A GENERIC SIMULATION
OF COUNTING NETWORKS

By

Eric Neil Klein

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF COMPUTER SCIENCE

Approved:

David R. Musser, Thesis Adviser

Costas Busch, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

July 2003
(For Graduation August 2003)
CONTENTS

LIST OF TABLES ......................................................... iv
LIST OF FIGURES ....................................................... v
ACKNOWLEDGMENT ....................................................... vii
ABSTRACT ................................................................. viii

1. Introduction .............................................................. 1
   1.1 The Counting Problem ............................................ 1
   1.2 Balancing Networks .............................................. 1
   1.3 Counting Networks ............................................... 2
   1.4 Counting Network Implementations .............................. 3
   1.5 Contributions .................................................. 4
   1.6 Outline ......................................................... 5

2. Counting Network Constructions ...................................... 6
   2.1 Bitonic Counting Network ....................................... 6
   2.2 BM Counting Network .......................................... 9
   2.3 Periodic Counting Network .................................... 11

3. Distribution ........................................................... 15
   3.1 Random Distribution .......................................... 15
   3.2 Trivial Distribution .......................................... 15
   3.3 Level Distribution ........................................... 17
   3.4 Longest-Run Distribution ..................................... 18

4. Simulation Model ....................................................... 20
   4.1 Network .......................................................... 20
   4.2 Balancers ...................................................... 20
   4.3 Counting Variables .......................................... 21
   4.4 Sources ....................................................... 22
5. Simulation Results .................................................. 23
   5.1 Distribution Method ............................................ 23
   5.2 Network Size ..................................................... 26
   5.3 Network Construction .......................................... 29
   5.4 Parallelism ....................................................... 32

6. Counting Network Implementation .................................. 35
   6.1 Description ....................................................... 35
   6.2 Performance ....................................................... 35

7. Software Details .................................................... 41
   7.1 Properties File .................................................. 41
   7.2 Generators ....................................................... 42
   7.3 Distributors ..................................................... 43
   7.4 Visualizers ...................................................... 43
   7.5 Execution ....................................................... 45
       7.5.1 COST Simulation ............................................ 46
       7.5.2 Socket Implementation .................................... 46
   7.6 Examples ....................................................... 47

8. Conclusions .......................................................... 49

LITERATURE CITED .................................................... 51
# LIST OF TABLES

2.1 Some properties of bitonic networks ........................................ 13  
2.2 Some properties of BM networks ............................................. 13  
2.3 Some properties of periodic networks ..................................... 14  
5.1 The optimal size bitonic counting network for a range of token loads . 28  
7.1 The metrics output by our simulator ....................................... 45
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The bitonic 8 network</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>The merger 8 balancing network</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>The bitonic 32 counting network</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>The BM 8 input / 48 output counting network</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>The $C(4, 8)$ network</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>The $M(8, \delta)$ network</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>The block 8 balancing network</td>
<td>12</td>
</tr>
<tr>
<td>2.7</td>
<td>The periodic 16 counting network</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>The bitonic 8 network distributed among 4 hosts using the random distribution algorithm</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>The bitonic 8 network distributed among 4 hosts using the trivial distribution algorithm</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>The bitonic 8 network distributed among 4 hosts using the level distribution algorithm</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>The bitonic 8 network distributed among 4 hosts using the longest-run distribution algorithm</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparing various balancer distribution algorithms in the bitonic 16 counting network</td>
<td>24</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparing various balancer distribution algorithms in the bitonic 16 counting network (faster network)</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>Comparing various balancer distribution algorithms in the bitonic 16 counting network (slower network)</td>
<td>27</td>
</tr>
<tr>
<td>5.4</td>
<td>Determining the optimal size bitonic counting network to use on different size networks</td>
<td>28</td>
</tr>
<tr>
<td>5.5</td>
<td>Determining the optimal size bitonic counting network to use on different size networks (lower contention)</td>
<td>30</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparing various counting network constructions</td>
<td>31</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparing various counting network constructions (faster network)</td>
<td>32</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.8</td>
<td>Comparing various counting network constructions (slower network)</td>
<td>33</td>
</tr>
<tr>
<td>5.9</td>
<td>Comparing various counting network constructions (slower balancer)</td>
<td>34</td>
</tr>
<tr>
<td>5.10</td>
<td>Determining the effect that parallelism in hosts has on the performance of counting networks</td>
<td>34</td>
</tr>
<tr>
<td>6.1</td>
<td>Comparing various bitonic counting network sizes</td>
<td>36</td>
</tr>
<tr>
<td>6.2</td>
<td>Comparing various BM counting network sizes</td>
<td>37</td>
</tr>
<tr>
<td>6.3</td>
<td>Comparing various periodic counting network sizes</td>
<td>38</td>
</tr>
<tr>
<td>6.4</td>
<td>Comparing various counting network constructions of input width 8</td>
<td>39</td>
</tr>
<tr>
<td>6.5</td>
<td>Comparing various counting network constructions of input width 16</td>
<td>40</td>
</tr>
<tr>
<td>7.1</td>
<td>The bitonic 4 network</td>
<td>42</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENT

I would like to thank my classmates, Matthew Abrams and Renee Guhde, for working with me on our original MPI implementation of counting networks.
ABSTRACT

Counting networks, a type of network consisting of a set of interconnected balancers, provide a low contention solution to the counting problem. One way in which counting networks can be implemented in a distributed environment is by modeling the balancers as independent units of execution. These balancers pass tokens over a pre-configured set of connections in order to communicate with each other. We present four novel algorithms for distributing balancers among hosts on a network. We create a simulation for counting networks in order to understand how the construction, the size, and the distribution methods of counting networks correlate to their overall performance. Finally, we provide a generic implementation that will run a counting network in a distributed environment.
CHAPTER 1
Introduction

1.1 The Counting Problem

Solving the counting problem is a common requirement of many fundamental computer algorithms such as implementing a queue using a random access array in memory. In the counting problem, sequential values must be assigned to multiple asynchronous processes. A solution to the counting problem can be used to create a queue by independently assigning the memory addresses of the head and the tail of the queue. The counting problem can be trivially solved using a single shared Fetch&Increment variable, but under high contention, this simple solution performs poorly. This solution requires all processes to synchronize on a single variable, thereby creating a hot spot in the program. If we were to use this simple solution to implement a distributed queue in shared memory, performance would degrade as access to the queue increased. [1] introduces counting networks, a class of concurrent data structures that solve the counting problem. Counting networks distribute the synchronizations among multiple locations thereby decreasing the contention at any single variable.

1.2 Balancing Networks

Counting networks are a subset of balancing networks, a type of network that balances its inputs among its outputs. Balancing networks are composed of a set of interconnected balancers. Wires form the interconnections between balancers. A balancer can be thought of as a toggle that receives tokens from a set of input wires and sends them out over a set of output wires. A balancer with $n$ output wires will forward the $i$th token that it receives to its $i \mod n$ output wire.

Figure 1.1 depicts the bitonic 8 counting network, a type of balancing network. In this common representation, horizontal lines represent wires and vertical lines represent balancers. The attachments between wires and balancers are denoted by dots, and the tokens flow through the network from the left side to the right side.
We call the set of unconnected wires on the left and right, *input wires* and *output wires*, respectively. Tokens are inserted into the balancing network on the input balancers and propagate through the network to the output wires.

