Implementing Dataflow With Threads

Leslie Lamport
Microsoft Research

23 November 2005
Modified 12 Dec 2006

MSR-TR-2006-181

Abstract

Dataflow computations are represented by a class of marked graphs called process marked graphs. A general algorithm is presented for implementing process marked graph synchronization with multiple threads that communicate using only reads and writes of shared memory.

Contents

1 Introduction **1**

2 Process Marked Graphs in Pictures **1**

3 Implementing A Process Marked Graph **7**
3.1 Preliminaries 8
3.1.1 Notation 8
3.1.2 Some Facts About Modular Arithmetic 8
3.2 Marked Graphs in Mathematics 9
3.3 Implementing a Process Marked Graph 11
3.3.1 A Simple Algorithm 11
3.3.2 An Algorithm With Unbounded Counters 12
3.3.3 Bounding the Counters 14
3.3.4 A Fine-Grained Algorithm 15
3.3.5 An Optimization 19
3.4 The Producer/Consumer System 20
3.5 Barrier Synchronization 21
3.6 Implementing the Algorithm 26
3.7 Caching Behavior 30

4 Keeping Track of Buffers **31**

5 Conclusion **33**

Acknowledgements **35**

References **35**

Appendix: Formal Definitions **36**

1 Introduction

A dataflow computation is performed by a set of computing elements that send one another data values in messages. Each computing element receives input values from other computing elements and uses them to compute output values that it then sends to other computing elements. When a dataflow computation is implemented with a shared-memory multiprocessor, data values are stored in buffers; processors act as computing elements, informing one another when a buffer used by one computation is available for use by another computation.

We describe multiprocessor dataflow computations by a type of marked graph [2] called a process marked graph. (Marked graphs are a restricted class of Petri net [13].) We describe a shared-memory multiprocess implementation of any process marked graph using only reads and writes. It yields a method for implementing an arbitrary multiprocessor dataflow computation by multiple threads that use only reads and writes for synchronization. The implementation can be chosen to have optimal caching performance.

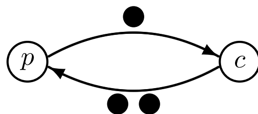
We begin in Section 2 by informally describing process marked graphs and how they are used to represent dataflow computations. Section 3 describes these graphs more precisely and develops the algorithm for using them to synchronize dataflow computations. The algorithm is applied to some simple examples, and practical aspects of its implementation are discussed. Section 4 explains how a computation determines which buffers contain its data. The conclusion summarizes related prior work and describes some performance tests. The tests indicate that our algorithm should perform well. Actual performance will depend very much on the details of the machine architecture, and there seems to be little point in extensive testing on any particular current multiprocessor—especially since the algorithm will probably be of most use in the coming generation of multi-core machines. The appendix provides formal definitions of the concepts and operators introduced in Section 3.

2 Process Marked Graphs in Pictures

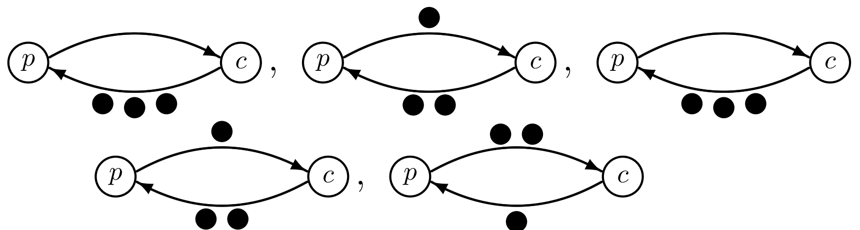
A *marked graph* consists of a nonempty directed graph and a placement of tokens on its edges, called a *marking*. Here is a simple marked graph.

(1)

A node in a marked graph is said to be *fireable* iff there is at least one token on each of its in-edges. Node p is the only fireable node in marked graph (1). Firing a fireable node n in a marked graph changes the marking by removing one token from each in-edge of n and adding one token to each of its out-edges. *Firing* node p in marked graph (1) produces



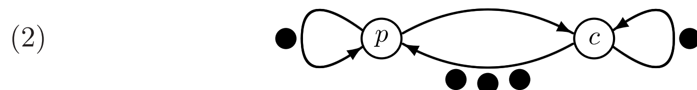
An *execution* of a marked graph consists of a sequence of marked graphs, each obtained from the previous one by firing a fireable node. For example, here is one possible 5-step execution of (1).



The marked graph (1) is a conventional representation of producer/consumer or bounded buffer synchronization with three buffers [3]. Node p represents the producer, node c represents the consumer, and the three tokens represent the three buffers. A buffer is empty if its token is on edge $\langle c, p \rangle$; it is full if its token is on edge $\langle p, c \rangle$. Firing node p describes the producer filling an empty buffer; firing node c represents the consumer emptying a full buffer.

We now modify this way of representing the producer/consumer problem. First, we add edges with tokens that represent the processes to obtain

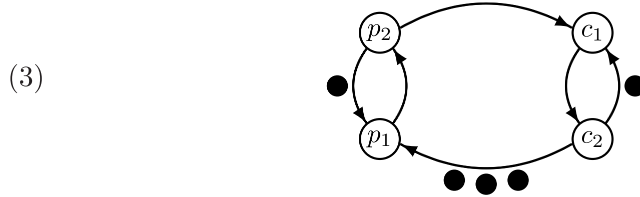
(2)



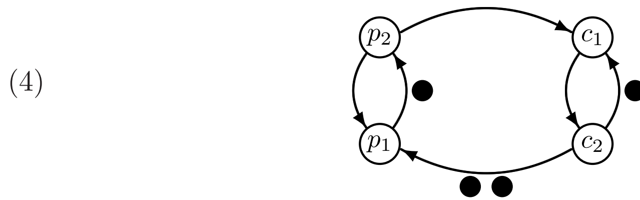
The token on edge $\langle p, p \rangle$ represents the producer process; the one on edge $\langle c, c \rangle$ represents the consumer process. As in (1), the producing and consuming operations are represented by the actions of firing nodes p and c, respectively.

Firing a node is an atomic step. We want to represent the operations of filling and emptying a buffer as having a finite duration. We therefore

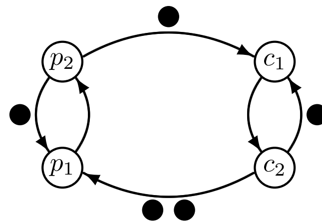
modify (2) by expanding nodes p and c into two-node subgraphs to obtain

(3) 

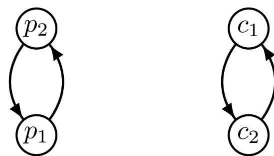
Firing node p_1 in the initial marking produces

(4) 

The token on edge $\langle p_1, p_2 \rangle$ indicates that the producer is performing the operation of filling the first buffer. Firing node p_2 in marking (4) ends that operation, producing

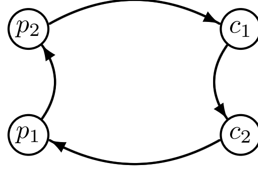


In marked graph (3), the producer and consumer processes are represented by the tokens on the two subgraphs



A token on edge $\langle p_1, p_2 \rangle$ represents the producer performing the operation of filling a buffer. A token on edge $\langle p_2, p_1 \rangle$ represents the producer waiting to fill the next buffer. Similarly, a token on $\langle c_1, c_2 \rangle$ represents the consumer emptying a buffer, and a token on $\langle c_2, c_1 \rangle$ represents it waiting to empty the next buffer. We call $\langle p_1, p_2 \rangle$ and $\langle c_1, c_2 \rangle$ *computation* edges; a token on one of those edges represents a process performing a computation on a buffer.

The tokens on the subgraph

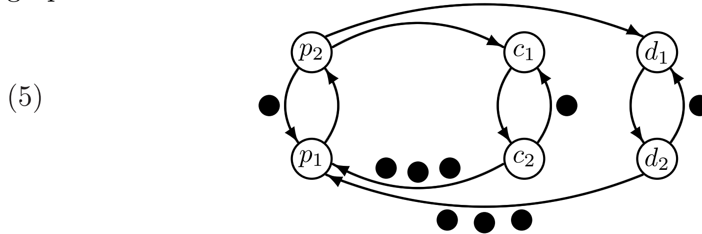


represent the pool of buffers. A token on $\langle c_2, p_1 \rangle$ represents an empty buffer; a token on $\langle p_1, p_2 \rangle$ represents one being filled; a token on $\langle p_2, c_1 \rangle$ represents a full buffer; and a token on $\langle c_1, c_2 \rangle$ represents one being emptied. A token on edge $\langle p_1, p_2 \rangle$ represents both the producer process and the buffer it is filling; a token on $\langle c_1, c_2 \rangle$ represents both the consumer and the buffer it is emptying.

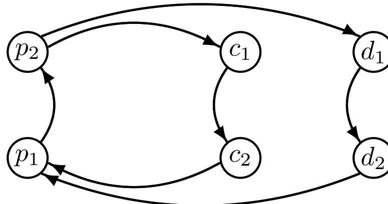
In general, a *process marked graph* is a marked graph containing disjoint cycles called *processes*, each node of the graph belonging to one process, and whose marking places a single token on each process. (This is the same definition as the one in [9], except we associate only processes and not their operations with the nodes.) Certain process edges are called *computation* edges, but they are irrelevant to our algorithms for executing process marked graphs.

As another example of a process marked graph, we expand the producer/consumer system by adding another consumer. The producer writes into a buffer, and the contents of that buffer are read, perhaps concurrently, by the two consumers. This system is represented by the following marked graph.

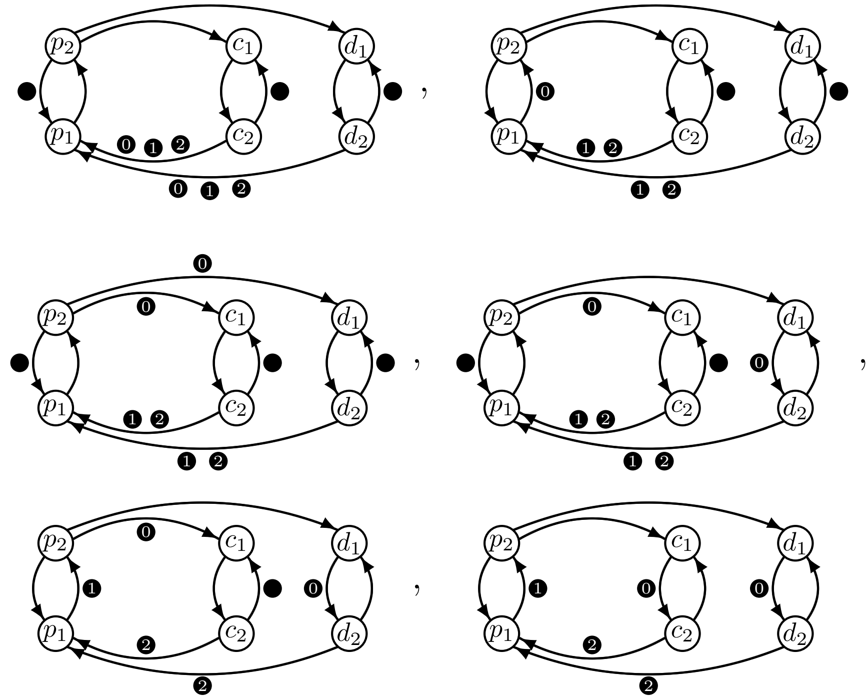
(5)



The additional consumer processes is represented by the subgraph containing nodes d_1 and d_2 and the edges joining them. Edge $\langle d_1, d_2 \rangle$ is an additional computation edge. The buffer pool is represented by the subgraph



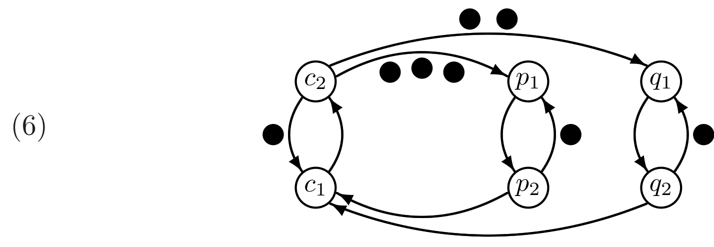
Here is a five-step execution of this marked graph, where we have labeled each token with the number of the buffer that it represents.



