that it stops after

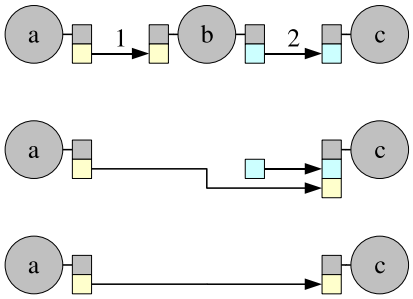

Figure 10: A reconfiguration, shown in 3 stages, in which a process removes itself from the program graph, splicing together its input and output channels.

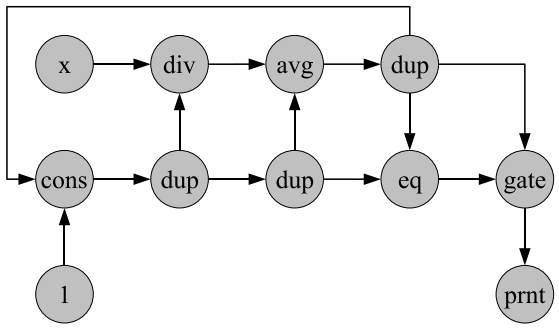

Figure 11: A process network that uses Newton's method to compute square roots according to the equation $r_n = (x/r_{n-1} + r_{n-1})/2$. Execution terminates when the limits of precision of the floating-point representation have been reached and the root estimate stops changing.

printing 100 numbers.

Closing an `InputStream` causes an exception to occur the next time the corresponding `OutputStream` is written to. When computing the first 100 prime numbers, the `Print` process stops after reaching its iteration limit and closes its channel. When the `Sift` process next attempts to write to its output, it encounters an exception, stops, and closes all its channels. This chain of events continues until all `Modulo` processes terminate and, finally, the `Sequence` process terminates. The `Sift`, `Modulo`, and `Sequence` processes may perform some unnecessary computation producing data that accumulates in channels but is never consumed, but all of the processes do terminate almost immediately after the `Print` process stops.

Closing an `OutputStream` does not induce and exception upon the next read of the corresponding `InputStream`. Instead an exception occurs only after the end of the data stream is reached. When computing all prime numbers less than 100, the `Sequence` process stops after reaching its iteration limit and closes its channel. The first `Modulo` process continues to execute, consuming the accumulated data on its input channel. When this data runs out, the `Modulo` process encounters an exception, stops, and closes all its channels. The next `Modulo` process also continues to run until it consumes all of the data on its input, at which point it encounters an exception, stops, and closes all of its streams. This chain of events continues until all `Modulo` processes, the `Sift` process, and the `Print` process terminate. In this case no unnecessary computation occurs and all data produced is eventually consumed. Termination is not immediate because downstream processes continue to execute until they have consumed all available data.

Data-dependent limits are also possible. For example, the `Guard` process in Figure 11 can be configured to stop after processing the first `true` value from its control input. In this example the `Divide` process divides two numbers, the `Average` process averages its inputs, the `Equal` process tests for equality, and the `Guard` process passes data to its output when its control input is `true` and discards data otherwise. This process network uses Newton's method to compute increasingly accurate estimates of the square root of x. When the limits of precison are reached and the estimate stops changing, the `Equal` process emits a `true` value on its output, causing the `Guard` to pass one value to the `Print` process and stop.

When one process stops, a chain reaction occurs that eventually leads to the termination of all processes in the program graph. Because a process closes all of its channels when it stops, the processes to which it is connected eventually encounter an exception that causes them to stop – and close all of their channels. These exceptions even propagate across network connections, leading to the graceful termination of all processes in a distributed execution environment. No remote processes are left running, consuming resources, after other processes have completed execution.

3.5 Limited-Capacity Channels

In Kahn's process network model, channels have unlimited capacity and writes are non-blocking. In Java, streams associated with pipes and network sockets inherently have limited buffer capacity and the `write` method of `java.io.OutputStream` blocks if the buffer is full. As it turns out, such blocking writes are useful for ensuring fair scheduling of processes. In the absence of any structural constraints in the program graph, such as directed cycles, allowing processes to run freely could unnecessarily consume large amounts of memory.

