PIPELINED OPTIMAL
BROADCAST WITH SELECTIVE REDUCTION (BSR)

by

Chester L. Langin

B.S., Southern Illinois University, Carbondale, Illinois, 1974

A Thesis
Submitted in Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

Department of Computer Science
in the Graduate School
Southern Illinois University
Carbondale
April, 2003
AN ABSTRACT OF THE THESIS OF

CHESTER L. LANGIN, for the Masters of Science degree in Computer Science, presented on 3/20/2003, at Southern Illinois University at Carbondale.

TITLE: PIPELINED OPTIMAL BROADCAST WITH SELECTIVE REDUCTION (BSR)

MAJOR PROFESSOR: Dr. Chih-Fang Wang

The Optimal Broadcast with Selective Reduction (BSR) implementation described in Lindon[13] was simplified and pipelined. This Pipelined Optimal BSR was programmed as an interactive real-time Java applet, and published examples of BSR algorithms were successfully run on it. Histogram and External Watchman Routes algorithms took advantage of the pipelining in $O(\log^2(n + m) + n'/n)$ real-world time, where, conventionally, $n$ represents the number of processors and $m$ represents the number of memory locations. $n'$ represents the total number of inputs produced by the $n$ processors. (In Pipelined Optimal BSR, $n$ and $m$ can also represent other things.) For large data sets, this is an improvement over the comparable time of $O((n'/n)\log^2(n + m))$ in Optimal BSR, yet the cost of $O((n + m)\log^2(n + m))$ is the same as for Optimal BSR.
ACKNOWLEDGEMENTS

I thoroughly enjoyed the people, classes, and research I encountered in the Computer Science Department of Southern Illinois University at Carbondale and I regret having to leave this pleasant environment. Dr. Chih-Fang Wang had a plan which met my needs and interest and patiently led me from start to finish. Without him the entire project would have been inconceivable. Thanks to Xuanwen Luo who was my research partner the first half of the project. Dr. Michael S. Wainer was full of ideas and led me from a text environment through OpenGL and into Java, so that we could actually see what the program was doing. Dr. Robert J. McGlinn sharpened my programming and Java technical skills. Dr. William Wright, as Departmental Chair, provided the positive environment where this could take place. Lorraine Fava Lindon wrote the detailed Optimal BSR report which was my starting point. I feel like we are friends even though we have never met or communicated. Thanks to everyone. Of course, any mistakes are my own doing.
# TABLE OF CONTENTS