The last state is one in which the producer is writing into buffer 1 and both consumers are reading from buffer 0.

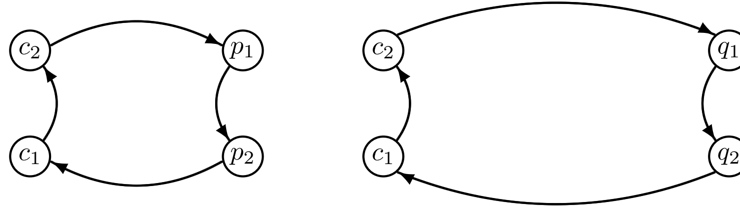
Our next example is a system with two producers and one consumer. The two producers write into separate pools of buffers, one buffer pool containing three buffers the other containing two. The consumer's computation reads one buffer written by each producer.

(6)



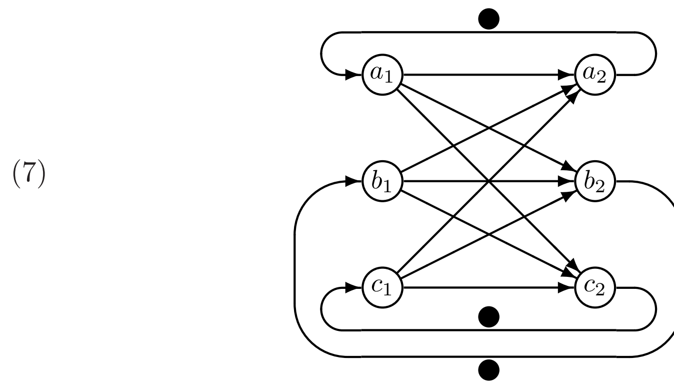
The underlying graph is isomorphic to that of (6), and it has the analogous three processes. There are two buffer pools, represented by these two

subgraphs:

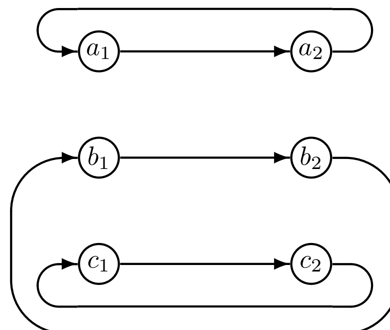


Our final example is barrier synchronization. In barrier synchronization, a set of processes repeatedly execute a computation such that, for each $i \geq 1$, every process must complete its ith execution before any process begins its $(i+1)^{\text{st}}$ execution. For three processes, barrier synchronization is described by the following process marked graph

(7)



where the processes are the three cycles



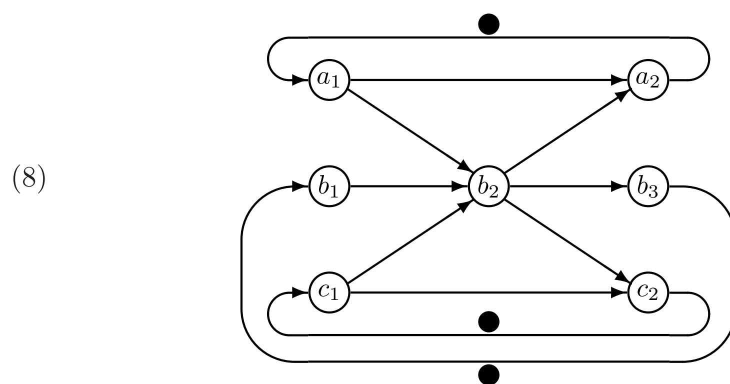
Edges $\langle a_2, a_1 \rangle$, $\langle b_2, b_1 \rangle$, $\langle c_2, c_1 \rangle$, are computation edges; a token on any of these edges represent the process performing its computation.[1] There are

[1] It would be more natural to describe barrier synchronization with a marked graph

no buffers in this problem. The 6 edges not belonging to a process effect barrier synchronization by ensuring that none of the nodes a_2, b_2, and c_2 are fireable for the $(i+1)^{\text{st}}$ time until all three nodes a_1, b_1 and c_1 have fired i times.

When modeling a system by a process marked graph, we are interested only in tokens on computation edges. Two process marked graphs whose executions place and remove tokens on these edges in the same sequence are equivalent models. For example, here is another way to represent barrier synchronization for three processes.

(8)



It guarantees barrier synchronization because none of the nodes a_2, b_3, and c_2 are fireable for the $(i+1)^{\text{st}}$ time before b_2 has fired i times, which can occur only after all three nodes a_1, b_1, and c_1 have fired i times. Applying our algorithm for implementing a process marked graph to the graphs (7) and (8) yields different barrier synchronization algorithms.

3 Implementing A Process Marked Graph

We now develop our algorithm for implementing a process marked graph. We begin in Section 3.1 by introducing some mathematical notation and stating some simple facts about modular arithmetic needed in our derivation. Section 3.2 provides more rigorous definitions of marked graphs and their executions. Section 3.3 develops our algorithm through a sequence of informal refinements of a very simple algorithm. Sections 3.4 and 3.5 apply the algorithm to the simple producer/consumer system (3), with an arbitrary

---

in which processes are not initially performing their computations, the initial marking instead having tokens on all the non-computation edges. However, it's easier to draw a graph with only three tokens instead of nine.

number of buffers, and to barrier synchronization, with an arbitrary number of processors. Section 3.6 discusses how the algorithm is implemented on modern memories with weak semantics that require additional operations to implement conventional read/write synchronization. Section 3.7 discusses the caching behavior of the algorithm.

3.1 Preliminaries

3.1.1 Notation

We use fairly standard mathematical notation. For example, $\forall\, x \in S : P(x)$ is the formula that is true iff $P(x)$ is true for all x in the set S.

We use square brackets to denote function application, writing $f[x]$ for the value obtained by applying function f to value x. We use the terms *array* and *function* interchangeably, an array indexed by a set S being a function with domain S.

We let $i\,..\,j$ be the set of all integers k such that $i \leq k \leq j$. For any integer a and positive integer b, we define $a \,\%\, b$ to be the integer in $0\,..\,(b-1)$ that is congruent to a modulo b.

We define the finite sequence $\langle s_1, \ldots, s_k \rangle$ to be a function σ with domain $1\,..\,k$ such that $\sigma[i]$ equals s_i for all i in $1\,..\,k$. A pair $\langle a, b \rangle$ is a sequence of length 2.

We define a directed graph Γ to consist of a set $\Gamma.nodes$ of nodes and a set $\Gamma.edges$ of edges, where an edge is a pair of nodes. We say that an edge $\langle m, n \rangle$ is an *in-edge* of node n and an *out-edge* of node m, and that m is the *source* and n the *destination* of $\langle m, n \rangle$. We define a *simple cycle* of Γ to consist of a non-empty finite sequence $\langle p_1, \ldots, p_k \rangle$ of distinct nodes such that $\langle p_i, p_{(i\%k)+1} \rangle$ is an edge for all $i \in 1\,..\,k$. (In particular, $\langle p_k, p_1 \rangle$ is an edge of Γ.)

3.1.2 Some Facts About Modular Arithmetic

We now assert some properties of modular arithmetic that we will need later. We assume that N is fixed positive integer, and we let $\oplus$ and $\ominus$ be addition modulo N. In other words,

$$a \oplus b \;=\; (a+b) \,\%\, N \qquad\qquad a \ominus b \;=\; (a-b) \,\%\, N$$

for any integers a and b.

The first two facts are straightforward properties of the operator %. Mathematicians would express the first by saying that % is a homomorphism from the integers to the integers modulo N.

Fact 1 For any integers a, b,

$$\begin{array}{rcl}(a \oplus b) & = & ((a\%N) \oplus (b\%N)) \\ (a \ominus b) & = & ((a\%N) \ominus (b\%N))\end{array}$$

Fact 2 For any integers a, b, and p, if $0 \leq p < N$ and $a \in b \mathrel{..} (b + p)$, then $a = b + p$ iff $a \,\%\, N = b \oplus p$.

The next two facts involve % and the operator $\lceil \; \rceil^Q$, where $\lceil i \rceil^Q$ is defined to be the smallest multiple of Q that is greater than or equal to i, for any natural number i. In other words, $\lceil i \rceil^Q = Q * \lceil i/Q \rceil$, where $\lceil i/Q \rceil$ is the smallest integer greater than or equal to i/Q.

Fact 3 For any integers a and Q with $0 < Q \leq N$, if Q divides N, then $\lceil a \rceil^Q \,\%\, N = \lceil a\%N \rceil^Q \,\%\, N$.

Fact 4 For any integers a, b, Q, and B with $0 < Q \leq N$ and $0 < B \leq N$, if $B * Q$ divides N, then

$$((\lceil a \rceil^Q / Q) + b) \,\%\, B \;=\; ((\lceil a \,\%\, N \rceil^Q / Q) + b) \,\%\, B$$

We omit the proofs of these facts.

3.2 Marked Graphs in Mathematics

A marked graph is a pair $\langle \Gamma, \mu_0 \rangle$, where Γ is a directed graph and μ_0 is the *initial marking* that assigns to every edge e of Γ a natural number $\mu_0[e]$, called the number of tokens on e. We assume a fixed Γ and μ_0.

A node n is fireable in a marking μ iff $\mu[e] > 0$ for every in-edge e of n. Let $InEdges(n)$ be the set of all in-edges of n. The condition for n to be fireable in μ is then

$$\forall\, e \in InEdges(n) \,:\, \mu[e] > 0$$

Let $Fire(n, \mu)$ be the marking obtained by firing n in marking μ. (The precise definitions of *Fire* and all other operators we use appear in the appendix.)