For example, in Figure 1, if the `Producer` process

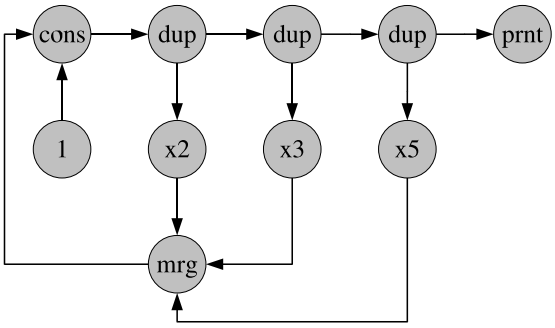

Figure 12: An unbounded process network that computes the sequence of integers of the form $2^k3^m5^n$. The `Scale` process multiplies each element of a stream by a constant. The `Merge` process performs an ordered merge, eliminating duplicates.

creates data at a faster rate than the `Worker` process consumes it, data accumulates in the channel's buffer. Because the Java run-time system makes no guarantees about fairness in the scheduling of threads [5], the `Worker` and `Consumer` processes might never get an opportunity to run if channels had unlimited capacity.

Imposing a bound on a channel's capacity can ensure fairness in process scheduling. If a producing process blocks when it eventually attempts to write to a full channel, the corresponding consuming process is given an opportunity to run. Thus, even if the Java run-time system does not provide time-slicing among threads, a degree of fairness is enforced through limited buffer capacity and blocking writes.

Using limited-capacity channels with blocking writes may introduce deadlock. Consider the example in Figure 12, considered by Kahn [11] and which Dijkstra attributes to Hamming [7]. This process network produces, in order, the sequence of integers whose prime factors are 2, 3, and 5. The first few elements of this sequence are 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20. Observe that if an integer x is an element of the sequence, then so too are $2x$, $3x$, and $5x$. The sequence is computed by injecting the initial element 1; recursively forming copies of the sequence multiplied by 2, 3, and 5; and then performing an ordered merge of those three streams, eliminating duplicates. Because each new element produced by the merge process results in 1, 2, or 3 new elements accumulating in the channels, the amount of storage required for the channels grows without bound as the program executes. If channel capacities are limited, deadlock will eventually occur when a cycle of processes are blocked attempting to write to full channels.

Using limited-capacity channels with blocking writes may introduce deadlock, even in program graphs with no

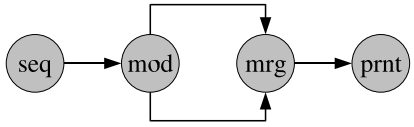

Figure 13: A directed, acyclic program graph that may experience deadlock if channel capacities are limited. The `source` process produces a sequence of consecutive integers. The `mod` process sends all values that are evenly divisible by some constant N to its upper output and all other values to its lower output. The `merge` process performs an ordered merge of its inputs. The `print` process prints the values read from its input.

directed cycles. Consider the example in Figure 13. For every N data elements read, the `Modulo` process produces 1 element on its first output and $N-1$ elements on its second output. If the capacity of this second channel between the `Modulo` and `Merge` processes is too small, then the `Modulo` process blocks writing to a full channel before it can produce all $N-1$ data elements. Unfortunately, the `Merge` process cannot consume data from this channel because it is blocked reading from its other input, which is empty. In order to break this deadlock, the buffer capacity of the channel must be increased.

The default buffer capacities for Java streams are sufficient for many programs, including all programs with no *undirected* cycles, and we can select arbitrarily large buffer sizes at the time that the program graph is constructed. However, determining the buffer capacities needed to avoid deadlock caused by blocking writes is undecidable (it is equivalent to the halting problem) so a procedure is needed to manage buffer sizes as the program graph executes. We have such a procedure [13] but need to extend it for distributed computing using a distributed deadlock detection algorithm.

With the implementation described thus far, a process network can run on a single computer. Because there is one thread per process, we can take advantage of multiple processors when they are available. In the next section, we describe how our implementation supports distributed computing, using several computers connected by a network.

4 Distributed Process Networks

In our distributed implementation of process networks, we envision having a collection of servers at our disposal that can perform computations on our behalf. These servers could be part of a local cluster, or they could be dispersed across the Internet. A process network program graph

can be created on one computer, and then some of the `Process` objects can be sent to other servers where they will be executed. If a `CompositeProcess` is sent to a server, then that server could decompose it and redistribute some or all of the component `Process` objects to other available servers. It is our intention that this distribution and redistribution of `Process` objects take place both before and during execution of the program graph.

4.1 Compute Servers

To support distributed computing, we have implemented a generic compute server that is accessible via Remote Method Invocation (RMI). The entire implementation can be contained in a single jar file that is less than 8K bytes in size, making it easy to install on a new host. Entries for each compute server in the RMI registry make it easy for client applications to locate remote compute servers.