Some basic properties of balancing networks are described using the following terms. The *width* of a balancing network is the number of wires that it contains. Traditionally, balancing networks have the same number of input and output wires, but we will see a class of counting networks, introduced by [2], that benefit from having more output wires than input wires. We define the *level* of a balancer recursively as one more than the maximum level of any balancer from which it receives tokens. A balancer that is only connected to input wires of the counting networks is on level 1. The *depth* of a network is the total number of levels that it contains.

### 1.3 Counting Networks

A counting network is a type of balancing network that has the step property. The step property asserts that at any time the counting network is in a quiescent state, a state in which there are no tokens in the network, the difference between the number of tokens received by any two output wires will be at most 1, and no output wire will receive more tokens than any other output wire that is above it.

In order to use a counting network to solve the counting problem, a *counting variable* is attached to each output wire. These counting variables are similar to shared *Fetch&Increment* counters, with the exception that they are incremented by
more than 1 each time they are accessed. Given a counting network with $n$ output wires, the counting variables are initialized with the values $0, 1, \ldots, n-1$. The value 0 is assigned to the topmost counting variable and the value $n-1$ is assigned to the bottommost counting variable. Each time a token propagates to an output wire, the output wire’s counting variable is accessed and incremented by $n$. Accordingly, no two counting variables will ever have the same value, and because of the step property, the counting variables will assign values sequentially.

1.4 Counting Network Implementations

Counting networks can be implemented in hardware as synchronous circuits or in software by programming balancers as sequential routines and token passing as either message passing or shared variable updating. In this thesis, we restrict our attention to software implementations using message passing over networks connecting multiple hosts. The purpose of using a balancing network over a single conventional shared counter is to reduce the contention caused by synchronizing processes on a single variable in memory. The notion of contention in distributed data structures, such as counting networks, is explained in [7]. A balancer is unable to handle multiple tokens concurrently, similarly to how a unit of computer hardware in a pipeline can only perform operations in a serial manner. Just as additional operations must stall the hardware pipeline, all other tokens that arrive while the balancer is busy must be queued. Contention in a counting network is defined as the number of stall steps during which a balancer is unable to handle any additional tokens because it is busy handling the first token.

In our model of counting networks we have two functions that must be performed: the balancers must route tokens that they receive, and the counters must assign values to the tokens that they receive. We put each of the functional units, the balancers and the counters, in their own thread of execution so that they can execute simultaneously. Each functional unit is assigned to a host that continuously executes its predefined function, either balancing or counting. We use multiple hosts that communicate by forwarding tokens to each other over a network in order to distribute the amount of work among multiple computers. Since multiple threads
can be executing on the same host, tokens may not need to traverse the physical network when they are passed between functional units.

We developed our counting network simulator using the Component-Oriented Simulation Toolkit (COST) [4], which provides a generic framework for performing discrete event simulations [3]. It allows for the creation of independent components that communicate by sending messages over a set of ports. The ports define which components are connected and the type of messages that can be sent on each connection. Additionally, the components have timers that can schedule events to be triggered at discrete times.

We also developed an actual distributed counting network implementation in which the balancers and the counters execute in independent threads on multiple computers. The balancers and counters communicate over the network using internet sockets. The implementation allows client processes to send tokens of a predefined structure to a set of balancers in the network. The client includes an internet address in its token, and after the token traverses the counting network, one of the counters will return this token to the client with an assigned value.

1.5 Contributions

We provide four novel algorithms used to distribute the balancers in a counting network among the hosts that can perform their functions. Using our counting network simulation, we provide experimental results that show that our longest-run distribution algorithm outperforms our other balancer distribution algorithms in most cases. We describe adversary conditions in which this distribution algorithm is outperformed by other algorithms. Additionally, we use our simulation in order to determine how certain experimental variables affect the performance of counting networks. We also show that the BM counting network usually outperforms the bitonic counting network and explain circumstances that cause the opposite behavior to be observed. We provide a distributed implementation of counting networks using sockets and give some evidence to show that its performance is similar to that seen in our simulation. We also provide utilities for visualizing counting networks and visualizing the distribution of their balancers among hosts. These utilities are used
to produce many of the illustrations in this paper.

1.6 Outline

In chapter 2, we give recursive algorithms for constructing the three counting networks that are used in this paper. In chapter 3, we explain the four algorithms that we use for distributing balancers among hosts. Chapter 4 describes how we model our counting network simulation using the COST system. Chapter 5 discusses the experimental results of our counting network simulation. In chapter 6, we discuss our distributed counting network implementation and describe its performance. Chapter 7 provides details about how we implemented the software used in this thesis. Finally, we summarize our contributions in chapter 8.
CHAPTER 2
Counting Network Constructions

In this chapter, we give recursive constructions for the bitonic counting network, the BM counting network, and the periodic counting network.

2.1 Bitonic Counting Network

The bitonic counting network has an isomorphic structure to the bitonic sorting network [1]. This network can only have a width of $2^i$ with $i > 0$. The following construction of the bitonic counting network, including figure 2.1 is copied verbatim from [1].

Define the width $w$ balancing network $\text{MERGER}[w]$ as follows. It has two sequences of inputs of length $w/2$, $x$ and $x'$, and a single sequence of outputs $y$, of length $w$. $\text{MERGER}[w]$ will be constructed to guarantee that in a quiescent state where the sequences $x$ and $x'$ have the step property, $y$ will also have the step property.

We define the network $\text{MERGER}[w]$ inductively (see example in figure 2.1). Since $w$ is a power of 2, we will repeatedly use the notation $2k$ in place of $w$. When $k$ is equal to 1, the $\text{MERGER}[2k]$ network consists of a single balancer. For $k > 1$, we construct the $\text{MERGER}[2k]$ network

![Diagram](image)

Figure 2.1: The merger 8 balancing network
with input sequences $x$ and $x'$ from two MERGER[$k$] networks and $k$ balancers. Using a MERGER[$k$] network we merge the even subsequence $x_0, x_2, \ldots, x_{k-2}$ of $x$ with the odd subsequence $x'_1, x'_3, \ldots, x'_{k-1}$ of $x'$ (i.e., the sequence $x_0, \ldots, x_{k-2}, x'_1, \ldots, x'_{k-1}$ is the input to the MERGER[$k$] network) while with a second MERGER[$k$] network we merge the odd subsequence of $x$ with the even subsequence of $x'$. Call the outputs of these two MERGER[$k$] networks $z$ and $z'$. The final stage of the network combines $z$ and $z'$ by sending each pair of wires $z_i$ and $z'_i$ into a balancer whose outputs yield $y_{2i}$ and $y_{2i+1}$.

The MERGER[$w$] network consists of log $w$ layers of $w/2$ balancers each. MERGER[$w$] guarantees the step property on its outputs only when its inputs also have the step property— but we can ensure this property by filtering these inputs through smaller counting networks. We define BITONIC[$w$] to be the network constructed by passing the outputs from two BITONIC[$w/2$] networks into a MERGER[$w$] network, where the induction is grounded in the BITONIC[1] network which contains no balancers and simply passes its input directly to its output. This construction gives us a network consisting of $\left\lceil \frac{\log w + 1}{2} \right\rceil$ layers each consisting of $w/2$ balancers.