The algorithm *MGSpec* that defines the execution of $\langle \Gamma, \mu_0 \rangle$ is given in Figure 1. It is written in the +CAL algorithm language [5]. The value of variable μ is the current marking; it is initialized with the value μ_0. Each node is described by a separate process. The **process** statement contains the code for *Node* process *self*, where *self* is in the set $\Gamma.nodes$ of nodes. A

```
--algorithm MGSpec
variable μ = μ_0 ;
process Node ∈ Γ.nodes
  begin lab : while TRUE do
                when ∀ e ∈ InEdges(self) : μ[e] > 0 ;
                μ := Fire(self, μ) ;
              end while
  end process
end algorithm
```

Figure 1: The algorithm describing all possible executions of the marked graph $\langle \Gamma, \mu_0 \rangle$.

single atomic step of a +CAL algorithm consists of an execution of a process from one label to the next. An atomic step of a *Node* process therefore consists of an execution from label *lab* back to *lab*—that is, an execution of the body of the **while** loop. The **when** statement can be executed only when its expression is true; so a step of process *self* can be executed only when node *self* is fireable in marking μ.

A +CAL algorithm is translated into a TLA+ specification that can be executed or model checked with the TLC model checker [10].[2] (Model checking essentially involves checking all possible executions.) In the actual +CAL code, $\forall$ is written `\A`, the symbol $\in$ is written `\in`, and μ, μ_0, and Γ must be replaced by ASCII identifiers. Similar replacements must be made for additional notation that appears in other algorithms given below.

We now make certain observations about marked graphs. The marked graph $\langle \Gamma, \mu_0 \rangle$ is said to be *live* iff all its executions are deadlock-free, meaning that they can be extended indefinitely. A finite marked graph is live iff every cycle contains at least one token. We assume that $\langle \Gamma, \mu_0 \rangle$ is live.

For a marking μ, we can consider $\mu[e]$ to be the length of edge e, thereby defining a distance function on the nodes. Let $\delta_\mu(m, n)$ be the distance from node m to node n. This distance operator δ_μ is non-commutative ($\delta_\mu(m, n)$ need not equal $\delta_\mu(n, m)$), but it satisfies the triangle inequality $\delta_\mu(m, p) \leq \delta_\mu(m, n) + \delta_\mu(n, p)$, for any nodes m, n, and p.

At any point during an execution of $\langle \Gamma, \mu_0 \rangle$, let $\#(n)$ be the number of times that node n has been fired thus far. Let $\langle m, n \rangle$ be an edge of Γ with

[2]TLC can execute only +CAL algorithms that involve finite computations. For example, we have not assumed that the graph Γ is finite, and algorithm *MGSpec* is perfectly well-defined for infinite graphs. However, TLC can execute the algorithm only if Γ is finite.

$m \neq n$, and let π be a path from n to m. Firing m adds a token to $\langle m, n \rangle$ and removes one from π; firing n removes a token from $\langle m, n \rangle$ and adds one to π. This implies:

Observation 1. For any edge $\langle m, n \rangle$ of Γ, throughout any execution of $\langle \Gamma, \mu_0 \rangle$:

(a) $\mu[\langle m, n \rangle] \; = \; \mu_0[\langle m, n \rangle] + \#(m) - \#(n)$

(b) $\#(n) - \#(m) \; \in \; -\delta_{\mu_0}(n, m) \mathinner{.\,.} \mu_0[\langle m, n \rangle]$

3.3 Implementing a Process Marked Graph

The easiest way to implement a marked graph is with message passing. A token on an edge $\langle m, n \rangle$ from process π_1 to a different process π_2 is represented by a message that is sent by π_1 to π_2 when the token is put on the edge; the message is removed by π_2 from its message buffer when the token is removed. By Observation 1(b), a buffer of capacity $\delta_{\mu_0}(n, m) + \mu_0[\langle m, n \rangle]$ can hold all messages representing tokens on $\langle m, n \rangle$. The messages need not contain any data.

Current multiprocessors do not provide message-passing primitives. So, we implement a process marked graph using read and write operations to shared memory.

3.3.1 A Simple Algorithm

Recall that a cycle is a sequence $\langle p_1, \ldots, p_k \rangle$ of distinct nodes such that each pair $\langle p_i,\, p_{(i\%k)+1} \rangle$ is an edge of Γ. Two cycles are disjoint iff they contain disjoint sets of nodes.

A process marked graph is a triple $\langle \Gamma, \mu_0, \Pi \rangle$ where

- $\langle \Gamma, \mu_0 \rangle$ is a marked graph.
- Π is a set of disjoint cycles of Γ called *processes* such that each node of Γ is in exactly one process.
- For each process $\langle p_1, \ldots, p_k \rangle$, the initial marking μ_0 assigns one token to edge $\langle p_k, p_1 \rangle$ and no tokens to the other $k-1$ edges of the process.

In any execution of a process marked graph, each process $\langle p_1, \ldots, p_k \rangle$ has a single token that cycles through its edges. The nodes of the process fire in cyclic order, starting with p_1.

```
--algorithm PMGAlg1
variable μ = μ0 ;
process Proc ∈ Π
  variable i = 1 ;
  begin lab : while TRUE do
                when ∀ e ∈ SInEdges(self[i]) : μ[e] > 0 ;
                μ := Fire(self[i], μ) ;
                i := (i % Len(self)) + 1 ;
              end while
  end process
end algorithm
```

Figure 2: A simple algorithm describing the execution of process marked graph $\langle \Gamma, \mu_0, \Pi \rangle$.

We execute a process marked graph with an algorithm having one process for every process in Π, each algorithm process firing the nodes of the corresponding process in Π. We identify a process π in Π with the algorithm's process that executes π.

Process π can remember in its internal state which edge of the cycle contains a token. It need only examine edges coming from other processes to determine when to fire a node. Let a *synchronizing* edge be an edge of Γ whose endpoints belong to different processes. Let $SInEdges(n)$ be the set of synchronizing in-edges of node n. The algorithm $PMGAlg1$ of Figure 2 then describes the execution of the process marked graph $\langle \Gamma, \mu_0, \Pi \rangle$. (The variable i is local, with a separate copy for each process.) It should be clear that algorithms $PMGAlg1$ and $MGSpec$ are equivalent in the sense that any execution of the marked graph $\langle \Gamma, \mu_0 \rangle$ allowed by one is allowed by the other. More precisely, for each behavior of one, there is a behavior of the other with the same sequence of changes to μ.

3.3.2 An Algorithm With Unbounded Counters

We now introduce a set *Ctrs* of counters and two arrays, *CtrOf* of counters and an array *Incr* of natural numbers, both indexed by the set of nodes. Every time node n is fired, counter $CtrOf[n]$ is incremented by $Incr[n]$. (If $Incr[n] = 0$, then the value of $CtrOf[n]$ doesn't matter, since firing n does not change the counter.) We make the following requirements on the arrays *CtrOf* and *Incr*.

C1. For any nodes m and n, if $CtrOf[m] = CtrOf[n]$ then m and n belong

to the same process. (We allow the same counter to be assigned to multiple nodes, but only if they all belong to the same process.)

C2. If a node n has a synchronizing out-edge, then $Incr[n] > 0$.

C3. For any counter c, let $IncrSum(c)$ equal the sum of $Incr[n]$ for all nodes n with $CtrOf[n] = c$. Then $IncrSum(c)$ has the same value for all counters c; we call this value Q.

As we explain in Section 3.3.4 below, condition C3 can be weakened.

Let algorithm *PMGAlg*2 be obtained from algorithm *PMGAlg*1 by adding a global variable ct, where $ct[c]$ is the value of counter c. We add to the global **variable** statement the declaration

$$ct = [c \in Ctrs \mapsto 0]$$

that initializes $ct[c]$ to 0 for every counter c. We add to the body of the **while** loop, before the assignment to i, the statement

$$ct[CtrOf[self[i]]] := ct[CtrOf[self[i]]] + Incr[self[i]] \ ;$$

In algorithm *PMGAlg*2, the counters do nothing except count. We now modify this algorithm by replacing the **when** test with an expression that depends on ct rather than μ. But first, we need some definitions.

Let $cnt0(n)$ be the amount by which node n's counter is incremented before n is fired for the first time. In other words, for each process $\langle p_1, \ldots, p_k \rangle$, we define $cnt0(p_i)$ to be the sum of all $Incr[p_j]$ such that $j < i$ and $CtrOf[p_j] = CtrOf[p_i]$. Let $bct(n)$ equal $ct[CtrOf[n]] - cnt0(n)$, so $bct(n)$ equals 0 just before n fires for the first time.

By condition C3, any time in the execution that n is about to fire, $bct(n)$ equals $Q * \#(n)$. More generally, if n is the next node of its process to fire, or if $Incr[n] > 0$, then

$$Q * \#(n) = \lceil bct(n) \rceil^Q \tag{9}$$

By Observation 1 and condition C1, this implies:

Observation 2 Throughout the execution of Algorithm *PMGAlg*2, if $\langle m, n \rangle$ is a synchronizing edge and $n = \pi[i]$ for some process π (so n is the next node of process π to fire), then

(a) $Q * \mu[\langle m, n \rangle] \ = \ Q * \mu_0[\langle m, n \rangle] + \lceil bct(m) \rceil^Q - \lceil bct(n) \rceil^Q$

(b) $\lceil bct(n) \rceil^Q - \lceil bct(m) \rceil^Q \ \in \ -Q * \delta_{\mu_0}(n, m) \,..\, Q * \mu_0[\langle m, n \rangle]$

Let $\langle m, n \rangle$ be a synchronizing in-edge of a node n. It follows from part (a) of Observation 2 that, if n is the next node of its process to fire, then $\mu[\langle m, n \rangle] > 0$ iff

$$Q * \mu_0[\langle m, n \rangle] + \lceil bct(m) \rceil^Q - \lceil bct(n) \rceil^Q > 0$$

This inequality is equivalent to

$$\lceil bct(n) \rceil^Q - \lceil bct(m) \rceil^Q \leq Q * \mu_0[\langle m, n \rangle] - 1$$

which by part (b) of Observation 2 is equivalent to

$$(10) \qquad \lceil bct(n) \rceil^Q - \lceil bct(m) \rceil^Q \neq Q * \mu_0[\langle m, n \rangle]$$

Define $CtTest(\langle m, n \rangle)$ to equal formula (10). (The formula depends only on ct, the edge $\langle m, n \rangle$, and constants.)

We have seen that, if $n = \pi[i]$ for some process π and $\langle m, n \rangle$ is a synchronizing in-edge of n, then $CtTest(\langle m, n \rangle)$ is equivalent to $\mu[\langle m, n \rangle] > 0$. We therefore obtain an algorithm that is equivalent to algorithm *PMGAlg*2 by replacing the **when** statement of *PMGAlg*2 with

when $\forall\, e \in SInEdges(self[i]) : CtTest(e)$

(Remember that *PMGAlg*2 is the same as algorithm *PMGAlg*1 of Figure 2 except with counter ct added.) We call the resulting algorithm *PMGAlg*3.

In algorithm *PMGAlg*3, the variable μ is never read, only written. In this and our subsequent algorithms for executing a process marked graph, the variable μ is a history variable whose only function is to demonstrate the equivalence of the algorithm with the specification *MGSpec* of a marked graph. It will not appear in the actual code for synchronizing a dataflow computation that is obtained from our final algorithm.