The `Server` interface defines two methods that can be invoked remotely:

```
 void run(Runnable target) throws IOException;
Object run(Task target)    throws IOException, ClassNotFoundException;
```

Any object implementing the `Runnable` or `Task` interface passed as an argument to the `run` method is serialized and sent to the remote compute server for execution. In the case of `Runnable` objects, the server's `run` method returns immediately. In the case of `Task` objects, the server's `run` method waits for the execution of the `Task` to complete so that it can return the result.

Specifying an appropriate value for the `java.rmi.server.codebase` property when invoking the client application allows the compute server to dynamically download the class definition for the `Runnable` or `Task` object passed as a parameter to its `run` method. Thus, rather than having to install the client application on all of the computers that host compute servers (and having to install updates whenever the application is modified) one can make the application available for download from a single web server.

4.2 Automatic Connection Establishment

When processes are located on different compute servers, the `LocalOutputStream` and `LocalInputStream` objects shown in Figure 3, which provide local communication through buffers in memory, must be replaced with objects that provide remote communication between servers using network sockets. Rather than burdening the programmer with the responsibility of performing this replacement, we have made this chore completely invisible and automatic by taking advantage of several features of Java Object Serialization.

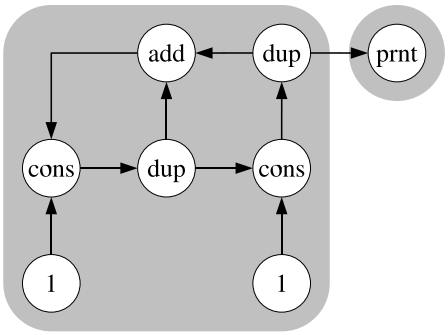

Figure 14: A process network that computes Fibonacci numbers partitioned for execution on two servers.

As Java objects are serialized, any objects to which they have references are also serialized. Thus, when a `Process` is serialized and sent to a remote compute server, the `ChannelInputStream` and `ChannelOutputStream` objects to which it refers (and the underlying objects to which they refer) are also serialized. We have implemented `writeObject`, `readObject`, `writeReplace` and `readResolve` methods for our input and output stream classes so that as they are serialized they perform actions to automatically establish network connections between servers to maintain the channels in the program graph.

Consider what happens when the example from Figure 2 is partitioned into two `CompositeProcess` objects as shown in Figure 14. Both composite processes are created on one computer (server A) and then the composite process containing `Print` is serialized and sent to a remote compute server B. The `ChannelInputStream` for the `Print` process, which initially has an underlying `LocalInputStream` as shown in Figure 3, is serialized along with the `Print` process.

As the `LocalInputStream` is being serialized, it causes the corresponding `LocalOutputStream` used by the `Duplicate` process to be replaced with a `RemoteOutputStream`, which listens for incoming network connections. When the `LocalInputStream` arrives at server B and is deserialized, it replaces itself with a `RemoteInputStream` that establishes a network connection back to the waiting `RemoteOutputStream` on server A, thus maintaining the communication path of the channel. A similar sequence of events takes place when a `LocalOutputStream` is serialized.

4.3 Decentralized Communication

Next consider what happens when the program graph is partitioned into three `CompositeProcess` objects

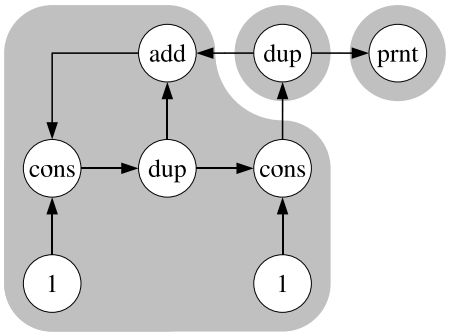

Figure 15: A process network that computes Fibonacci numbers partitioned for execution on three servers.

as shown in Figure 15. All of the composite processes are created on server A, and then the composite process containing `Print` is sent to server B as before. This causes a network connection to be established between servers A and B to maintain the channel between `Duplicate` and `Print`. The composite process containing `Duplicate` is then serialized and sent to server C. The `ChannelInputStream` for the `Duplicate` process and the `ChannelOutputStream` leading from `Duplicate` to `Add`, which have underlying `LocalInputStream` and `LocalOutputStream` objects, are handled as before. The remaining `ChannelOutputStream` for the `Duplicate` process, which at this point has an underlying `RemoteOutputStream` connected to a `RemoteInputStream` associated with the `Print` process on server B must be handled differently. We wish to avoid having all communication between the processes on servers B and C pass through server A. Instead we prefer a direct connection between servers B and C.