The network thus constructed is isomorphic to that produced by the method in [5] (which relies on a different recursive construction). For example, the network on the left in figure 2.1 is isomorphic to the one on the right, as can be seen if the recursions in the left are expanded out, the wires are all straightened out, and it is noted that the (top or bottom) order of inputs to a balancer is not significant. In constructing bitonic networks for both our simulations and our actual networks, we used the construction in [5] rather than the one above from [1].

Figure 2.2 depicts the bitonic 32 network. The overall depth of the bitonic counting network of width $n$ is exactly $(\log n)(\log n + 1)/2$ [5].
Figure 2.2: The bitonic 32 counting network

Figure 2.3: The BM 8 input / 48 output counting network
2.2 BM Counting Network

The Busch-Mavronicolas (BM) counting network presented in [2] can have an output width that is greater than its input width. It has input width $t$ and output width $w$, where $t \leq w$, $t = 2^k$ and $w = p2^l$. Like the bitonic network, the BM network of input width $t$ has a depth of $O(\log^2 t)$, but unlike the bitonic network, the output width can be larger than the input width without affecting the depth of the network. By increasing the output width, contention is decreased on the right side of the network. Figure 2.3 depicts the BM counting network with an input width of 8 and an output width of 48.

The BM counting network is constructed recursively and includes three sequential stages, a set of 2-input, 2-output balancers in parallel, two smaller BM counting networks in parallel, and a bounded difference $\delta$-merging network. Let $C(t, w)$ denote the BM counting network with $t$ input wires, $x_0, \ldots, x_{t-1}$, and $w$ output wires, $y_0, \ldots, y_{w-1}$ (see example in figure 2.4 from [2]). $t$ and $w$ have the aforementioned restrictions, namely $t \leq w$, $t = 2^k$ and $w = p2^l$. The base case is $t = 2$; it is simply a 2-input, $w$-output balancer.

Stage one of the recursive case includes $t/2$ 2-input, 2-output balancers $b_0, \ldots, b_{t/2-1}$. The first input wire of balancer $b_i$ is connected to $x_{2i}$, and the second input wire of balancer $b_i$ is connected to $x_{2i+1}$.

Stage two of the recursive case includes two copies of $C(t/2, w/2)$, $C_0(t/2, w/2)$ and $C_1(t/2, w/2)$. The first output wire of balancer $b_i$ from stage one is connected

Figure 2.4: The $C(4, 8)$ network
Stage three of the recursive case is the bounded difference \( t/2 \)-merging network, \( M(w, t/2) \) (see example in figure 2.5 from [2]). The first \( w/2 \) input wires of \( M(w, t/2) \) are connected to the \( w/2 \) output wires of \( C_0 \) from stage two, and the second \( w/2 \) output wires of \( M(w, t/2) \) are connected to the \( w/2 \) output wires of \( C_1 \) from stage two. The \( w \) output wires of \( M(w, t/2) \) are the output wires of the BM counting network, \( C(t, w) \) [2].

The bounded difference \( \delta \)-merging network is constructed recursively in two sequential stages, a set of two smaller bounded difference \( \delta \)-merging networks in parallel and a set of 2-input, 2-output balancers in parallel. Let \( M(w, \delta) \) denote the bounded difference \( \delta \)-merging network where \( w = p2^l, \delta = 2^k, p \geq 1, l \geq 2, \) and \( 1 \leq k < l \). Let \( x_0, \ldots, x_{w/2-1} \) represent the sequence of the first half of the input wires to \( M \), and let \( y_0, \ldots, y_{w/2-1} \) represent the sequence of the second half of the input wires to \( M \). Let \( z_0, \ldots, z_{w-1} \) represent the sequence of the output wires of \( M \).

The base case is when \( \delta = 2 \); it consists of \( w/2 \) 2-input, 2-output balancers, \( b_0, \ldots, b_{w/2-1} \). The first input wire of balancer \( b_0 \) is connected to \( x_0 \), and the second input wire of balancer \( b_0 \) is connected to \( y_{w/2-1} \). The first output wire of \( b_0 \) is connected to \( z_0 \), and the second output wire of balancer \( b_0 \) is connected to \( z_{w-1} \).
Balancers \(b_1, \ldots, b_{w/2-1}\) are connected as follows. The first input wire of balancer \(b_i\) is connected to \(y_{i-1}\), and the second input wire of balancer \(b_i\) is connected to \(x_i\). The first output wire of \(b_i\) is connected to \(z_{2i-1}\), and the second output wire of balancer \(b_i\) is connected to \(z_{2i}\).

The first stage of the recursive case consists of two copies of \(M(w/2, \delta/2)\), \(M_0\) and \(M_1\). The first half of the inputs to \(M_0\) are connected to \(x_0, x_2, \ldots, x_{w/2-2}\), and the second half of the inputs to \(M_0\) are connected to \(y_0, y_2, \ldots, y_{w/2-2}\). The first half of the inputs to \(M_1\) are connected to \(x_1, x_3, \ldots, x_{w/2-1}\), and the second half of the inputs to \(M_1\) are connected to \(y_1, y_3, \ldots, y_{w/2-1}\).

The second stage of the recursive case consists of \(M(w, 2)\), the base case of the bounded difference \(\delta\)-merging network. The first half of the inputs of \(M(w, 2)\) are connected to the output sequence of \(M_0\), and the second half of the inputs of \(M(w, 2)\) are connected to the output sequence of \(M_1\). The output sequence of \(M(w, 2)\) is connected to \(z_0, \ldots, z_{w-1}\).

Similarly to how we constructed the bitonic network on the right in figure 2.1 from the network on the left in figure 2.1, we also straightened out the wires of the \(\delta\)-merging network in figure 2.5 when we constructed the BM counting network in figure 2.3.

For example, the network on the left in figure 2.1 is isomorphic to the one on the right, as can be seen if the recursions in the left are expanded out, the wires are all straightened out, and it is noted that the (top or bottom) order of inputs to a balancer is not significant.

### 2.3 Periodic Counting Network

The following construction of the periodic counting network, including figure 2.6, is copied verbatim from [1].

We start by defining chains and cochains, notions taken from [6]. Given a sequence \(x = \{x_i|i = 0, \ldots, n-1\}\), it is convenient to represent each index (subscript) as a binary string. A level \(i\) chain of \(x\) is a subsequence of \(x\) whose indices have the same \(i\) low-order bits. For example, the subsequence \(x^E\) of entries with even indices is a level 1
chain, as is the subsequence $x^O$ of entries with odd indices. The $A$-
cochain of $x$, denoted $x^A$, is the subsequence whose indices have the two
low-order bits 00 or 11. For example, the $A$-
cochain of the sequence
$x_0, \ldots, x_7$ is $x_0, x_3, x_4, x_7$. The $B$-
cochain $x^B$ is the subsequence whose
low-order bits are 01 or 10.