3.3.3 Bounding the Counters

We now modify algorithm *PMCAlg*3 to bound the counter values by incrementing them modulo N, for a suitably chosen N. By part (b) of Observation 2, applying number fact 2 with

$$\begin{aligned} a &\leftarrow \lceil bct(n) \rceil^Q - \lceil bct(m) \rceil^Q \\ b &\leftarrow -Q * \delta_{\mu_0}(n, m) \\ p &\leftarrow Q * \delta_{\mu_0}(n, m) + Q * \mu_0[\langle m, n \rangle] \end{aligned}$$

shows that if

$$(11) \qquad N > Q * \delta_{\mu_0}(n, m) + Q * \mu_0[\langle m, n \rangle]$$

then (10) is equivalent to

$$\lceil bct(n) \rceil^Q \ominus \lceil bct(m) \rceil^Q \;\neq\; (Q * \mu_0[\langle m, n \rangle]) \,\%\, N \tag{12}$$

We assume that (11) holds for all synchronizing edges $\langle m, n \rangle$. If N is also a multiple of Q, then applying number facts 1 and 3 shows that (12) is equivalent to

$$\lceil bct(n)\%N \rceil^Q \ominus \lceil bct(m)\%N \rceil^Q \;\neq\; (Q * \mu_0[\langle m, n \rangle]) \,\%\, N \tag{13}$$

The definition of bct implies that $bct(n)\%N$ equals $ct[CtrOf[n]] \ominus cnt0(n)$, which by number fact 1 equals $(ct[CtrOf[n]]\%N) \ominus cnt0(n)$. Let cnt be an array such that $cnt[c] = ct[c]\%N$ for every counter c. Define $bcnt$ by

$$bcnt(n) \;\triangleq\; cnt[CtrOf[n]] \ominus cnt0(n)$$

Then $bcnt(n) = bct(n)\%N$ for every node n. Formula (13) is therefore equivalent to

$$\lceil bcnt(n) \rceil^Q \ominus \lceil bcnt(m) \rceil^Q \;\neq\; (Q * \mu_0[\langle m, n \rangle]) \,\%\, N \tag{14}$$

Define $CntTest(e)$ to equal formula (14). We have seen that, if $cnt[c] = ct[c] \,\%\, N$ for every counter c, then $CntTest(e)$ is equivalent to $CtTest(e)$. Number fact 1 therefore implies that we obtain an algorithm equivalent to $PMGAlg3$ by replacing ct with an array cnt of counters that are incremented modulo N and replacing $CtTest(e)$ with $CntTest(e)$. This algorithm, which we call $PMGAlg4$, is given in Figure 3. Unlike operators such as $\oplus$ that are defined solely in terms of constants, $CntTest$ is defined in terms of the variable cnt. Its definition therefore appears with the algorithm code in a **define** statement.

3.3.4 A Fine-Grained Algorithm

Algorithm $PMGAlg4$ uses bounded counters. However, in a single atomic step it reads the counters of every node with a synchronizing in-edge to $self[i]$ and writes $self[i]$'s counter. Our final algorithm is a finer-grained version that performs these reads and writes as separate atomic steps.

To see why such a finer-grained algorithm works, we first write a corresponding finer-grained version of the simple algorithm $MGSpec$ for executing a marked graph. For each node n, algorithm $MGSpec$ reads $\mu[e]$ for every in-edge e of n in one atomic step. When a token is placed on an edge of a marked graph, it remains on that edge until the edge's destination node

--algorithm $PMGAlg4$
variable $\mu = \mu_0$; $cnt = [c \in Ctrs \mapsto 0]$
define $CntTest(e) \;\triangleq$

$$
\begin{array}{ll}
\text{LET} & bcnt(p) \triangleq cnt[CtrOf[p]] \ominus cnt0(p) \\
 & m \triangleq e[1] \\
 & n \triangleq e[2] \\
\text{IN} & \lceil bcnt(n) \rceil^{Q} \ominus \lceil bcnt(m) \rceil^{Q} \neq (Q * \mu_0[\langle m, n \rangle]) \,\%\, N
\end{array}
$$

process $Proc \in \Pi$
  variable $i = 1$;
  begin lab: **while** TRUE **do**
      when $\forall\, e \in SInEdges(self[i]) : CntTest(e)$;
      $\mu := Fire(self[i], \mu)$;
      $cnt[CtrOf[self[i]]] := cnt[CtrOf[self[i]]] \oplus Incr[self[i]]$;
      $i := (i \,\%\, Len(self)) + 1$;
    end while
  end process
end algorithm

Figure 3: An algorithm describing the execution of the process marked graph $\langle \Gamma, \mu_0, \Pi \rangle$.

is fired. Hence, the algorithm will work if the read of each $\mu[e]$ is made a separate action. This is done in algorithm *MGFGSpec* of Figure 4. (It will not matter that the write of $\mu[e]$ for all out-edges of n is a single atomic step.)

Algorithm *MGFGSpec* uses the local process variable *ToCheck* to keep track of the set of in-edges of node *self* that have not yet been found to contain a token. Its initial value is irrelevant and is left unspecified. The empty set is written $\{\}$ and $\setminus$ is set difference, so $S \setminus \{e\}$ is the set of all elements of S other than e. The **with** statement nondeterministically sets e to an arbitrary element of *ToCheck*, so the inner **while** loop checks the in-edges of node *self* in an arbitrary order, exiting after finding that they all contain a token. The atomic actions are the first assignment to *ToCheck*, each iteration of the inner **while** loop, and the assignment to μ. Because only the process for node n removes tokens from n's in-edges, algorithm *MGFGSpec* is equivalent to the specification *MGSpec* of the execution of marked-graph $\langle \Gamma, \mu_0 \rangle$.

In a similar way, we can replace the simple algorithm $PMGAlg1$ by an equivalent one in which the read of $\mu[e]$ for each edge e in $SInEdges(self[i])$

```
--algorithm MGFGSpec
variable μ = μ₀ ;
process Node ∈ Γ.nodes
  variable ToCheck
  begin lab : while TRUE do
                ToCheck := InEdges(self) ;
         loop : while ToCheck ≠ {} do
                  with e ∈ ToCheck do
                    if μ[e] > 0 then ToCheck := ToCheck \ {e} end if
                  end with
                end while
         fire : μ := Fire(self, μ) ;
              end while
  end process
end algorithm
```

Figure 4: A finger-grained algorithm for executing the marked graph $\langle\Gamma,\ \mu_0\rangle$.

is made a separate action. If we add a modulo-N counter *cnt* to this finer-grained algorithm, it remains the case that if n is the next node in its process to be fired, then $\mu[\langle m, n\rangle] > 0$ is equivalent to $CntTest(\langle m, n\rangle)$ for any synchronizing in-edge $\langle m, n\rangle$ of n. The same argument that showed algorithm *PMGAlg*4 to be equivalent to *MGSpec* therefore shows that algorithm *FGAlg* of Figure 5 is equivalent to algorithm *MGFGSpec*, and hence to the specification *MGSpec*.

Each iteration of the outer **while** loop implements the firing of node $self[i]$. When applying the algorithm, we usually unroll this loop as a sequence of separate copies of the body for each value of i. If $self[i]$ has no input synchronizing edges, then the inner **while** statement performs 0 iterations and can be eliminated, along with the preceding assignment to *ToCheck*, for that value of i. If $Incr[self[i]] = 0$, then the statement labeled *fire* does nothing and can be eliminated.

For convenience, we list here all the assumptions that we used in showing that *FGAlg* is equivalent to *MGSpec*.

C0. $\langle\Gamma,\ \mu_0,\ \Pi\rangle$ is a live process marked graph.

C1. For any nodes m and n, if $CtrOf[m] = CtrOf[n]$, then m and n belong to the same process.

C2. If node n has a synchronizing out-edge, then $Incr[n] > 0$.

```
--algorithm FGAlg
variable μ = μ0 ; cnt = [c ∈ Ctrs ↦ 0]
define CntTest(e) ≜
            LET bcnt(p) ≜ cnt[CtrOf[p]] ⊖ cnt0(p)
                m ≜ e[1]
                n ≜ e[2]
            IN  ⌈bcnt(n)⌉^Q ⊖ ⌈bcnt(m)⌉^Q ≠ (Q ∗ μ0[⟨m, n⟩]) % N
process Proc ∈ Π
  variables i = 1 ; ToCheck
  begin lab : while TRUE do
                ToCheck := SInEdges(self[i]) ;
         loop : while ToCheck ≠ {} do
                  with e ∈ ToCheck do
                    if CntTest(e) then
                      ToCheck := ToCheck \ {e} end if
                  end with
                end while
         fire : cnt[CtrOf[self[i]]] := cnt[CtrOf[self[i]]] ⊕ Incr[self[i]] ;
                μ := Fire(self, μ) ;
                i := (i%Len(self)) + 1 ;
              end while
  end process
end algorithm
```

Figure 5: Our fine-grained algorithm for executing the process marked graph $\langle \Gamma, \mu_0, \Pi \rangle$.

C3. $IncrSum(c) = Q$ for every counter c.

C4. (a) N is divisible by Q, and
  (b) $N > Q * \delta_{\mu_0}(n, m) + Q * \mu_0[\langle m, n \rangle]$, for every synchronizing edge $\langle m, n \rangle$.

Assumption C3 is stronger than necessary. If we replace Q by $IncrSum(n)$ in the definition of $CntTest(e)$ (formula (14)), then we can replace C3 and C4(a) by

C3. $IncrSum(CtrOf[m]) = IncrSum(CtrOf[n])$ for every synchronizing edge $\langle m, n \rangle$.

C4. (a) N is divisible by $IncrSum(c)$, for every counter c.

However, we expect that most applications will use either a single counter for all nodes of a process or else enough counters so that $\mathit{IncrSum}(c) = 1$ for every counter c. With a single counter for each process, the generalized version of C3 implies that $\mathit{IncrSum}(c)$ is the same for all counters c if the graph Γ is connected. This generalization therefore does not seem to be useful.

3.3.5 An Optimization

When implementing a multiprocess algorithm like *FGAlg* on today's multiprocessors, an access to a shared memory location that is not in the processor's cache is many times slower than an operation local to the process. We can optimize *FGAlg* by eliminating some unnecessary reads by one process of another process's counters if there can be more than one token on a synchronizing edge. (This is the case for our producer/consumer examples, but not for barrier synchronization.)

When process *self* computes $\mathit{CntTest}(e)$, it is determining whether the number of tokens on edge e is greater than 0. It could just as easily determine $\mu[e]$, the actual number of tokens on e. If it finds that $\mu[e] > 1$, then the process knows that the tokens needed to fire node $\mathit{self}[i]$ the next $\mu[e] - 1$ times are already on edge e. Therefore, it can eliminate the next $\mu[e] - 1$ tests for a token on edge e, eliminating those reads of the counter for e's source node.