As the `RemoteOutputStream` associated with the `Duplicate` process is being serialized, it creates a `RedirectedInputStream`, which is sent to server B to replace the `RemoteInputStream` there. When the `RedirectedInputStream` arrives at server B and is deserialized, it listens for incoming network connections. When the `RemoteOutputStream` associated with `Dupulicate` arrives at server C and is deserialized, it establishes a network connection to the waiting `RedirectedInputStream` on server B. A direct connection now exists between the output of `Duplicate` on server C and the input of `Print` on server B. After this point, no further communication with server A is required to maintain this channel.

5 Embarrasingly Parallel Computing

Many computations can be broken down into a collection of independent tasks that can be carried out in parallel. Such computations are sometimes referred to as "embarrassingly parallel." For example, an image can be divided into 16x16 blocks of pixels that are compressed independently with the results collected and written in order to an image file. In this example, a `Producer` breaks the image down into blocks of pixels, one or more `Worker`s compress each block, and a `Consumer` writes each compressed block to an image file. Other embarrassingly parallel problems include the search for intelligent life on other planets[2][3], the search for large prime numbers[3], and the study of protein folding[4].

Figure 1 shows one way of organizing such an embarrassingly parallel computation. The Producer, worker, and Consumer are arranged in a pipeline so that some speedup can be achieved when each process is executed on a separate computer. The resulting speedup is limited by the degree to which the three stages of the pipeline are balanced. Unfortunately, the computation is typically dominated by the worker with the producer and consumer frequently being idle waiting to send tasks to or collect results from the worker.

Figure 16 shows one way to replace a single worker with several workers executing in parallel. A `Scatter` process takes N tasks from the producer and distributes one to each of N workers. A `Gather` process collects one result from each worker and sends these N results to the consumer. Because the gather process collects results in the same order in which tasks are sent to the workers by the scatter process, the parallel composition in Figure 16 is, from the point of view of the producer and consumer processes, equivalent to a single worker. Identical results are presented to the consumer in the same order as the single-worker computation in Figure 1.

In this arrangement of parallel workers, the same number of tasks is sent to each worker. Such static load balancing works well in a homogeneous computing environment where the same amount of work is required by each task and the computers on which the worker processes execute all have the same available computing power. However, in a heterogeneous computing environment the speedup attainable with this arrangement will be limited by the rate at which the slowest worker can execute tasks.

Figure 17 shows another way to replace a single worker with several workers executing in parallel. The scatter and gather processes in Figure 16 are replaced, respectively,

[2] http://setiathome.berkeley.edu/
[3] http://mersenne.org/
[4] http://foldingathome.stanford.edu/

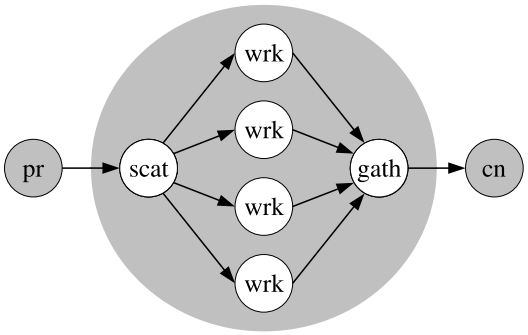

Figure 16: Parallel `Workers` (w) with static load balancing. The `Scatter` (s) and `Gather` (g) processes cause equal numbers of tasks to be distributed to and collected from the `Workers`.

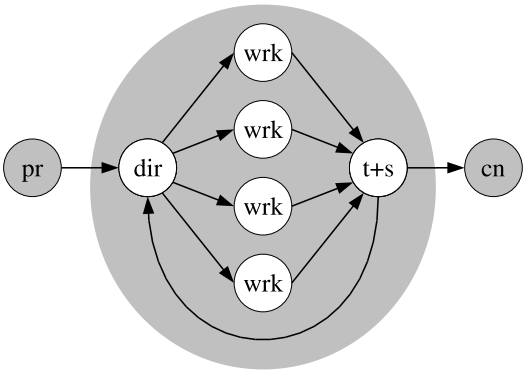

Figure 17: Parallel `Workers` (w) with dynamic, on-demand load balancing. The `Direct` (d) and indexed merge (t+s) processes cause a new task to be distributed to a `Worker` for every result collected from that `Worker`. The `Turnstile` (t) and `Select` (s) processes within the composite indexed merge process (t+s) are shown in Figure 18.

by a `Direct` process and an indexed merge. The indexed merge is composed of a `Turnstile` process and a `Select` process as shown in Figure 18. This indexed merge collects results from the workers as they become available and creates an index stream indicating the order in which the results are sent to the consumer. This index stream then controls the order in which new tasks are distributed to the workers by the `Direct` process.