Figure 2.6: The block 8 balancing network

Figure 2.7: The periodic 16 counting network

Define the network $\text{BLOCK}[k]$ as follows. When $k$ is equal to 2, the
$\text{BLOCK}[k]$ network consists of a single balancer. The $\text{BLOCK}[2k]$ network
for larger $k$ is constructed recursively. We start with two $\text{BLOCK}[k]$
Table 2.1: Some properties of bitonic networks

<table>
<thead>
<tr>
<th>width</th>
<th>depth</th>
<th>balancers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>80</td>
</tr>
<tr>
<td>32</td>
<td>15</td>
<td>240</td>
</tr>
<tr>
<td>64</td>
<td>21</td>
<td>672</td>
</tr>
</tbody>
</table>

Table 2.2: Some properties of BM networks

<table>
<thead>
<tr>
<th>input width</th>
<th>output width</th>
<th>depth</th>
<th>balancers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>10</td>
<td>128</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>15</td>
<td>400</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>21</td>
<td>1152</td>
</tr>
</tbody>
</table>

networks $A$ and $B$. Given an input sequence $x$, the input to $A$ is $x^A$, and the input to $B$ is $x^B$. Let $y$ be the output sequence for the two subnetworks, where $y^A$ is the output sequence for $A$ and $y^B$ the output sequence for $B$. The final stage of the network combines each $y_i^A$ and $y_i^B$ in a single balancer, yielding final outputs $z_{2i}$ and $z_{2i+1}$. Figure 2.6 (from [1]) describes the recursive construction of a Block[8] network. The Periodic[2k] network consists of log $k$ Block[2k] networks joined so that the $i$th output wire of one is the $i$th wire of the next. Figure 2.7 is a Periodic[16] counting network.\(^1\)

\(^1\)Despite the apparent similarities between the layouts of the Block and Merger networks, there is no permutation of wires that yields one from the other.

Again, we straightened out the wires in the network on the left side of figure 2.6 when we constructed the network on the right side of figure 2.6.

The depth of the periodic counting network of width $n$ is $O(\log^2 n)$ [1]. Each BM counting network listed in table 2.2 has the same depth as a bitonic network in table 2.1 and the same number of balancers as a periodic network in table 2.3. We
Table 2.3: Some properties of periodic networks

<table>
<thead>
<tr>
<th>width</th>
<th>depth</th>
<th>balancers</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>36</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>32</td>
<td>25</td>
<td>400</td>
</tr>
<tr>
<td>64</td>
<td>36</td>
<td>1152</td>
</tr>
</tbody>
</table>

will use these similarities in comparing the performance of these counting network constructions.
CHAPTER 3
Distribution

The performance of a counting network is dependent on the distribution of the balancers among the hosts in the network. If two adjacent balancers are placed on the same host, there will be no network delay when these two balancers communicate. On the other hand, if adjacent balancers are placed on the same host, contention will result because that host will create a bottleneck. In order to observe the effect that the distribution of balancers has on the performance of the counting network, we developed four different algorithms for distributing the balancers among the hosts. We named these four algorithms random distribution, trivial distribution, level distribution and longest-run distribution.

3.1 Random Distribution

Random distribution assigns each balancer in the counting network to a random host. A side effect of this algorithm is that some hosts might not have any balancers assigned to them. Random distribution is used as a baseline to determine if an organized distribution has a beneficial or a deleterious effect on the performance of the counting network.

Figure 3.1 illustrates the bitonic 8 counting network distributed among 4 different hosts using the random distribution algorithm.

3.2 Trivial Distribution

We use recursive algorithms to generate the list of balancers in a counting network. Trivial distribution enumerates through this list of balancers in the order that it is generated by the algorithm and assigns each balancer to a host based on its position in the list. Given a counting network configuration and a network with $h$ hosts, balancer $i$ in the list of balancers will be assigned to host $i \mod h$. The same counting network can have its balancers generated in a different order depending on how the recursive algorithm is written. Unlike the random distribution algorithm,
Figure 3.1: The bitonic 8 network distributed among 4 hosts using the random distribution algorithm

Figure 3.2: The bitonic 8 network distributed among 4 hosts using the trivial distribution algorithm
the trivial distribution will satisfy the following property: The number of balancers in the counting network assigned to any two hosts in the network will differ by at most one. Figure 3.2 illustrates the bitonic 8 counting network distributed among 4 different hosts using the trivial distribution algorithm.

3.3 Level Distribution

Figure 3.3: The bitonic 8 network distributed among 4 hosts using the level distribution algorithm

Level distribution assigns balancers to hosts in a similar fashion as trivial distribution with a single exception. Instead of enumerating through the balancers solely in the order that they are generated by the recursive algorithm, level distribution enumerates through each level in the counting network individually and sequentially. The result of this algorithm is that the difference between the numbers of balancers assigned to any two hosts in a single level will be at most one. This distribution algorithm will also satisfy the property mentioned in the trivial distribution algorithm: the number of balancers in the counting network assigned to any two hosts in the network will differ by at most one. Figure 3.3 illustrates the bitonic 8 counting network distributed among 4 different hosts using the level distribution
3.4 Longest-Run Distribution

![Diagram of the bitonic 8 network distributed among 4 hosts using the longest-run distribution algorithm.]

Figure 3.4: The bitonic 8 network distributed among 4 hosts using the longest-run distribution algorithm

Longest-run distribution is the most complex algorithm that we introduce in this paper for distributing balancers in a counting network among hosts. This distribution algorithm attempts to minimize network delay by placing strings of adjacent balancers on the same host. Additionally, given a counting network with \( b \) balancers and a network with \( h \) hosts, the longest-run distribution algorithm assigns at most \( b/h \) balancers to any host. The longest-run distribution algorithm recursively finds the longest path through adjacent balancers that have not yet been assigned to any host and assigns these balancers to the same host. Figure 3.4 illustrates the bitonic 8 counting network distributed among 4 different hosts using the longest-run distribution algorithm.

We observed that when the longest-run distribution algorithm is used to assign balancers in the bitonic-n network to \( n/2 \) hosts, each internal balancer in the network will have exactly one preceding and one succeeding balancer on the same host.
Additionally, we observed that when the longest-run distribution algorithm is used to assign the \(n\)-input BM counting network to \(n/2\) hosts, each balancer in the network will have exactly half of its preceding balancers and half of its succeeding balancers on the same host. The observation about the BM counting network is just a generalization of the observation about the bitonic counting network because each internal balancer in the bitonic counting network has exactly two preceding balancers and two succeeding balancers.
CHAPTER 4
Simulation Model

Our simulation consists of four types of components: sources, balancers, counting variables and a network. The balancers and the counting variables provide the functionality of the counting network. The sources are the clients that inject tokens into the input wires of the counting network to request the sequential values that the counting network can provide. The counting variables assign values to the tokens and pass them back to the source that requested the value. The sources, balancers and counting variables all communicate through the network component that simulates network latency based on network traffic. In our simulation, there is an abstract concept of a host or a computer with a network interface. The sources, balancers and counting variables are each assigned to a host.

4.1 Network

The network is the most complex component in the simulation. It receives tokens and forwards them to their destination after a calculated delay. If the source and destination host are the same, the network does not delay the token. Otherwise, the delay is NETWORK_DELAY plus some amount of delay caused by contention. The contention is created in the following manner. A token traverses the network in two stages, each lasting one half of NETWORK_DELAY. During the first stage, other balancers on the source host are unable to communicate on the network. If another balancer on this host wants to communicate with the network, the request is queued. During the second stage, other balancers on the destination host are unable to communicate on the network. The network queues requests from other balancers on this host in a similar manner.

4.2 Balancers

Each balancer has a list of destinations to which it sends tokens that it receives. The balancer receives tokens from the network and sets the destination of the token
to the next balancer in the list. It forwards the token back on the network after a calculated delay. The delay is based on a base delay, the number of balancers on the same host that are currently handling tokens, and a parallelism factor that tells how well each host can execute multiple threads in parallel. If the base delay is BALANCER_DELAY, the parallelism factor is PARALLELISM, and n balancers on the same host are handling tokens, we set the delay to BALANCER_DELAY \cdot (\text{PARALLELISM} + (1 - \text{PARALLELISM})n). This function for calculating the actual delay that a balancer takes to process a token has the following properties.