A derivation similar to that of the definition (14) of $\mathit{CntTest}(\langle m, n \rangle)$ shows that, if n is the next node of its process to fire, then

$$\lceil \mathit{bcnt}(m) \rceil^Q \ominus \lceil \mathit{bcnt}(n) \rceil^Q \;\oplus\; Q * \mu_0[\langle m, n \rangle]$$

equals $Q * \mu[\langle m, n \rangle]$. Hence, $\mu[\langle m, n \rangle]$ equals

$$\frac{\lceil \mathit{bcnt}(m) \rceil^Q \ominus \lceil \mathit{bcnt}(n) \rceil^Q}{Q} \;\oplus\; \mu_0[\langle m, n \rangle] \tag{15}$$

Define $\mathit{CntMu}(\langle m, n \rangle)$ to equal (15). We modify the inner **while** statement of *FGAlg* so it evaluates $\mathit{CntMu}(e)$ rather than $\mathit{CntTest}(e)$, removing e from *ToCheck* if it finds $\mathit{CntMu}(e) > 0$. We also make the process remove e from *ToCheck* the next $\mathit{CntMu}(e) - 1$ times it executes the **while** statement before re-evaluating $\mathit{CntMu}(e)$. To do this, we add a local variable $\mathit{toks}[e]$ whose value is set to $\mathit{CntMu}(e) - 1$ when that expression is evaluated. Let $\mathit{ProcInEdges}(\pi)$ be the set of all synchronizing in-edges of process π's nodes. We optimize algorithm *FGAlg* by adding the local variable declaration

$$toks = [e \in \mathit{ProcInEdges}(\mathit{self}) \mapsto \mu_0[e] - 1]$$

to process *Proc* and changing the **do** clause of the **with** statement within the inner **while** loop to

$$\begin{array}{l} \textbf{if}\ \ toks[e] \leq 0\ \ \textbf{then}\ toks[e]\ := \mathit{CntMu}(e) - 1 \\ \qquad\qquad\qquad\ \ \textbf{else}\ \ toks[e]\ := toks[e] - 1 \\ \textbf{end if}\ ; \\ \textbf{if}\ toks[e] \neq -1\ \textbf{then}\ \mathit{ToCheck}\ := \mathit{ToCheck} \setminus \{e\}\ \textbf{end if} \end{array}$$

Remember that this optimization can eliminate memory accesses only for edges e of the process marked graph that can contain more than one token. It accomplishes nothing for barrier synchronization.

3.4 The Producer/Consumer System

As our first example, let us apply algorithm *FGAlg* to the process marked graph (3) representing producer/consumer synchronization, except with an arbitrary number B of tokens on edge $\langle c_2, p_1 \rangle$ instead of just 3. Recall that each of those tokens represents a buffer, a token on edge $\langle p_1, p_2 \rangle$ represents a *Produce* operation, and a token on edge $\langle c_1, c_2 \rangle$ represents a *Consume* operation. We let the producer and consumer processes each have a single counter that is incremented by 1 when p_2 or c_2 is fired. We then have $Q = 1$, and condition C4 becomes $N > B$.

Since firing p_1 or c_1 does not increment a counter, statement *fire* can be eliminated in the iterations of the outer **while** loop for $i = 1$. Since p_2 and c_2 have no synchronizing in-edges, statement *loop* can be eliminated in the iteration for $i = 2$. We unroll the loop to combine the iterations for $i = 1$ and $i = 2$ into one loop body that contains the statement *loop* for $i = 1$ followed by statement *fire* for $i = 2$. Since the execution of the *Produce* or *Consume* operation begins with the firing of p_1 or c_1 and ends with the firing of p_2 or c_2, its code comes between the code for the two iterations.

Instead of a single array *cnt* of variables, we use variables p and c for the producer's and consumer's counters. We have $cnt0(p_1) = cnt0(c_1) = 0$, so the conditions $\mathit{CntTest}(\langle c_2, p_1 \rangle)$ and $\mathit{CntTest}(\langle p_2, c_1 \rangle)$ become $p \ominus c \neq B$ and $p \ominus c \neq 0$, respectively. Writing the producer and consumer as separate **process** statements, we get the algorithm of Figure 6. The statements *Produce* and *Consume* represent the code for executing those operations. This algorithm was apparently first published in [6]. It can be optimized as described in Section 3.3.5 above.

```
--algorithm ProdCons
variables p = 0 ; c = 0
process Prod = "p"
  begin lab : while TRUE do
             loop : while p ⊖ c = B do skip end while
                      Produce ;
             fire : p := p ⊕ 1 ;
                 end while
  end process
process Cons = "c"
  begin lab : while TRUE do
             loop : while p ⊖ c = 0 do skip end while
                      Consume ;
             fire : c := c ⊕ 1 ;
                 end while
  end process
end algorithm
```

Figure 6: The producer/consumer algorithm obtained from algorithm *FGAlg* for the process marked graph (3), with B buffers instead of 3.

3.5 Barrier Synchronization

We now apply algorithm *FGAlg* to barrier synchronization. We begin with the process marked graph (7) of Section 2 (page 6). We use one counter per process, incremented by 1 by the process's first node and left unchanged by its second node. Hence, $Q = 1$. Condition C4(b) requires $N > 2$—for example, for edge $\langle a_1, b_2 \rangle$ we have $\delta_{\mu_0}(b_2, a_1) + \mu_0[\langle a_1, b_2 \rangle]$ equals $2 + 0$. We use the name of the process as its counter name, so process a (the top process of (7)) uses counter a. Since $cnt0(a_1) = 0$ and $cnt0(b_2) = 1$, formula $CntTest(\langle a_1, b_2 \rangle$ equals

$$cnt[b] \ominus cnt[a] \neq 1 \tag{16}$$

We generalize from (7) in the obvious way to a process marked graph with a set Π of processes, and we apply algorithm *FGAlg* to this process marked graph. We let the set of counters be the set of processes, and we let each process π increment $cnt[\pi]$ by 1 when firing node $\pi[1]$ and leave it unchanged when firing node $\pi[2]$. Since $\pi[1]$ has no synchronizing in-edges and firing $\pi[2]$ does not increment counter π, unrolling the body of the outer **while** yields a loop body with statement *fire* for $i = 1$ followed by statement *loop* for $i = 2$. The statement *PerformComputation* containing the code for

```
--algorithm Barrier1
variable cnt = [c ∈ Π ↦ 0]
process Proc ∈ Π
   variable ToCheck
   begin lab : while TRUE do
                  Perform Computation ;
          fire :  cnt[self] := cnt[self] ⊕ 1 ;
                  ToCheck := Π \ {self} ;
          loop :  while ToCheck ≠ {} do
                     with π ∈ ToCheck do
                        if cnt[self] ⊖ cnt[π] ≠ 1 then
                           ToCheck := ToCheck \ {π} end if
                     end with
                  end while
               end while
   end process
end algorithm
```

Figure 7: The barrier synchronization algorithm obtained from algorithm *FGAlg* for the generalization of process marked graph (7).

the computation corresponding to edge $\langle self[2], self[1]\rangle$ precedes the *fire* statement. For each process π, we have $cnt0(\pi[1]) = 0$ and $cnt0(\pi[2]) = 1$, so $CntTest(\langle \pi[1], self[2]\rangle)$ equals $cnt[self] \ominus cnt[\pi] \neq 1$, for any process $\pi \neq self$. We then get algorithm *Barrier*1 of Figure 7, for any $N > 2$. This algorithm appears to be new. It is the simplest barrier synchronization algorithm using only reads and writes that we know of.

In a similar fashion, we can derive a barrier synchronization algorithm from algorithm *FGAlg* applied to the generalization of the process marked graph (8) of Section 2 (page 7). In that generalization, a single distinguished process π_0 plays the role of process b (the middle process). Again, each process has a single counter. The algorithm for every process *self* other than π_0 is the same as in algorithm *Barrier*1, except that node $self[2]$ has only a single synchronizing in-edge for whose token it must wait. Since nodes $\pi_0[1]$ and $\pi_0[3]$ have neither synchronizing in-edges nor out-edges, the iterations of process π_0's loop for $i = 1$ and $i = 3$ do nothing. For any process $\pi \neq \pi_0$ we have $CntTest(\langle \pi[1], \pi_0[2]\rangle)$ equals $cnt[\pi_0] \ominus cnt[\pi] \neq 0$ which is equivalent to $cnt[\pi_0] \neq cnt[\pi]$, since $cnt[\pi_0]$ and $cnt[\pi]$ are in $0 \,..\, (N-1)$. This leads to the algorithm *Barrier*2 of Figure 8, where assumption C4 again requires $N > 2$. This algorithm is also new. It may be more efficient than

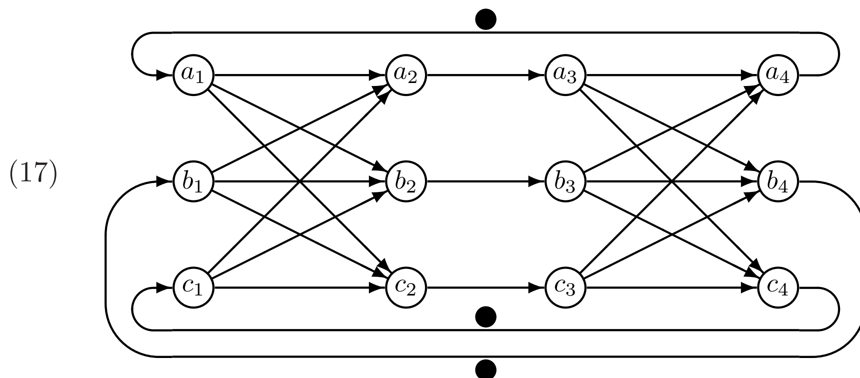
```
--algorithm Barrier2
variable cnt = [c ∈ Π ↦ 0]
process Proc ∈ Π \ {π0}
  begin lab : while TRUE do
                Perform Computation ;
         fire : cnt[self] := cnt[self] ⊕ 1 ;
         loop : while cnt[self] ⊖ cnt[π0] = 1 do skip
                end while
              end while
  end process
process Proc0 = π0
  variable ToCheck
  begin lab : while TRUE do
                Perform Computation ;
                ToCheck := Π \ {π0} ;
         loop : while ToCheck ≠ {} do
                  with π ∈ ToCheck do
                    if cnt[π0] = cnt[π] then
                       ToCheck := ToCheck \ {π} end if
                  end with
                end while
         fire : cnt[π0] := cnt[π0] ⊕ 1 ;
              end while
  end process
end algorithm
```

Figure 8: The barrier synchronization algorithm obtained from algorithm *FGAlg* for the generalization of process marked graph (8).

algorithm *Barrier*1 because it performs fewer memory operations (about $2P$ rather than P^2, for P processes). However, the synchronization uses a longer information-flow path (length 2 rather than length 1), which can translate into a longer synchronization delay.

Let us unroll the loops once in the marked graph (7) to obtain the following process marked graph.

(17)



Each process π has two computation edges, $\langle \pi[4], \pi[1] \rangle$ and $\langle \pi[2], \pi[3] \rangle$. This graph also describes barrier synchronization, where successive executions of the processes' computations are represented by separate computation edges.

We implement this process marked graph with two counters per process that are each incremented by 1, one when $\pi[1]$ fires and the other when $\pi[3]$ fires. We call process π's two counters $\langle \pi, 0 \rangle$ and $\langle \pi, 1 \rangle$, and we write $cnt[\pi, j]$ instead of $cnt[\langle \pi, j \rangle]$. We again have $Q = 1$, but this time condition C4(b) requires only $N > 1$, so we can let $N = 2$ and use 1-bit counters. Again, $CntTest(\langle a_1, b_2 \rangle)$ equals (16), which for $N = 2$ is equivalent to $cnt[b] = cnt[a]$. Algorithm *FGAlg* therefore yields the barrier synchronization algorithm *Barrier*3 of Figure 9. Algorithm *Barrier*3 is known and is used in the BSP toolkit [4].