The `Turnstile` introduces a small degree of nondeterminism in that the sequence of values produced on the index stream is not completely determined by the program specification and depends in part on the ordering of events in the execution environment. However, because `Direct` and `Select` share this index stream, results are collected from the workers in the same order in which tasks are sent to the workers *independent of the ordering of index values*. Thus the order in which results are sent to the consumer by the dynamically balanced parallel composition in Figure 17 is identical to that for the statically balanced composition in Figure 16 and the pipelined computation in Figure 1. Despite the non-determinism introduced by the `Turnstile` process, the final results of the computation are determinate. The `MetaDynamic` schema is "well behaved" [6, 8], having an input-output relation that is independent of the values on the index stream.

In this arrangement, each worker is sent a new task for every task it completes. With this on-demand load balancing, faster workers process more tasks, slower workers process fewer tasks, and all computing resources are fully utilized. Such dynamic load balancing works well not only in homogeneous computing environments, but also in heterogeneous environments where the amount of work required by each task may not be uniform and the computers on which the worker processes execute may have different available computing power and different competing workloads.

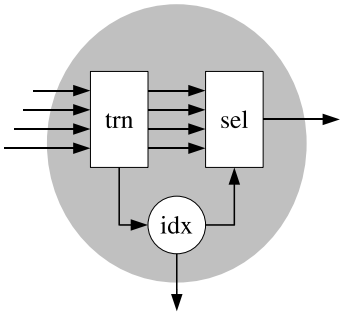

Figure 18: An indexed merge composed of a `Turnstile` and a `Select`. The `Turnstile` passes results through to the `Select` in the order that they become available (which can depend on the execution speeds of the different `Workers`) and produces an index stream indicating that order. This index stream is made available both to the `Select` and to the `Direct` that is shown in Figure 17 so that results are collected from workers in the same order in which tasks are distributed. An initial sequence is inserted into the stream of index values by (n) to control the distribution of the first set of tasks to the workers.

5.1 Generic Computing with Active Objects

To facilitate the creation of new "embarrassingly parallel" applications, we have implemented generic `Producer`, `Worker`, and `Consumer` processes that operate on `Task`s. The `Task` interface defines one method:

```
Object run();
```

Our generic `Producer` repeatedly invokes `run` on a single `Task` object supplied as a constructor argument and writes the results (which are `Task`s) to its output channel. Our generic `Worker` repeatedly reads a `Task` from its input channel, runs it, and then writes the result (which is itself a `Task`) to its output channel. Our generic `Consumer` repeatedly reads a `Task` from its input channel, runs it, and discards the result. Thus the computation to be carried out on the data is defined not in the processes, but in the objects containing the data itself.

The creation of a new application simply requires the implementation of application-specific producer, worker, and consumer `Task`s. The result returned by the `run` method of a producer `Task` is a worker `Task`, and the result returned by the `run` method of a worker `Task` is a consumer `Task`. The `Producer`, `Worker`, and `Consumer` processes and the `MetaStatic` and `MetaDynamic` composite processes are compmletely generic and can be applied to new applications by defining new producer, worker, and consumer `Task`s.

5.2 Parallel Factorization

RSA public keys are created using the product of two large prime numbers, $N = P \times Q$. The security of the cryptosystem depends on the difficulty of factoring large numbers, so that knowledge of N does not reveal P or Q. A "weak" key would be one for which the difference between P and Q is relatively small. A brute-force approach for finding such "weak" keys searches for a value of P such that $N = P \times (P + D)$ for small differences D.

This problem can be broken down into a set of independent tasks to be executed in parallel. The `ProducerTask` breaks up the search space into many small ranges of differences and each `WorkerTask` searches for P, a factor of N, for each difference within its range. The `ConsumerTask` collects the results from the `WorkerTasks` and, if a factor is found, prints the result and stops.

An experimental test case was created using a 512-bit randomly selected prime number P to which a small difference D was added. This resulted in a 1024-bit value of N with factors P and $P + D$. The value of D was chosen so that the factor P would be found after executing 2048 worker tasks. The amount of work required for each task, and thus the total amount of work required to factor N, was controlled by adjusting the range of differences to be tested by each task. In all of our experiments, each task tested 32 even values of D. The batch size of 32 struck a balance between computation and communication that prevented the producer and consumer tasks from creating bottlenecks that could have limited available parallelism.