If only a single balancer on a host is processing a token, the actual delay that the balancer waits to forward the token is BALANCER_DELAY. If a second token arrives while the balancer is still processing the first token, the worst case delay for the second token should be 2 \cdot BALANCER_DELAY. The first BALANCER_DELAY would be the maximum amount of time required to finish processing the original token, and the second BALANCER_DELAY would be the time to process the second token. This would occur if the tokens were processed sequentially with no parallelism, and thus, PARALLELISM is 0. If a second token arrives while the balancer is still processing the first token, the best case delay for the second token should be BALANCER_DELAY. This would occur if both tokens are processed completely in parallel, and thus, PARALLELISM is 1. We used .75 as the parallelism factor for most of our experiments. Therefore, our function will assign the second token a delay of (.75 + (1 - .75)2) \cdot BALANCER_DELAY or 1.25 \cdot BALANCER_DELAY. We chose this delay model because modern computing machinery is able to benefit by running processes in parallel. Multiprocessing allows additional processes to make progress while a first process is waiting for events such as input / output to complete in other computer hardware.

The balancers are a source of contention. A balancer cannot handle multiple tokens in parallel. While it is handling a token, it queues additional tokens.

4.3 Counting Variables

Like the balancers, the counting variables receive tokens from the network. In a counting network with n counting variables, each counting variable has a unique
initial value, \( m \), between 0 and \( n - 1 \). It assigns this initial value to the first token that it receives. It also has an increment value equal to the number of counting variables in the counting network. If \( m \) is the initial value and \( n \) is the increment value of the counting variable, it assigns the value \( m + i \cdot n \) to the \( i \)th token that it receives. The counting variable then forwards the token back to the network so that it can be returned to the source that requested a value.

### 4.4 Sources

The source components simply create tokens and put them on the network with a certain frequency. We refer to the inverse of this frequency, or the period between which two tokens are put on the network, as the `SOURCE_DELAY`. A random host in the simulation is chosen to be the source of each token, and the first hop of the token is set to a random input balancer. In addition to the hosts that contain balancers and counting variables, our simulation also consists of additional hosts that only make requests to the counting network. We refer to the total number of hosts in the simulation as `HOST_COUNT`.
CHAPTER 5
Simulation Results

There are many variables that account for the performance of a counting network. Our simulation allows one or more of these parameters to be varied so that the possible effects that this has on the results can be observed. Due to the myriad of interactions between elements of a counting network, it is not feasible to find correlations that encompass all of the variables. As an example of the utility of our simulation suite, we present experiments that vary only a few parameters and hold the rest constant. This enables us to graph the results and observe correlations. It must be pointed out that when other parameters are varied, the correlations might not exist and could even be contradicted.

5.1 Distribution Method

Given a counting network, we would like to find the optimal algorithm to distribute it among a set of hosts in a network. Our first experiment compares the performance of distributing the 80 balancers of the bitonic 16 network over 20 hosts using four different distribution algorithms, random distribution, trivial distribution, level distribution and longest-run distribution. We use random distribution as a baseline and compare the organized algorithms to this baseline in order to see if each distribution has a beneficial or detrimental effect on the performance. The BALANCER_DELAY parameter is set to 10, and the NETWORK_DELAY parameter is set to 100. The PARALLELISM parameter is set to 75%. We vary the SOURCE_DELAY over a range of 1 to 100 in the experiment in order to change the token load on the counting network. A total time of $1 \times 10^6$ units are allotted in order to process up to 1000 tokens. The results of our experiment are graphed in figure 5.1. The x axis of the graph represents the source delay; as the source delay increases, the load on the balancing network decreases.

A first trivial observation is that all of the functions in the graph are monotonically decreasing. This corresponds to the fact that the performance of counting
networks diminishes under increased contention. The slope of all the functions is clearly negative for a period, but at a certain point in each function, the slope levels off, and the line appears to become horizontal. The point at which the slope levels off corresponds to the token load that the counting network can handle with a negligible effect due to contention. A crucial goal in choosing a counting network is to choose a construction and distribution that can support the load requirements for the desired application. This will prevent subjecting the counting network to periods of high contention and avoid the poor performance that would result.

The level distribution algorithm performs significantly worse than the random baseline while the trivial and longest-run distribution algorithms perform significantly better than the baseline. The longest-run distribution algorithm assigns adjacent balancers to the same hosts in order to decrease the amount of network
delay to which each token is subjected. The tradeoff is that we increase the amount of contention on the hosts since tokens remain on the same host for a sequence of balancer hops. The longest-run distribution algorithm performs better than all the other algorithms except when subjected to the highest amounts of contention. When the source delay is less than 3, the trivial distribution algorithm performs slightly better than the longest-run distribution algorithm.

In our simulation, there are three parameters in units of time, SOURCE_DELAY, NETWORK_DELAY and BALANCER_DELAY. The parameters have no actual units, such as seconds, so they are only significant in relation to each other. Our previous experiment varied the source delay and held the other two delays constant. Our next two experiments use different network delays. The unequivocal result is a direct correlation between the network delay and the average delay per token; clearly a shorter network delay will decrease the time that it will take a token to traverse the network and reduce the amount of contention in the network.

We run the next experiment with the exact same parameters as the previous experiment except that the NETWORK_DELAY is set to 20, five times faster than the network delay in the previous experiment. The results of the experiment are graphed in figure 5.2. The results are similar to those of the previous experiment, but there are two noticeable differences. The first difference is that the points at which the lines in figure 5.2 level off is at about one fifth of the source delays of the points at which they level off in figure 5.1. The second difference is that the average times per token are about one-fifth of what they were in the previous experiment. Both of these differences have a factor of five, the same ratio of the two NETWORK_DELAYs. This implies that the network delay and contention are the dominating factors that contribute to the overall delay of each token. This is a common circumstance in applications that communicate over a network and are not processor intensive.

We run the final experiment in this section with the exact same parameters as the original experiment except that the NETWORK_DELAY is set to 500, five times slower than the network delay in the original experiment. The results of the experiment are graphed in figure 5.3. Again, we see that the average delays per token
Figure 5.2: Comparing various balancer distribution algorithms in the bitonic 16 counting network (faster network). (a) includes the entire range of y values, and (b) is zoomed in to the range 0 to 500 of y values are proportional to those in the original graph, and the factor is approximately the ratio of the network delays. The range displayed in figure 5.3 corresponds to the condition of high contention in the previous two experiments.

5.2 Network Size

Optimally, we would like to place each balancer on its own host. When we are to distribute a counting network over a network of hosts, we sometimes have more hosts than a certain size counting network but fewer hosts than the immediately larger network. We can use the larger network and place multiple balancers on hosts, but by placing multiple balancers on the same host, we increase the contention on the processor and in the network. This contention will cause the performance of the network to decline. If we use the smaller network, we can distribute it with a single balancer on each host, but if the network is too small to handle the token load, we
Figure 5.3: Comparing various balancer distribution algorithms in the bitonic 16 counting network (slower network)

will face the devastating contention that plagues Fetch&Increment shared counting variables.