We can also obtain *Barrier*3 from algorithm *Barrier*1 by letting $N = 4$ and representing counter values with a 2-bit Gray code.[3] We let $cnt[\pi, j]$ be bit j of the Gray code representation of $cnt[\pi]$, and we use part (b) of Observation 2 to show that the tests in *Barrier*1 and *Barrier*3 are equivalent.

These three barrier synchronization algorithms require that at least one process reads the counters of every other process. This may be impractical for a large set of processes. A number of "composite" barrier synchronization

[3] A k-bit Gray code represents the integers from 0 through $2^k - 1$ using k bits in such a way that incrementing a number by 1 modulo 2^k changes only one bit of its representation.

```
--algorithm Barrier3
variable cnt = [π ∈ Π, i ∈ {0,1} ↦ 0]
process Proc ∈ Π
   variables ToCheck ; j = 0
   begin lab : while TRUE do
                  Perform Computation ;
           fire : cnt[self, j] := cnt[self, j] ⊕ 1 ;
                  ToCheck := Π \ {self} ;
           loop : while ToCheck ≠ {} do
                     with π ∈ ToCheck do
                        if cnt[self, j] = cnt[π, j] then
                              ToCheck := ToCheck \ {π} end if
                     end with
                  end while ;
                  j := j ⊕ 1
                end while
   end process
end algorithm
```

Figure 9: The barrier synchronization algorithm obtained from algorithm *FGAlg* for the generalization of process marked graph (17).

algorithms have been proposed in which multiple barrier synchronizations are performed, each involving a small number of processes [12]. It is obvious how to implement these algorithms using multiple instances of any barrier synchronization algorithm, each with its own set of variables, for the component barriers.

Each of these composite barrier synchronization algorithms can be described by a process marked graph. If we assign a separate counter to every node with synchronizing out-edges and apply algorithm *FGAlg*, we get a version of the composite algorithm using *Barrier*1 as the component algorithm. However, we can also use only a single counter per process. Applying *FGAlg* then gives a simpler version of the composite algorithm in which all the component synchronizations use the same variables. It appears that implementing composite barrier synchronization with only one synchronization variable per process is a new idea.

3.6 Implementing the Algorithm

FGAlg is an algorithm for executing a process marked graph. The discussion of barrier synchronization above shows that it is easy to turn *FGAlg* into an algorithm for performing a dataflow computation described by the graph. If the process out-edge from node $self[i]$ is a computation edge, the computation represented by that edge is executed immediately after statement *fire*.

To implement a multithreaded dataflow computation, we must translate algorithm *FGAlg* into an actual program. Unlike an algorithm, a program has no clearly defined atomic actions. Conventional wisdom says that we can turn a multiprocess algorithm directly into multithreaded code if it satisfies the following condition: Each atomic action of the algorithm accesses only a single shared variable whose value can be stored in a single word of memory. This condition does not quite hold of algorithm *FGAlg* because the evaluation of $CntTest(e)$ accesses the counters of both the source and destination nodes of edge e, which are shared variables. Also, statement *fire* contains two accesses to the counter of node $self[i]$. (The shared variable μ can be ignored, since it is a history variable that is not implemented.) The destination of edge e is $self[i]$, and the counter of $self[i]$ is modified only by process *self*. The process can therefore read from a local copy of that counter, so these two actions access only a single shared variable—the counter of the source node of e in the evaluation of $CntTest(e)$ and the counter of $self[i]$ in statement *fire*. With most memory systems, it is not necessary to introduce a separate local copy of the counter. The thread's

read of a variable that only it modifies is performed from a local cache that is not accessed by any other thread, so it is effectively a read of a local variable.

Conventional wisdom implies that we can therefore implement algorithm *FGAlg* directly as a multithreaded program. However, this conventional wisdom applies only to an implementation in which reads and writes of shared variables are atomic. More precisely, it applies only to programs run on a multiprocessor with a sequentially consistent memory [7]. It can be shown that if N is at least one larger than is required by condition C4(b), then a simple modification to the definition of *CntTest* makes the algorithm work with regular memory registers—a weaker class than atomic registers [8]. However, most modern multiprocessors provide neither sequentially consistent memories nor regular memory registers. An implementation of algorithm *FGAlg* on such computers requires additional synchronization operations. We now examine what operations are needed.

We cannot consider every type of multiprocessor memory that has been proposed. Instead, we make some general observations that apply to most of them. Most memory models are defined in terms of two (transitive) partial orders on memory accesses—a *program order* $\rightarrow$ and a *memory order* $\prec$. The relation $\rightarrow$ is defined by $A \rightarrow B$ iff A and B are accesses by the same process and A precedes B in the process's program. The memory model requires that, for any execution of a program, there exists a relation $\prec$ satisfying certain properties. The following properties are common to most models.

M1. All accesses to the same memory location are totally ordered by $\prec$.

M2. If $A \rightarrow B$ and A and B are accesses to the same memory location, then $A \prec B$.

M3. A read R obtains the value written by the write W to the same location such that $W \prec R$ and there is no write W' of the same location with $W \prec W' \prec R$. (Read R obtains the initial value if there is no write W of the same location with $W \prec R$.)

In sequential consistency, M1 and M2 is strengthened to require that $\prec$ orders all accesses and that $A \rightarrow B$ implies $A \prec B$ even for accesses A and B to different memory locations. We assume a multiprocessor memory satisfying M1–M3.

Let $R \xrightarrow{D} A$ mean that $R \rightarrow A$ and that R is a read whose value was used to determine that access A should have occurred. For example, $R \xrightarrow{D} A$

holds if R was executed in evaluating an **if** test that equaled TRUE and A was executed in the **then** clause. Many memories models also assume:

M4. If $R \stackrel{D}{\rightarrow} A$, then $R \prec A$.

Condition M4 does not hold for the Alpha memory model [1], and executions by some Alpha processors violate it. We suspect that memory models satisfying M4 will be less common as engineers aggressively optimize multiprocessor memory access—especially for processors on different chips. We therefore do not assume it.

Multiprocessors that do not have sequentially consistent memories usually provide a *memory barrier* (MB) instruction for synchronizing memory accesses. Execution of an MB instruction is a memory access that neither reads nor writes, but satisfies:

MB. If $A \rightarrow M \rightarrow B$ and M is an execution of an MB instruction, then $A \prec B$.

MB instructions tend to be quite expensive, often stalling a processor until the executions of all previously issued memory accesses are completed. We will show that the following rules imply that an implementation of algorithm *FGAlg* ensures the necessary synchronization in any memory model satisfying M1–M3 and MB.

S1. For any computation code $\mathcal{E}$, there must be an MB between the end of $\mathcal{E}$ and both (a) the beginning of the next computation code in the same process (which may be $\mathcal{E}$ itself) and (b) the next non-zero increment of a counter by the process.

S2. If M4 is not satisfied, then there must be an MB between the code generated for the inner **while** loop and for the *fire* statement in algorithm *FGAlg*. This MB must occur in the obvious place even if either the loop or the *fire* statement is a no-op and is omitted (either because there are no synchronizing in-edges or because the *fire* statement increments the counter by 0). The MB is not needed if both are no-ops.

Condition S1 can be satisfied by putting an MB at the end of each computation code. This is sufficient if M4 is satisfied.

As an example of how S1 and S2 are satisfied for a memory model in which M4 does not hold, we consider the algorithm for the graph (8). An MB after the computation code corresponding to edge $\langle a_2, a_1 \rangle$ satisfies S1

for that computation code. It also satisfies S2 for the code corresponding to node a_1, since a_1 has no synchronizing in-edges and hence no inner **while** loop. An MB at the beginning of that process's computation code satisfies S2 for the code corresponding to node a_2, since that code has no *fire* statement. Similarly, barriers at the beginning and end of the computation code corresponding to $\langle c_2, c_1 \rangle$ satisfy S1 and S2 for process c. Process b just needs a single MB between the inner **while** and the *fire* statement of the code that implements node b_2. This obviously satisfies S2 for that node. Nodes b_1 and b_3 generate no code, since they have no synchronizing in-edges and increment the counter by 0. Therefore, S2 does not apply to those two nodes, and this single MB also satisfies S1 for the computation corresponding to edge $\langle b_3, b_1 \rangle$.

We now show that conditions S1 and S2 are sufficient for any arbitrary implementation of *FGAlg*. Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be the computation code represented by edges e_1 and e_2 of the marked graph. Let E_1 and E_2 be executions of $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively, such that E_2 must follow E_1. Remember that each execution E_i is represented by the presence of a token on edge e_i, and execution E_2 must follow E_1 if the token representing E_1 must be removed from e_1 before the token representing E_2 can be placed on e_2. We must ensure that if A_1 and A_2 are memory accesses in E_1 and E_2, then $A_1 \prec A_2$.

By assumption MB, condition S1 implies that $A_1 \prec A_2$ if $\mathcal{E}_1$ and $\mathcal{E}_2$ are in the same process. We now consider the case in which $\mathcal{E}_1$ and $\mathcal{E}_2$ are in different processes.

Let p be the destination node of edge e_1 and let q be the source node of edge e_2. Computation $\mathcal{E}_1$ must precede $\mathcal{E}_2$ iff the firing of p that removes from e_1 the token corresponding to $\mathcal{E}_1$ must precede the firing of q that places on e_2 the token corresponding to $\mathcal{E}_2$. Let F_p and F_q be these two node-firing events. Event F_p must precede F_q iff there is a path $\langle n_1, \ldots, n_k \rangle$ with $p = n_1$ and $q = n_k$, and there are firing events F_i of n_i for each i, with $F_p = F_1$ and $F_q = F_k$, such that each F_i precedes F_{i+1}.

For each i, let W_i be the write of the counter value during the firing F_i, or a no-op if the counter is not written (because it is incremented by 0). For $i > 1$, let R_i be the read of node n_{i-1}'s counter during the firing F_i if $\langle n_{i-1}, n_i \rangle$ is a synchronizing edge; otherwise let R_i be a no-op. For the purposes of the proof, we pretend that those no-ops are actual instructions that appear in the code. We then have $R_i \rightarrow W_i$ for each i in $2 \,..\, k$.

For an edge $\langle n_{i-1}, n_i \rangle$, firing F_{i-1} must precede firing F_i iff the token removed from $\langle n_{i-1}, n_i \rangle$, by F_i is the one placed on that edge by F_{i-1} or by a later firing of node n_{i-1}. If $\langle n_{i-1}, n_i \rangle$ is process edge (not a synchronizing edge), then this is the case iff $W_{i-1} \rightarrow R_i$. If If $\langle n_{i-1}, n_i \rangle$ is a synchronizing

edge, then this is the case iff the value of node n_{i-1}'s counter read by R_i is the one written by W_{i-1} or by a later write. By M1, M3, and the transitivity of $\prec$, this implies $W_{i-1} \prec R_i$. We therefore have a sequence of reads, writes, and no-ops $W_1, R_2, W_2, \ldots W_{k-1}, R_k$ with, for each $i > 1$

- $R_i \rightarrow W_i$, with $R_i \stackrel{D}{\rightarrow} W_i$ if R_i is not a no-op.
- $W_{i-1} \rightarrow R_i$ if W_{i-1} and R_i are performed by the same process, and $W_{i-1} \prec R_i$ if they are performed by different processes. (In the latter, case C2 implies that neither W_{i-1} nor R_i is a no-op.)