	Time	Speed	CPU Class
A	11.63	1.93	2.4 GHz Pentium 4
B	13.13	1.71	2.2 GHz Pentium 4
C	22.50	1.00	1.0 GHz Pentium III
D	22.78	0.99	2 × 1.0 GHz Pentium III
E	28.14	0.80	8 × 700 MHz Pentium III Xeon

Table 1: Sequential Execution. Time is given in minutes and speed is normalized with respect to a 1 GHz Pentium III.

We had at our disposal several computers of different vintages connected by 100 Mb/s switched ethernet. Many of these computers are part of a computing lab with a 3-year replacement schedule in which the oldest third of the computers are replaced each year. Some of the computers had a single CPU, some had two, and one computer had eight CPUs. The five different classes of CPUs used to execute worker tasks are shown in Table 1. A total of 25 computers with 34 CPUs were used in our experiments: 1 in class A, 6 in class B, 15 in class C, 2 in class D, and 1 in class E.

We first ran a strictly sequential implementation of the factoring problem on a representative computer from each class. The computation was carried out by directly invoking the `run` methods of the producer, worker, and consumer tasks without the use of process networks. The timing results are shown in Table 1. All execution times are reported in minutes with speeds normalized to a 1 GHz Pentium III. We have reported the CPU time used rather than total elapsed time in order to minimize effects on our measurements of other background processes executing on the computers.

We then constructed process networks for parallel execution using static load balancing, as shown in Figure 16, and dynamic load balancing, as shown in Figure 17. The factoring problem was executed in parallel using from 1 up to 32 CPUs configured to execute the workers. The elapsed time and speedup achieved are summarized in Table 2, Figure 19, and Figure 20. ¡/p¿

The ideal times and speeds are calculated using the timing results from Table 1. The speed is simply the sum of the speeds for all of the CPUs in use for a particular run and the time is scaled from the execution time for a class C CPU using this computed ideal speed. CPUs in the fastest categories, classes A and B, are used first and CPUs from slower categories, classes C through E, are used as ad-

Work-	Ideal		Static		Dynamic	
ers	Time	Speed	Time	Speed	Time	Speed
1	11.63	1.93	12.15	1.85	12.39	1.82
2	6.17	3.65	6.93	3.25	6.57	3.43
4	3.18	7.08	3.55	6.34	3.44	6.54
8	1.70	13.22	3.03	7.42	1.87	12.02
16	1.06	21.22	1.63	13.80	1.20	18.73
32	0.63	35.97	1.00	22.42	0.76	29.77

Table 2: Parallel Execution. Elapsed time is given in minutes and speed is normalized with respect to a 1 GHz Pentium III.

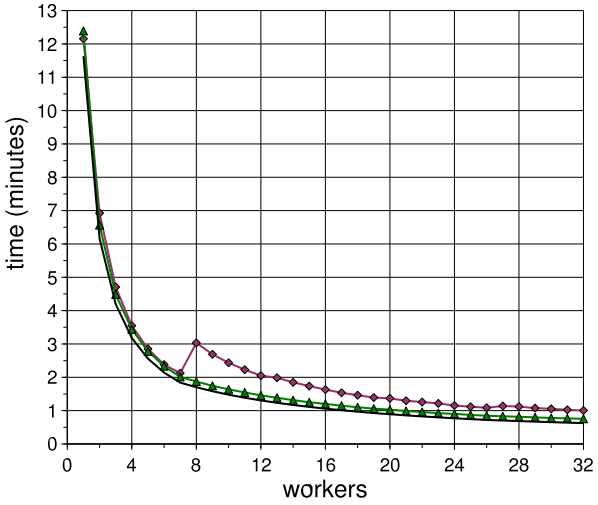

Figure 19: Elapsed time for static load balancing (diamonds) and dynamic load balancing (triangles) compared to a theoretical ideal (line).