Our experiment compares the performance of different size bitonic counting networks distributed over a range of 1 and 100 hosts. The bitonic networks of width 2, 4, 8, 16 and 32 are used. The balancers are distributed among hosts using our longest-run distribution algorithm. The BALANCER_DELAY parameter is set to 10, and the NETWORK_DELAY parameter is set to 100. The PARALLELISM parameter is set to 75%. Unlike the condition of the previous experiments, we now hold the SOURCE_DELAY parameter constant at 10. A total time of $1 \times 10^6$ units are allotted in order to process up to 1000 tokens.

Figure 5.4 shows the results of our experiment. There is an overall downward slope in all of the lines until a certain point at which the line levels off completely
Figure 5.4: Determining the optimal size bitonic counting network to use on different size networks

<table>
<thead>
<tr>
<th>source delay</th>
<th>network width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 3</td>
<td>2</td>
</tr>
<tr>
<td>4 - 10</td>
<td>4</td>
</tr>
<tr>
<td>11 - 38</td>
<td>8</td>
</tr>
<tr>
<td>39 - 58</td>
<td>16</td>
</tr>
<tr>
<td>59 - 63</td>
<td>32</td>
</tr>
<tr>
<td>64 - 71</td>
<td>16</td>
</tr>
<tr>
<td>72 - 73</td>
<td>32</td>
</tr>
<tr>
<td>74 - 100</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 5.1: The optimal size bitonic counting network for a range of token loads

to a constant value. Since a counting network cannot be distributed among more hosts than the number of balancers of which it consists, when more hosts are used than the number of balancers in the network, the performance will not change. The fact that the graphs are not monotonically decreasing at all points is evidence that longest-run distribution is not the optimal distribution algorithm and that a better distribution algorithm might exist. In table 5.1, we have extracted the optimal size bitonic network that should be used with different numbers of hosts under our experimental conditions. The results in table 5.1 are taken directly from figure 5.4. It is interesting to note that the optimal size network changes back and forth between the bitonic 16 and the bitonic 32 network when the number of hosts is greater than 59. As seen in the distribution methods experiment, there is a point at
which the counting network is able to handle the load to which it is subjected with little contention. This is the same phenomenon that we see with a network delay of greater than 59 in these two bitonic networks, and the varying optimal size is probably a result of longest-run distribution not really being the ideal distribution algorithm.

The following experiment exemplifies how changing a single parameter can cause the results of a simulation to contradict the correlation observed in a previous experiment. In the previous experiment, we observed larger networks performing better than their smaller counterparts by spreading out tokens over a greater area and decreasing contention. In this experiment we lower the amount of contention by changing the SOURCE_DELAY parameter from 10 to 100. This significantly decreases the amount of load placed on the network. The result is that the larger networks no longer perform better than the smaller networks. The contention no longer acts as a significant hinderance on the flow of the tokens through the network, so the greater depth of the network dominates by giving tokens a longer path through which to traverse.

5.3 Network Construction

In addition to being able to vary the size of a counting network, we can also vary the actual construction of the network. Different counting network constructions perform differently under different conditions based on traits such as their depth and width. For example, the BM network is better able to handle higher contention than the bitonic network since it has a greater output width with no change in the depth of the network. In this experiment, we compare three different constructions of counting networks, the bitonic 16 network, the BM network with 16 inputs and 32 outputs, and the periodic 16 network. The hosts are distributed using the longest-run distribution algorithm. The same parameters are used as in the previous experiments. The BALANCER_DELAY is 10, and the NETWORK_DELAY is 100. The PARALLELISM parameter is set to 75%. A total time of $1 \times 10^6$ units are allotted in order to process up to 100 tokens. The experiment is conducted with a range of 2 to 100 used for the SOURCE_DELAY.
Figure 5.5: Determining the optimal size bitonic counting network to use on different size networks (lower contention). (a) includes the entire range of $y$ values, and (b) is zoomed in to the range 0 to 2500 of $y$ values.

Figure 5.6 shows the results of our experiment. The periodic network performs the worst because it has both the largest depth and the most balancers. The BM network has better performance under the highest load situations, with a source delay between 2 and 10, but the bitonic network performs a little better when the load is decreased with source delay greater than 10. Since the BM network requires more balancers than the bitonic network of the same input width, the decrease in performance of the BM network could be due to the increased contention of having more balancers placed on each host. This raises the question of whether another distribution algorithm exists that is tailored better to the BM network.

As in the first section of this chapter, we want to see how the relative difference between the network delay and the other two delays, the source delay and the balancer delay, correlates to the performance of the network. We vary the NETWORK_DELAY parameter the same way that we did in the first section; we
decrease it by a factor of five to simulate a faster network and increase it by a factor of five in order to simulate a slower network. The results of using a faster network are graphed in figure 5.7, and the results of using a slower network are graphed in figure 5.8. Again, we see a correspondence between the ratios of the average network delays and the ratios of the average time per token.

Finally, we support our hypothesis that the network delay dominates the balancer delay in contributing to the overall delay that it takes to process each token. We repeat the first experiment of this section, but use a BALANCER_DELAY parameter that is five times larger. The results of this experiment are graphed in figure 5.9. Unlike the graph in figure 5.8, there is no noticeable difference between the graphs in figure 5.9 and 5.6. This implies that the need to use a fast network for communication between balancers is more important than having faster computers.
Figure 5.7: Comparing various counting network constructions (faster network). (a) includes the entire range of y values, and (b) is zoomed in to the range 0 to 1000 of y values

5.4 Parallelism

We can also show that the overall performance of counting networks has little correlation to the amount of parallelism in each host. In this final experiment, we vary the PARALLELISM parameter between 0% and 95%. We perform the experiment using between 4 and 40 hosts in order to vary the amount of contention on each host. The BALANCER_DELAY parameter is set to 10, the NETWORK_DELAY parameter is set to 100, and the SOURCE_DELAY parameter is set to 2. A total time of $1 \times 10^6$ units are allotted in order to process up to 100 tokens. The results of the experiment are plotted in figure 5.10. The amount of parallelism in each host does not seem to have an effect on the overall performance of the counting network. This fact further supports that the delay from the tokens traversing the network dominates the delay of the balancers processing the tokens.
Figure 5.8: Comparing various counting network constructions (slower network)
Figure 5.9: Comparing various counting network constructions (slower balancer)

Figure 5.10: Determining the effect that parallelism in hosts has on the performance of counting networks
CHAPTER 6
Counting Network Implementation

6.1 Description

In our implementation of a counting network we implement each balancer and counting variable as a thread executing on a networked host. Each thread listens for tokens on a UDP socket. The token has three fields, the internet address to which the token should be returned after a value is assigned to it, an initially empty field to which the counting variable writes its value before returning it to the source, and an extra data field that the source can use to identify the token. Each balancer is configured with a list of balancers or counting variables that are connected to its output wires. The balancers simply forward each token that they receive to next balancer or counting variable in its list. Each counting variable is configured with an initial value and an increment value. Each time the counting variable receives a token, it assigns its current value to the token, forwards the token back to its original source, and increments its current value by its increment value. Any host on the network can send a token to an input balancer of the counting network and listen on the UDP socket on which it requested the response to be sent.