Moreover, since W_1 occurs during the firing F_p and R_k occurs during the firing F_q, we also have $A_1 \rightarrow W_1$ and $R_k \rightarrow A_2$, with $R_k \stackrel{D}{\rightarrow} A_2$ if R_k is not a no-op.

It is not hard to see that if we remove the W_i and R_i that are no-ops and renumber them to remove gaps in the sequence, we obtain a sequence $W_1, R_2, W_2, \ldots W_{j-1}, R_j$ with $R_i \stackrel{D}{\rightarrow} W_i$ and $W_{i-1} \prec R_i$ for all $i > 1$, and with $A_1 \rightarrow W_1$ and $R_j \stackrel{D}{\rightarrow} A_2$. (Since $\mathcal{E}_1$ and $\mathcal{E}_2$ are assumed to be in different processes, $j > 1$ and the sequence is non-empty.) Condition S1 implies $A_1 \prec W_1$. If we can replace the relations $R_i \stackrel{D}{\rightarrow} W_i$ and $R_j \stackrel{D}{\rightarrow} A_2$ by $R_i \prec W_i$ and $R_j \prec A_2$, the required conclusion $A_1 \prec A_2$ follows from the transitivity of $\prec$. Condition M4 obviously allows us to make this replacement. If M4 does not hold, then the required $\prec$ relations follow from S2 and MB. This completes the proof that S1 and S2 suffice.

3.7 Caching Behavior

We now consider the efficiency of the optimized version of algorithm *FGAlg* with caching memories. In a caching memory system, a process may acquire either a read/write copy of a memory location or a read-only copy in its cache. Acquiring a read-write copy invalidates any copies in other processes' caches. We first consider systems in which multiple processors can have read-only copies, so acquiring a read-only copy does not invalidate other copies, but converts a read/write copy to read-only.

As observed above, the read of a process's counter by that process is local or can be performed on a local copy of the counter. The accesses of shared variables are the write by process *self* of node $\mathit{self}[i]$'s counter in statement *fire*, and its read of a node m's counter by the evaluation of $\mathit{CntMu}(\langle m, \mathit{self}[i] \rangle)$. After π performs that read of m's counter, the value it read will remain in its cache until the counter is written again.

Assume now that each counter is incremented when firing only one node, in which case we can let $Q = 1$. Each write of node m's counter then announces the placing of another token on edge $\langle m, self[i] \rangle$. Therefore, when the previous value of the counter is invalidated in process *self*'s cache, the next value *self* reads allows it to remove the edge from *ToCheck*. This implies that there is exactly one invalidation of *self*'s copy of m's counter for every time *self* waits on that counter. Since transferring a new value to *self*'s cache is the only way another process communicates with it, no implementation of marked graph synchronization can use fewer cache invalidations. Hence, the optimized version of algorithm *FGAlg* is optimal with respect to caching when each counter is incremented by firing only one node.

If a node m's counter is incremented by nodes other than m, then there are writes to that counter that do not put a token on edge $\langle m, self[i] \rangle$. A process waiting for the token on that edge may read values of the counter written when firing those other nodes, leading to additional cache invalidations. Therefore, cache utilization is optimal only when we can let $Q = 1$. Of course, we can always assign counters to nodes to make Q equal 1.

We have been assuming a memory system that allows multiple processors to share read-only copies of a memory location. Some systems allow only a single processor to have a valid copy of a memory location. In that case, the algorithm can maintain multiple shared copies of a counter—one for each process that reads it—plus a local copy for the counter's writer. A process writes the counter by writing each of those copies separately. Because the counter is written by only a single process, this implementation is equivalent to the algorithm that uses just a single copy. Hence, the resulting algorithm makes optimal use of caching if $Q = 1$.

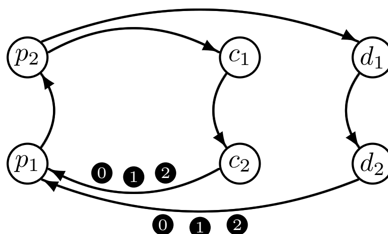
4 Keeping Track of Buffers

In the producer/consumer examples of Section 2, certain subgraphs of the marked graph represent buffer pools. A token on each of those subgraphs represents a buffer in the pool. The two-consumer example (5) has one pool of three buffers. A token on $\langle c_1, c_2 \rangle$ and one on $\langle d_1, d_2 \rangle$ could represent the same buffer. This means that the computations represented by those two edges can only read from a buffer.[4] In the two-producer example (6), there are two buffer pools—one with three buffers and one with two. A single token on $\langle c_1, c_2 \rangle$ represents one buffer from each pool. This means that the

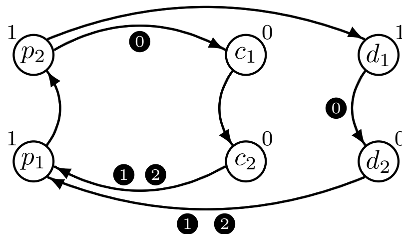
[4]The two computations could write to different parts of the buffers, but that is equivalent to splitting each buffer in two and having two separate buffer pools.

computation represented by the edge accesses a buffer from each pool.

Using pools of more than a single buffer allows pipelining of a dataflow computation, so processes can compute concurrently even though they cannot concurrently access the same data at the same time. Increasing the number of buffers can prevent processes from having to wait because of variation in the time it takes to perform a computation. In order to implement a dataflow computation using buffer pools, we must determine which buffers are to be accessed by the computations. This means determining which buffers are represented by the token on the corresponding computation edge. We indicate how this is done using the two-consumer producer/consumer system as an example. Here is the subgraph of that marked graph that represents the buffer pool, where we have assigned buffer numbers to the tokens.



Any assignment of tokens to the edges of this subgraph that occurs when executing the complete marked graph can occur when executing this subgraph. (The converse is not true, since an execution of the complete graph can put only a single token on the three vertical edges.) Consider the following marking of this graph that can arise during an execution, where we have labeled each node with the number of the next buffer to pass through that node.



We call such an assignment β of a value $\beta[n]$ to each node n a *buffer assignment* for the marking μ. A buffer assignment β for a marking μ satisfies

$$\beta[m] = (\beta[n] + \mu[\langle m, n \rangle]) \,\%\, B \tag{18}$$

where B is the number of buffers in the pool. A buffer assignment β determines an assignment of buffer numbers to tokens in the obvious way,

assigning to the first token on an arc $\langle m, n \rangle$ the buffer number $\beta[n]$.

We define a *buffer marked graph* with B buffers to be a triple $\langle \Gamma, \mu_0, \beta_0 \rangle$ such that $\langle \Gamma, \mu_0 \rangle$ is a marked graph and β_0 is a buffer assignment for μ_0. It is not hard to see that we can find a buffer assignment that turns a marked graph into a buffer marked graph with B buffers iff every simple cycle in the marked graph $\langle \Gamma, \mu \rangle$ contains B tokens.

We define the result of firing node n in a buffer assignment β to equal the buffer assignment obtained from β by replacing $\beta[n]$ by $(\beta[n] + 1) \,\%\, B$. If β is a buffer assignment for μ and n is fireable in μ, then firing n in β produces a buffer assignment for the marking obtained by firing n in μ. We can therefore define an execution of a buffer marked graph to consist of a sequence of buffer marked graphs obtained by repeatedly firing arbitrarily chosen fireable nodes. To specify executions of a buffer marked graph, we modify algorithm *MGSpec* of Figure 1 by adding a variable β initialized to β_0 and adding the statement

$$\beta[self] := (\beta[self] + 1) \,\%\, B$$

to the body of the **while** loop.

To keep track of buffers in a dataflow computation defined by a marked graph, we can add a variable β for each buffer pool and increment $\beta[n]$ modulo the number of buffers whenever node n is fired, for each node n in the subgraph corresponding to the buffer pool. (Each counter $\beta[n]$ is accessed only by node n's process.) The computation corresponding to a computation edge $\langle m, n \rangle$ then uses buffer $\beta[n]$.

It is not necessary to add new counters. If we take N to be a multiple of $Q * B$, we can use the modulo N counters that implement the marked graph synchronization. For the algorithms with the array ct of unbounded counters, formula (9), which holds when n is the next node of its process to fire, implies that

$$\beta[n] = (\lceil bct(n) \rceil^Q / Q \;+\; \beta_0[n]) \,\%\, B$$

Number fact 4 then implies that, if $Q * B$ divides N, in algorithm *FGAlg* the computation corresponding to computation edge $\langle m, n \rangle$ uses buffer number

$$(\lceil bcnt(n) \rceil^Q / Q \;+\; \beta_0[n]) \,\%\, B$$

5 Conclusion

We have shown how to implement an arbitrary dataflow computation with multiple threads using shared buffers. Threads are synchronized by reads

and writes to shared memory, using memory barriers where needed.

We know of two special cases of our algorithm that have been published—the producer/consumer example in Section 3.4 and algorithm *Barrier*3 in Section 3.5. An instance of Section 4's method for keeping track of buffers in the producer/consumer algorithm appears in [11]. A method of implementing an arbitrary process marked graph using even weaker primitives than read and write appeared in [9]. However, that implementation is not designed for shared-memory multiprocessors and is mainly of theoretical interest.

If multiple buffering is not used, so each buffer pool contains only a single buffer, then dataflow computations can be implemented using conventional barrier synchronization. However, the use of multiple buffers permits greater concurrency when there is variance in the amount of time needed to perform a computation. For example, if the times to perform the *produce* and *consume* operations have Poisson distributions with the same mean and standard deviation, then the throughput of a producer/consumer system with B buffers approaches twice that of a system with a single buffer as B goes to infinity.

The performance of our method for synchronizing dataflow computations will depend on the multiprocessor hardware. Hardware message passing primitives are likely to provide more efficient synchronization. However, shared memory may be better for passing the computation's data from one process to the next.

In many if not most cases, we expect our implementation to perform better than ones based on special hardware-supported synchronization primitives such as locks and atomic test-and-set operations. Reads and writes are less expensive than those more powerful synchronization operations and our algorithm can provide optimal caching performance, requiring the minimum number of cache invalidations.

We have compared algorithms *Barrier*1–3 of Section 3.5 with a standard barrier algorithm using a fetch-and-decrement operation—namely, the "sense-reversing centralized barrier" of [12, Figure 8]. We tested the algorithms on a 4-processor Pentium 4 system using 4 and 8 threads—in the latter case, using the Pentium's hyper-threading feature. The performance of the three algorithms using only reads and writes were within 15% of one another. The fetch-and-decrement algorithm was about twice as slow with 4 threads and 3 times as slow with 8 threads. The superiority of *Barrier*3 over algorithms based on more complicated operations was also reported by Hill and Skillicorn [4].

Dataflow is a natural way to describe many data processing tasks. It

may provide a good way to program the coming generation of multiprocessor chips. Our algorithm can efficiently implement dataflow computations on a shared-memory architecture.

Acknowledgements

Dahlia Malkhi helped me understand the barrier synchronization algorithms and devised Algorithm *Barrier*2. Neill Clift helped me understand the need for memory barriers. He and John Rector did the performance testing of the barrier synchronization algorithms that is described in the conclusion.