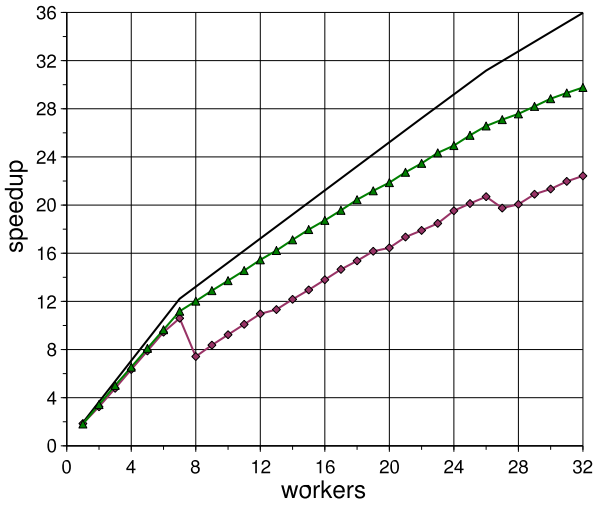

Figure 20: Speedup for static load balancing (diamonds) and dynamic load balancing (triangles) compared to a theoretical ideal (line). The scale on the vertical axis is relative to the speed of a 1 GHz Pentium III.

ditional workers are needed. Two inflection points are evident in the ideal speed curve in Figure 20. The first occurs when the number of workers increases from 7 to 8, corresponding to the addition of the first CPU from class C, which is significantly slower than CPUs in classes A and B. The second, less obvious inflection point occurs when the number of workers increases from 26 to 27, corresponding to the addition of the first CPU from class E, which is the slowest class.

The disadvantages of static load balancing in a heterogeneous computing environment become evident in Figures 19 and 20. Because the same number of tasks are sent to each worker, the speedup attainable is limited by the slowest worker. When the first CPU from class C is added to the computation, the elapsed time actually *increases* and the speedup *decreases*. All workers are forced to process tasks in lock-step with the slowest worker and end up spending a significant fraction of the time idle. In this example, parallel execution with static load balancing fails to take full advantage of available computing resources.

Employing dynamic, on-demand load balancing overcomes the inefficiencies of static load balancing. Because the next task is sent to the worker that most recently completed a task, faster workers end up processing more tasks and slow workers do not interfere with fast workers. Each worker is kept busy and is never idle. Parallel execution with dynamic load balancing fully utilizes available computing resources.

Despite its efficient use of resources, parallel execution with dynamic load balancing fails to achieve the pre-

dicted ideal speedup. The explanation for this shortcoming is Amdahl's Law. Parts of the computation, such as constructing the process network and distributing worker processes to compute servers, cannot be executed in parallel. In fact, this startup overhead increases as the number of workers increases and accounts for virtually the entire difference between the ideal case and the dynamically load balanced case. Additional minor sources of overhead include Object Serialization and network communication associated with the channels in the process network. The times reported in the first row of Table 2 show that this additional overhead is no more than 6% to 7% for this example, which is a small price to pay for the ability to execute processes in parallel.

6 Conclusion and Future Work

We have described our implementation of process networks in Java which supports computing platforms ranging from a single computer to clusters of multiprocessor servers to a geographically distributed collection of servers. Our implementation of first-in first-out channels enforces blocking reads, even while parts of the program graph are in the midst of being redistributed to other servers. This is necessary to ensure that computations are determinate: the program will produce correct results independent of the timing details of the execution, which may be beyond our control in a distributed environment.

Each process executes in a separate Java thread and multiple threads are used for composite processes to avoid introducing deadlock through composition. Multithreaded execution also makes it possible to take advantage of the parallelism available with multiprocessor servers and to overlap communication with computation to help hide communication latencies.

Our implementation works well with program graphs that are designed to be non-terminating, such as signal processing applications with indefinitely long streams of input samples. It works equally well with terminating program graphs through a graceful system of cascading exceptions.

6.1 Load Balancing

We can currently construct a process network program graph on one server, then distribute parts of that graph to several servers before execution begins. All network connections necessary for communication between processes are automatically established when objects are serialized and deserialized as they are being distributed to remote servers. One focus of our future work is making it possible to re-distribute processes after execution has already begun with the possibility that processes will be moved more than once. This is important with applications that have self-reconfiguring graphs. As more processes are created, it becomes necessary to enlist more servers to partake in the computation and to have processes migrate from one server to another for load balancing. Load balancing is also important when using a collection of heterogeneous servers with a wide range of processing speeds.

6.2 Buffer Management

Another problem to be addressed is that of distributed deadlock detection. As discussed earlier, the use of limited-capacity channels can introduce deadlock. If deadlock occurs, it is first necessary to detect it. It is then necessary to determine whether increasing buffer capacities on the channels will relieve the deadlock. One method of buffer management that we have used in the past is described in [13]. We plan to apply those ideas to our distributed Java implementation and then investigate possible improvements.