6.2 Performance

The purpose of this section is to validate that our counting network implementation and our COST simulation have similar performance characteristics. The experiments were performed using ten UltraSPARC servers in Rensselaer Polytechnic Institute’s computer science department. The servers were public machines so the performance could have been affected by other processes that were running. Nonetheless, the results provide evidence that the same correlations that we observed in the simulation also exist in our implementation.

Each graph plots the average time that it takes for a client to receive a response after sending a token to the counting network as a function of the load on the counting network. The numbers on the x axis correspond to the number of sources
that are sending tokens to the network, but since each source sends tokens to random input balancers at a rate of one token per microsecond this number also corresponds to the overall rate that tokens are placed on the counting network. Each source forwards a total of 100 tokens to the network and we record the average time that it takes to receive the 100 responses to these tokens. Balancers are assigned to the ten UltraSPARC servers using the longest-run distribution algorithm.

Figure 6.1: Comparing various bitonic counting network sizes

Figures 6.1, 6.2, and 6.3 graph the performance of different size bitonic, BM, and periodic networks. In general, as a greater load is placed on the network, the performance of each network decreases. The networks of width 4 performed better than the networks of width 2, especially under greater contention. The networks of width 8 did not perform well and the network of with 16 performed terribly. This could be attributed to the limited number of hosts available. According to the results of the simulation, a larger number of hosts are required in order to benefit
Figure 6.2: Comparing various BM counting network sizes

from counting networks of larger widths.

Figures 6.4 and 6.5 compare the performance of the bitonic, BM, and periodic networks. In figure 6.4 each counting network has an input width of 8 and the BM network has an output width of 16. In figure 6.5 each counting network has an input width of 16 and the BM network has an output width of 32. As expected, the periodic network performs the worst of the three because of its largest depth. The BM network outperforms the bitonic network at points of higher load. These correlations were similar to those observed in the analogous experiment that we performed using our counting network simulation.
Figure 6.3: Comparing various periodic counting network sizes
Figure 6.4: Comparing various counting network constructions of input width 8
Figure 6.5: Comparing various counting network constructions of input width 16
CHAPTER 7
Software Details

Our software can be downloaded from http://www.cs.rpi.edu/~musser/gp. All of our code is written in C++.

7.1 Properties File

In our simulations, the counting network is configured from a properties file. This file lists each balancer in the counting network and to which host it is assigned. The balancers are encoded by listing each wire to which the balancer connects. If two balancers are connected to the same wire, the balancer in the lower level must be listed first. In addition to the wires to which the balancer connects, the host id to which the balancer is assigned may also be specified after a hyphen. The properties file can also list parameters used by the simulation. These parameters include the SOURCE_DELAY, BALANCER_DELAY, NETWORK_DELAY, HOST_COUNT, and PARALLELISM. These additional parameters will be ignored when the properties file is used to configure our socket implementation of a counting network. The following properties file represents the bitonic 4 network with balancers assigned to two different hosts with id’s 0 and 1.

```
SOURCE_DELAY 10
BALANCER_DELAY 5
NETWORK_DELAY 40
PARALLELISM 75
0 1 -0
2 3 -1
0 3 -0
1 2 -1
0 1 -0
2 3 -1
```

Figure 7.1 illustrates the structure of the counting network described by the previous properties file. The balancers on host 0 are highlighted in the network on
7.2 Generators

Counting networks, such as the bitonic network and the BM counting network [2], have a repeating structure and can be generated by recursive algorithms. We have implemented algorithms that generate counting networks encoded in the format of the aforementioned properties file. Each program takes one or more arguments that determine the width of the counting network and writes the counting network to standard output. Our first two programs, bitonic_generator and periodic_generator, both require a single argument, the base 2 logarithm of the number of wires in the network. Our third program, bm_generator, requires three arguments, the three parameters that determine the input and output widths of the BM counting network.

The algorithm for generating the bitonic sorting network is outlined explicitly in [5]. The bitonic sorting network is isomorphic to the bitonic counting network with each comparator replaced by a balancer. Since the bitonic counting network can only have a width of $2^i : i \in 1, 2, 3, ...$, we only need to provide $i$ in order to specify the network uniquely. It is fairly trivial to generate any bitonic counting network using the recursive algorithm from [5]. The algorithm for generating the periodic sorting network is even simpler because it has a less complex structure.

7.3 Distributors

We discussed four distribution algorithms in this paper and, thus, have four programs to implement them: random_distrib, trivial_distrib, level_distrib,
and \texttt{longest\_run\_distrib}. Each program reads the structure of the counting network from standard input and requires a single argument, the number of hosts on which to distribute the balancers. The programs write to standard output a list of balancers and the host to which each is assigned in the format prescribed for the properties file.

### 7.4 Visualizers

A counting network can be illustrated in a two-dimensional diagram using horizontal lines to represent the wires, vertical lines to represent the balancers, and dots to show where balancers connect to wires. In order to get a clearer picture of the structure of specific counting networks, we generate such diagrams by generating input scripts for the \texttt{plot} program.\(^2\)

Drawing two-dimensional plots of counting networks is fairly trivial and only requires calculating the coordinates of each of the lines and dots. After importing the balancers from the properties file, a dependency graph must be generated. This dependency graph has vertices that each represent a balancer and edges that each represent a wire connecting the output of one balancer to the input of a second balancer. Using this dependency graph, the level on which each balancer sits in the counting network can be determined. The overall depth of the counting network is the greatest level.

The balancers must be plotted such that if a balancer is at a greater level than another balancer, the vertical line that represents the first balancer must appear to the right of the vertical line that represents the second balancer. One candidate solution for plotting each balancer such that this property is satisfied is to divide the page up into \(n\) vertical sections, where \(n\) is the depth of the counting network, and plot each balancer in the section corresponding to its level in the network. This simple solution does not work because balancers may overlap each other. For example, if there are two balancers, the first connecting wires 0 to 2 and the second connecting wires 1 to 3, the balancers would overlap between wires 1 and 2. This

\(^2\)Plot is currently distributed as part of the GNU plotutils (plotting utilities) package, \url{http://www.gnu.org/software/plotutils/plotutils.html}. 

vertical line would appear on the plot to be a single balancer connecting wires 0, 1, 2 and 3.

In order to avoid overlapping balancers, we divide each level in the counting network into sublevels. We enumerate through each balancer in each level. As we enumerate through the balancers, we add the balancer to the first sublevel in which it does not overlap any other balancer already in that sublevel. If no such sublevel is found, we place the balancer in a new sublevel. This solution allows us to plot all balancers such that no balancer appears to the left of another balancer that is in a lower level in the counting network. We have implemented this solution in a program named \texttt{vis\_basic}.

A less trivial problem is representing the hosts to which each balancer is assigned. One solution is to color each vertical line based on the host to which the balancer is assigned. Using the RGB scale we can encode discrete colors using three digit numbers, with red being the first digit, green being the second digit, and blue being the third digit. We shall represent an RGB value as an ordered triple, \((\text{red value}, \text{green value}, \text{blue value})\). Given a 3 digit base \(n\) number, \(n^3 - 1\) colors, excluding the white background \((n, n, n)\), can be represented. Therefore, given \(h\) hosts, we must solve the inequality, \(n^3 - 1 \geq h\), in order to determine the base of the number that we should use. Plot allows a value between 0 and 65535 for each of the red, green, and blue values, with \((0, 0, 0)\) representing black and \((65535, 65535, 65535)\) representing white. In order to give a greater contrast, we multiply each digit by a scaling factor, \(65535/n\). Using this scale we can encode each host as a different discrete color. This solution fails because the human eye is unable to distinguish a contrast between many colors with different RGB values.