References

[1] Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Digital Press, Boston, third edition, 1998.

[2] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(6):511–523, December 1971.

[3] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*. Academic Press, New York, 1968.

[4] Jonathan M. D. Hill and David B. Skillicorn. Practical barrier synchronisation. In *6th Euromicro Workshop on Parallel and Distributed Processing*. IEEE, Computer Society Press, 1998.

[5] Leslie Lamport. The +CAL algorithm language. URL `http://research.microsoft.com/users/lamport/tla/pluscal.html`. The page can also be found by searching the Web for the 25-letter string obtained by removing the "`-`" from `uid-lamportpluscalhomepage`.

[6] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[7] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[8] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.

[9] Leslie Lamport. Arbiter-free synchronization. *Distributed Computing*, 16(2–3):219–237, September 2003.

[10] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. Also available on the Web via a link at `http://lamport.org`.

[11] Leslie Lamport and Fred B. Schneider. Pretending atomicity. Research Report 44, Digital Equipment Corporation, Systems Research Center, May 1989.

[12] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):272–314, February 1991.

[13] Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets.* Springer-Verlag, 1998.

Appendix: Formal Definitions

We now formalize the informal mathematical definitions of Section 3. This is done in the TLA$^+$ specification language [10]. Most of the TLA$^+$ constructs used here have already been introduced or are sufficiently close to standard mathematics that they should be easily understood by a mathematically sophisticated reader. We briefly explain the rest.

The definitions appear in a module named *Definitions*. The module starts with an EXTENDS statement that imports some common mathematical operators from the standard modules *Naturals* and *Sequences*.

——— MODULE *Definitions* ———

EXTENDS *Naturals*, *Sequences*

Most of the nonstandard notation introduced in the *Naturals* and *Sequences* modules is explained in Section 3.1.1. Not mentioned there is that *Nat* is the set of natural numbers.

We now define some operators that are needed later on. First is the operator *SeqSum* that sums a sequence of numbers, so

$$SeqSum(\langle s_1, \ldots, s_n \rangle) \;=\; \sum_{i=1}^{n} s_i \tag{19}$$

for any sequence $\langle s_1, \ldots, s_n \rangle$ of numbers. Because TLA$^+$ allows recursive definitions only of functions, we define the sum (19) in terms of a function *sum* with domain $0 \,..\, n$ such that $sum[i]$ equals $\sum_{j=1}^{i} s_i$.

$$
\begin{array}{l}
SeqSum(seq) \triangleq \\
\quad \text{LET } sum[i \in 0 \,..\, Len(seq)] \triangleq \text{IF } i = 0 \text{ THEN } 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } seq[i] + sum[i-1] \\
\quad \text{IN } \; sum[Len(seq)]
\end{array}
$$

We next define $SeqToSet(\sigma)$ to be the set of elements contained in sequence σ and $Min(S)$ to be the minimum element of a set S of numbers, if S has one. The TLA$^+$ expression CHOOSE $x \in S : exp$ equals an arbitrary element x in S satisfying exp, if such an element exists.

$$SeqToSet(seq) \triangleq \{seq[i] : i \in 1 \,..\, Len(seq)\}$$

$$Min(S) \triangleq \text{CHOOSE } s \in S : \forall t \in S : s \leq t$$

We next declare all the constants that we have assumed. We write G, $mu0$, and Pi instead of Γ, μ_0, and Π.

CONSTANTS N, G, $mu0$, Pi, $CtrOf$, $Incr$

The arithmetic operators introduced in Section 3.1.2 are defined as follows, where we write $ceiling(k, Q)$ instead of $\lceil k \rceil^Q$.

$$a \oplus b \triangleq (a + b) \,\%\, N$$

$$a \ominus b \triangleq (a - b) \,\%\, N$$

$$ceiling(k, Q) \triangleq \text{CHOOSE } i \in \{k + j : j \in 0 \,..\, (Q - 1)\} : i \,\%\, Q = 0$$

We now formalize the concepts introduced above for the graph G. In addition to the operator $InEdges$, we define the analogous operator $OutEdges$, the set $Paths$ of all paths in G, the set $PathsFromTo(m, n)$ of paths from m to n, the predicate $IsStronglyConnectedGraph$ asserting that G is a strongly connected directed graph, and the set $SimpleCycles$ of all simple cycles of G.

$$InEdges(n) \triangleq \{e \in G.edges : e[2] = n\}$$

$$OutEdges(n) \triangleq \{e \in G.edges : e[1] = n\}$$

$$
\begin{array}{l}
Paths \triangleq \{pi \in Seq(G.nodes) : \land\ Len(pi) > 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ \forall i \in 1 \,..\, (Len(pi)-1) : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle pi[i], pi[i+1] \rangle \in G.edges\}
\end{array}
$$

$$PathsFromTo(m, n) \triangleq \{pi \in Paths : (pi[1] = m) \land (pi[Len(pi)] = n)\}$$

$$
\begin{array}{l}
IsStronglyConnectedDirectedGraph \triangleq \\
\quad \land\ G.nodes \neq \{\} \\
\quad \land\ G.edges \subseteq G.nodes \times G.nodes \\
\quad \land\ \forall m, n \in G.nodes : PathsFromTo(m, n) \neq \{\}
\end{array}
$$

$$SimpleCycles \triangleq \{pi \in Paths : \land \langle pi[Len(pi)], pi[1] \rangle \in G.edges \\ \land \forall i, j \in 1 \,..\, Len(pi) : \\ (i \neq j) \Rightarrow (pi[i] \neq pi[j])\}$$

Next come definitions for the marked graph $\langle G, mu0 \rangle$ (originally called $\langle \Gamma, \mu_0 \rangle$). We define *Dist* to be the distance function δ_{μ_0} of Section 3.2, first defining $PathLen(\pi)$ to be the length of the path π in that distance measure. We also define the operator *Fire* and the assertion *IsLiveMarkedGraph* that $\langle G, mu0 \rangle$ is a live marked graph. The latter definition uses the fact that a marked graph is live iff every cycle contains a token.

$$PathLen(pi) \triangleq \\ SeqSum(\,[i \in 1 \,..\, (Len(pi) - 1) \mapsto mu0[\langle pi[i], pi[i+1] \rangle]\,]\,)$$

$$Dist(m, n) \triangleq Min(\{PathLen(pi) : pi \in PathsFromTo(m, n)\})$$

$$Fire(n, mu) \triangleq \\ [e \in G.edges \mapsto \text{IF } e \in InEdges(n) \setminus OutEdges(n) \\ \text{THEN } mu[e] - 1 \\ \text{ELSE IF } e \in OutEdges(n) \setminus InEdges(n) \\ \text{THEN } mu[e] + 1 \\ \text{ELSE } mu[e]\,]$$

$$IsLiveMarkedGraph \triangleq \\ \land IsStronglyConnectedDirectedGraph \\ \land mu0 \in [G.edges \rightarrow Nat] \\ \land \forall pi \in SimpleCycles : \\ PathLen(pi) + mu0[\langle pi[Len(pi)], pi[1] \rangle] > 0$$

The following definitions are related to process marked graph $\langle G, mu0, Pi \rangle$, starting with the assertion C0 that it is indeed a process marked graph. We also define $ProcOf(n)$ to be the process containing node n, *SyncEdges* to be the set of synchronization edges, and $SinEdges(n)$ and $SOutEdges(n)$ to be the sets of in- and out-edges of node n.

$$C0 \triangleq \land IsLiveMarkedGraph \\ \land Pi \subseteq SimpleCycles \\ \land \forall n \in G.nodes : \exists pi \in Pi : n \in SeqToSet(pi) \\ \land \forall pi1, pi2 \in Pi : \\ (pi1 \neq pi2) \Rightarrow (SeqToSet(pi1) \cap SeqToSet(pi2) = \{\}) \\ \land \forall pi \in Pi : \land PathLen(pi) = 0 \\ \land mu0[\langle pi[Len(pi)], pi[1] \rangle] = 1$$

$$ProcOf(n) \triangleq \text{CHOOSE } pi \in Pi : n \in SeqToSet(pi)$$

$$SyncEdges \triangleq \{e \in G.edges : ProcOf(e[1]) \neq ProcOf(e[2])\}$$

$$SInEdges(n) \triangleq InEdges(n) \cap SyncEdges$$

$$SOutEdges(n) \triangleq OutEdges(n) \cap SyncEdges$$

We next define *IncrSum* and Q, first defining the set *Ctrs* of all counters. We also define properties C1–4. For our algorithms to be correct in the trivial case of a single process, we need to add the requirement $Q > 0$. (This is implied by C0 and C2 if there is more than one process.) We add the requirement to C3. We also add to C4 the explicit assumption that N is a natural number.

$$Ctrs \triangleq \{CtrOf[n] : n \in G.nodes\}$$

$$\begin{array}{l} IncrSum(c) \triangleq \\ \quad \text{LET } pic \triangleq ProcOf(\text{CHOOSE } n \in G.nodes : c = CtrOf[n]) \\ \quad \qquad seq \triangleq [i \in 1 \,..\, Len(pic) \mapsto \text{IF } CtrOf[pic[i]] = c \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } Incr[pic[i]] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } 0\,] \\ \quad \text{IN } \quad SeqSum(seq) \end{array}$$

$$Q \triangleq IncrSum(\text{CHOOSE } c : c \in Ctrs)$$

$$\begin{array}{l} C1 \triangleq \forall\, m, n \in G.nodes : \\ \qquad\qquad (CtrOf[m] = CtrOf[n]) \Rightarrow (ProcOf(m) = ProcOf(n)) \end{array}$$

$$\begin{array}{l} C2 \triangleq \land\; Incr \in [G.nodes \to Nat] \\ \qquad\quad \land\; \forall\, n \in G.nodes : (SOutEdges(n) \neq \{\}) \Rightarrow (Incr[n] > 0) \end{array}$$

$$\begin{array}{l} C3 \triangleq \land\; Q > 0 \\ \qquad\quad \land\; \forall\, c \in Ctrs : IncrSum(c) = Q \end{array}$$

$$\begin{array}{l} C4 \triangleq \land\; (N \in Nat) \land (N \,\%\, Q = 0) \\ \qquad\quad \land\; \forall\, n \in G.nodes : \\ \qquad\qquad\quad \forall\, e \in SInEdges(n) : \\ \qquad\qquad\qquad \text{LET } m \triangleq e[1] \\ \qquad\qquad\qquad \text{IN } \quad N > Q * Dist(n, m) + Q * mu0[\langle m, n \rangle] \end{array}$$

Finally, we define the operator $cnt0$.

$$\begin{array}{l} cnt0(n) \triangleq \\ \quad \text{LET } pi \triangleq ProcOf(n) \\ \quad \qquad k \triangleq \text{CHOOSE } i \in 1 \,..\, Len(pi) : n = pi[i] \\ \quad \qquad seq \triangleq [i \in 1 \,..\, (k-1) \mapsto \text{IF } CtrOf[pi[i]] = CtrOf[n] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } Incr[pi[i]] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } 0\,] \\ \quad \text{IN } \quad SeqSum(seq) \end{array}$$

The operator $CtTest$, used in algorithm $PMGAlg3$, would be defined within the algorithm just like $CntTest$.