One reason for choosing Java for this implementation was its portable nature. A compiled class file can be executed on a variety of platforms, and the code is compact so that it can easily be distributed at the time that computation is initiated. In our current implementation, the compiled class files for the application must be available on the local file system of each server. In our laboratory we have a cluster of servers that have a shared network file system, so that no effort is required on our part to distribute the code to each server. However, as we experiment with more geographically dispersed servers the issue of transporting code becomes an issue. Just as the network connections for the channels are established automatically, so too the executable code should be distributed automatically.

Java Object Serialization makes provisions for attaching annotations to class descriptions in the serialization data stream. This annotation could take the form of a URL (Universal Resource Locator) as with RMI (Java Remote Method Invocation) so that class files can be retrieved from a web server on demand. Another alternative is to include the Java bytecode directly in the class annotation. This has the advantage that the distribution of code is now just as scalable as the distribution of data. There is no central repository for the code that forms a bottle neck. We have already had some initial success with embedding the bytecode in the data stream.

The examples presented here are simple, but they serve to demonstrate the capabilities of our Java implementation of distributed process networks. Work has begun on the implementation of a parallel algorithm for factoring large numbers, which has practical applications to cryptography, using both our implementation of process networks

and a Java implementation of CSP (Communicating Sequential Processes). We will use this and other example applications to evaluate the performance of our implementation with several different configurations of servers, including a multiprocessor and a cluster of several servers.

References

[1] Gregory E. Allen, Brian L. Evans, and David C. Schanbacher. Real-time sonar beamforming on a Unix workstation using process networks and POSIX threads. In *Asilomar Conference on Signals, Systems, and Computers*, Monterey, California, November 1998. http://www.ece.utexas.edu/~allen/Asilomar98/.

[2] Abdelkader Amar, Pierre Boulet, and Jean-Luc Dekeyser. Towards distributed process networks with CORBA. Technical Report LIFL 2002-04, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille, May 2002. http://www.lifl.fr/west/publi/AmBoDe.pdf.

[3] David Pl Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002. http://setiathome.berkeley.edu/cacm/cacm.html.

[4] Twan Basten and Jan Hoogerbrugge. Efficient execution of process networks. In Alan Chalmers, Majid Mirmehdi, and Hank Muller, editors, *Communicating Process Architectures*, Bristol, U.K., September 2001. http://www.ics.ele.tue.nl/~tbasten/abstracts/eepn.html.

[5] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial*. Addison-Wesley, 3rd edition, 2000. http://java.sun.com/docs/books/tutorial/.

[6] J. B. Dennis, J. B. Fosseen, and J. P. Linderman. Data flow schemas. In *International Symposium on Theoretical Programming*, number 5 in LNCS, pages 187–215, Berlin, 1972. Springer-Verlag.

[7] Edsger W. Dijkstra. *A Discipline of Programming*. Printice-Hall, Englewood Cliffs, New Jersey, 1976.

[8] G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved dataflow programs for DSP communication. In *ICASSP: International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 561–564, San Francisco, March 1992.

[9] Mudit Goel. Process networks in Ptolemy II. Master's thesis, University of California, Berkeley, December 1998. http://ptolemy.eecs.berkeley.edu/publications/papers/98/PNinPtolemyII/.

[10] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of IFIP Congress*, pages 471–475, Stockholm, August 1974.

[11] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In Bruce Gilchrist, editor, *Information Processing 77: Proceedings of IFIP Congress*, pages 993–998, Toronto, August 1977.

[12] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995. http://ptolemy.eecs.berkeley.edu/papers/processNets.

[13] Thomas M. Parks. *Bounded Scheduling of Process Networks*. Technical report UCB/ERL-95-105, PhD dissertation, EECS Department, University of California, Berkeley CA 94720, December 1995. http://ptolemy.eecs.berkeley.edu/papers/parksThesis.

[14] Richard S. Stevens, Marlene Wan, Peggy Laramie, Thomas M. Parks, and Edward A. Lee. Implementation of process networks in Java. http://www.ait.nrl.navy.mil/pgmt/PNpaper.pdf, July 1997.

[15] Darren Webb, Andrew Wendelborn, and Kevin Maciunas. Process networks as a high-level notation for metacomputing. In *Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, April 1999. http://ipdps.eece.unm.edu/1999/java/webb.pdf.

[16] Darren L. Webb, Andrew L. Wendelborn, and Julien Vayssière. A study of computational reconfiguration in a process network. In *IDEA*, Victor Harbour, South Australia, February 2000. http://www.cs.adelaide.edu.au/users/idea/idea7/PDFs/webb.pdf.