This leads us to another possible solution to the problem of picking colors with large contrasts. Using the HSV scale we can encode colors with a hue between 0 and 359 degrees, a saturation between 0 and 1, and a value or luminescence between 0 and 1. Using a saturation of 1 and a luminescence of .5, we can find a good contrast of colors by choosing equally spaced hues around the circle. This solution falls apart when more than about eight hosts exist because of our limited ability to distinguish between similar colors. We have implemented this solution in \texttt{vis\_color}. 
<table>
<thead>
<tr>
<th>column</th>
<th>metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>host count</td>
</tr>
<tr>
<td>2</td>
<td>balancer count</td>
</tr>
<tr>
<td>3</td>
<td>level count</td>
</tr>
<tr>
<td>4</td>
<td>average delay</td>
</tr>
<tr>
<td>5</td>
<td>balancer contention</td>
</tr>
<tr>
<td>6</td>
<td>balancer time</td>
</tr>
<tr>
<td>7</td>
<td>network contention</td>
</tr>
<tr>
<td>8</td>
<td>network time</td>
</tr>
</tbody>
</table>

Table 7.1: The metrics output by our simulator

An alternate solution is to make separate plots for each host. In each plot the balancers on a specific host are highlighted in a darker color (or shade of grey) than the rest of the illustration. Using the plot program, we create multiple two-dimensional images with a single image on each page. Each image has the balancers assigned to a single host highlighted. We have implemented this solution in vis_highlight.

7.5 Execution

“We should have some ways of connecting programs like garden hose—
screw in another segment when it becomes necessary to massage data in
another way. This is the way of IO also.”

All of the programs in our counting network suite are generic and can interact via pipes with other programs. Five types of executables are provided: network generators, balancer distributors, visualizers, a simulator, and our socket implementation. The network generators take one or more arguments as input, and they output the counting network to standard output. The balancer distributors take a single argument, the number of hosts on which to distribute the balancers. The distributors read the counting network from standard input, and they write the distributed counting network to standard output. The visualizers read a counting network or a distributed counting network from standard input, and they write the

\[ ^3 \text{The first requirement in a memo dated October 11, 1964 from Doug McIlroy to Dennis Ritchie [9].} \]
plot script that can draw the network to standard output. Finally, the simulation
and the implementation both read a counting network or a distributed counting
network from standard input.

7.5.1 COST Simulation

The simulator allows a single optional argument, the number of time steps
for which the simulation should be executed. It outputs one line of tab-delimited
data containing the metrics listed in table 7.1 to standard output. It also outputs
messages to standard error about the progress of each token.

7.5.2 Socket Implementation

The socket implementation consists of three main executables and a fourth
one for diagnostic purposes. The network is configured using the master / slave
paradigm. A daemon program called slave must be running on each host to which
balancers and counting variables may be assigned. This program takes a single argu-
ment, the port on which the daemon listens for requests to start balancer or counting
variable processes. There are two programs that act as the masters, master and
resetter. Both of these programs require a data file named hosts that lists the host
names and ports on which daemons are running. master reads a counting network
or distributed counting network from standard input and instructs the hosts listed
in the data file to create threads that perform the roles of the described balancers
and counting variables. The counting variables are implicit in the structure of the
counting network; a counting variable is assigned for each wire and is placed on the
same host as the output balancer that connects to that wire. The master program
writes a binary file that contains an array of struct sockaddr_ins, the locations on
which input balancers in the counting network are listening for requests. resetter
takes no arguments and simply resets all of the slaves listed in the hosts file by
instructing them to close the ports that are listening for tokens and kill the threads
that perform the balancer and counting variable functions.

tester is simply a diagnostic program that causes tokens to be sent out to
the counting network from slaves listed in the hosts file. It prints out the average
time that it takes the tokens to be returned to their source. Any program can send
tokens to input balancers in the counting network by sending it a *struct token* using the UDP protocol. The *struct token* is defined as follows:

```c
struct Token {
    struct sockaddr_in source;
    int value;
    char data[64];
};
```

The *source* field contains the address to which the counting network should return the token once a counting variable populates the *value* field. The *data* field can be used by the source for bookkeeping and is not altered by the counting network.

### 7.6 Examples

The following examples outline the use of our simulation suite.

```bash
./bitonic_generator 4 | ./random_distrib 10 | ./counting_network 1000
```

This command will generate the bitonic 16 network, distribute the balancers randomly among 10 hosts, and run the simulation for 1000 units on this distributed network using the default simulation delays.

```bash
./bm_generator 4 3 5 | ./longest_run_distrib 16 | \
./vis_highlight | plot -T X
```

This command will generate the 16 input / 96 output BM counting network, distribute the balancers among 16 hosts using the longest-run distribution algorithm, create a *plot* script that highlights the balancers assigned to each host on a single page, and plots the graphics in a series of X Windows.

```bash
./periodic_generator 6 | ./vis_basic | plot -T ps > periodic64.ps
```

This command will generate the periodic 64 counting network, create a *plot* script that plots the structure of this counting network on a single page, converts the *plot* script to a postscript image, and saves this image as periodic64.ps.

```bash
./bm_generator 2 1 3 | ./longest_run_distrib 8 | ./master
```
This command will generate the 4 input / 8 output BM counting network, distribute the balancers among 8 hosts using the longest-run distribution algorithm, and assign the counting network to the hosts listed in `hosts`.

`./tester -h 4 -d 100 -c 50`

This command will cause the first four hosts listed in `hosts` to send tokens to the counting network. Each of the four hosts will send a total of 50 tokens at a rate of one token every 100 microseconds. This program must be run from the same directory as the previous example because it requires the data file that lists the addresses on which the input balancers are listening for tokens.
CHAPTER 8
Conclusions

In this thesis we describe a method for implementing counting networks in software and distributing these counting networks among multiple computers. We provide a simulation that is modeled after our method for implementing counting networks. Our simulation can help determine the best counting network construction, size, and distribution method that should be used under various conditions. We also provide a generic version of this distributed counting network implementation that communicates by sending messages over internet sockets.

We introduce four novel algorithms for distributing the balancers in a counting network among multiple hosts. Using our simulation, we found that the longest-run distribution algorithm usually outperforms the other three distribution methods that we use in this paper. When the counting network is subjected to a condition causing maximum contention, the trivial distribution algorithm outperforms the longest-run distribution algorithm.

We also found that the BM counting network outperforms the bitonic and periodic networks under most conditions. The bitonic counting network outperforms the BM counting network under conditions of minimal contention. Our counting network implementation appears to perform similarly to our counting network simulation.

Possible extensions to our software could include a program to generate the linearizable counting networks of [8]. The architecture of our software can be extended to support any balancing network. A possible application for balancing networks could be load balancing. Furthermore, our software makes it easy to introduce new algorithms for distributing balancers among hosts. We believe that there are balancer distribution algorithms that perform better than longest-run distribution, and their discovery requires further investigation.

Our results support the benefit of using counting networks to relieve contention in algorithms. An infinite number of counting networks can be generated. Our
counting network simulation can be used to help determine the network that performs best under a variety of conditions. Finally, we believe that our generic counting network implementation will facilitate using counting networks in distributed applications.
LITERATURE CITED