**ABSTRACT**  

**ACKNOWLEDGEMENTS**  

**LIST OF TABLES**  

**LIST OF FIGURES**  

1 **INTRODUCTION**  

1.1 The General Concept of BSR .................................. 1  

1.2 The PRAM ...................................................... 2  

1.2.1 Naïve PRAM ............................................. 3  

1.2.2 Vishkin’s PRAM .......................................... 4  

1.2.2.1 Sorting Circuits ....................................... 6  

1.2.2.1.1 Batcher’s Odd-Even Sorting Circuit ............... 6  

1.2.2.1.2 AKS Sort. ........................................ 7  

1.2.2.2 A Merge Circuit ....................................... 9  

1.2.2.3 Vishkin’s Complexity .................................. 10  

1.3 BSR .......................................................... 11  

1.3.1 Naïve BSR ............................................... 11  

1.3.2 Optimal BSR ............................................... 14  

1.3.2.1 The Prefix Box ..................................... 16  

1.3.2.2 The Distribute Box ................................... 19  

1.3.2.3 Optimal BSR Overview ............................. 22  

1.3.2.4 Optimal BSR’s Complexity ......................... 26  

1.3.3 Other BSR Models and Implementations .................... 26  

1.4 Complexity Issues ........................................... 27  

1.5 Statement of the Problem .................................. 27
# 2 REVIEW OF THE LITERATURE

2.1 BSR Notations ................................................................. 29
  2.1.1 Naïve BSR Notation ..................................................... 29
  2.1.2 BSR Pseudocode .......................................................... 30
  2.1.3 Optimal BSR Notation .................................................. 31
  2.1.4 One-Line BSR Notation .................................................. 32
2.2 Published BSR Algorithms ................................................ 32
  2.2.1 Prefix Sums ............................................................... 33
  2.2.2 Element Uniqueness ..................................................... 33
  2.2.3 Count Sort ................................................................. 34
  2.2.4 Sieve of Eratosthenes ................................................ 34
  2.2.5 Maximal Sum Subsequence ............................................ 35
  2.2.6 Maximal Vectors ........................................................ 36
  2.2.7 Convex Hull ............................................................... 37
  2.2.8 Maximal Sum Subrectangle ............................................ 37
  2.2.9 External Watchman Routes ............................................ 39
  2.2.10 Overlapping Intervals ................................................. 40
  2.2.11 Union of Intervals ..................................................... 40
  2.2.12 ε-Closeness ............................................................. 40
  2.2.13 Maximum Gap .......................................................... 41
  2.2.14 Intersection of Two Convex Polygons .............................. 41
  2.2.15 Construct a Voronoi Diagram ....................................... 42
  2.2.16 ECDF Searching ........................................................ 42
  2.2.17 2-Set Dominance ........................................................ 42
  2.2.18 Intersection of Isothetic Line Segments ............................ 42
  2.2.19 Vertical Segment Visibility .......................................... 42
  2.2.20 High Dimensional Maximal Elements .............................. 42
2.2.21 Rectangle Containment in d-Dimensional Space ........................................... 42
2.2.22 Rectangle Enclosure Counting in $R^d$ .......................................................... 43
2.2.23 Rectangle Intersection Counting in $R^d$ .......................................................... 43
2.2.24 Histogram ........................................................................................................ 43
2.2.25 All Nearest Neighbors and Furthest Pairs ....................................................... 43
2.2.26 Distance Transform ......................................................................................... 43
2.2.27 Medial Axis Transform .................................................................................... 46
2.2.28 Area and Perimeter of Isooriented Rectangles ............................................... 46
2.2.29 Discrete Voronoi Diagram for Labeled Images ............................................... 46
2.2.30 Parenthesis Matching ..................................................................................... 46
2.2.31 Decoding Binary Trees .................................................................................... 48
2.2.32 Generating Binary Trees .................................................................................. 48
2.2.33 Reconstruction of a Binary Tree From Its Traversals ...................................... 49
2.2.34 All Nearest Smaller Values ............................................................................. 50
2.2.35 Maximal Sum Subsegment Problem .............................................................. 51
2.2.36 2D maximal Sum Subsegment Problem ......................................................... 51
2.2.37 Longest Increasing Sequence Problem ............................................................ 52
2.2.38 Sequence Alignment Problem ......................................................................... 52
2.2.39 Longest Increasing Subsequence .................................................................... 52
2.2.40 Longest Common Subsequence ...................................................................... 52
2.2.41 Sorting Vertices of a Multidimensional Space Lexicographically ...................... 52
2.2.42 Three Counting Problems for a Relational Database ...................................... 52
2.2.43 Decoding a Binary Tree from its $i-p$ Sequence ............................................. 52
2.2.44 Drawing a Binary Tree from its $i-p$ Sequence ................................................ 52
2.2.45 Rearranging Scattered Information ................................................................ 53
2.2.46 $k$-compaction .................................................................................................. 53
2.2.47 Extremal Search of Convex Polygons .............................................................. 53
2.2.47 Vertex-Vertex Pairs of Convex Polygons ........................................ 53
2.2.49 Vector Sum of Two Convex Polygons ........................................... 54
2.2.50 Critical Support Lines of Two Convex Polygons ............................... 54
2.2.51 Maximal Distance Between Two Convex Polygons ............................ 54
2.2.52 Diameter of a Polygon ................................................................. 54
2.2.53 Width of a Polygon ................................................................. 54
2.2.54 Detection of Repetitions ............................................................ 54
2.2.55 k-LCS Problem ................................................................. 55
2.2.56 Geometrical Problems ............................................................. 55

3 METHODOLOGY ................................................................. 57
3.1 The Distributed PRAM Simulator ..................................................... 58
3.2 Pipelined Optimal BSR Issues ....................................................... 59
  3.2.1 Storage of Operations, Comparators, and Limits ....................... 59
  3.2.2 Disconnecting BSR from the PRAM ........................................ 60
  3.2.3 Inconsistent Record Lengths .................................................. 61
  3.2.4 Processor and Memory Records are Really the Same Type of Records .... 62
  3.2.5 Differentiating Source and Target Records ............................... 62
  3.2.6 Determining Where to Send the Output .................................... 63
  3.2.7 The Identity does Not Work .................................................. 63
  3.2.8 Indicating Valid Output ....................................................... 63
  3.2.9 Control in Box B is Bidirectional ......................................... 65
  3.2.10 Table 3’s Location is Vague ............................................... 65
  3.2.11 Uneven Stages ................................................................... 66
  3.2.12 Placement of Box C .......................................................... 68
  3.2.13 Additional Registers .......................................................... 68
  3.3 Pipelined Optimal BSR .......................................................... 70
    3.3.1 An Overview of BSR in Java ........................................... 70
3.3.1.1 An Overview of the Source Code .............................. 76
3.3.2 Box A .......................................................... 78
  3.3.2.1 Box A Source Code ........................................ 79
3.3.3 The Table ......................................................... 81
  3.3.3.1 The Table Source Code ..................................... 82
3.3.4 Box B .......................................................... 83
  3.3.4.1 Box B Source Code ........................................ 84
   3.3.4.1.1 Mode 5 .................................................. 86
    3.3.4.1.1.1 Mode 5 Source Code .............................. 88
3.3.5 Box C .......................................................... 89
  3.3.5.1 Box C Source Code ........................................ 89
3.3.6 Box D .......................................................... 89
  3.3.6.1 Box D Source Code ...................................... 90
3.3.7 Box E .......................................................... 90
  3.3.7.1 Box E Source Code ...................................... 93
3.3.8 Box F .......................................................... 94
  3.3.8.1 Box F Source Code ...................................... 94
3.3.9 A Read .......................................................... 94
3.4 Analysis of the Algorithms .......................................... 95
  3.4.1 Basic PRAM algorithms ...................................... 96
    3.4.1.1 Random .................................................. 96
    3.4.1.2 Priority .................................................. 96
    3.4.1.3 Arbitrary ............................................... 96
3.4.2 The Published Algorithms ...................................... 96
3.5 Pipelined Algorithms ................................................. 98
  3.5.1 Histogram ..................................................... 98
    3.5.1.1 Histogram Complexity ................................. 99
3.5.2 External Watchman Routes ........................................ 101

4 RESULTS ................................................................. 103

5 CONCLUSIONS .......................................................... 105
  5.1 General Conclusions .................................................. 105
  5.2 Summary of Contribution ............................................ 105
  5.3 Future Research ...................................................... 105

APPENDIX ............................................................. 107

A LONGER SOURCE CODE SEGMENTS ................................. 108
  A.1 A Source Code Segment for Box A Nodes ....................... 108
  A.2 A Source Code Segment for Executing the Table ................ 109
  A.3 A Source Code Segment for Box B Nodes ....................... 110
  A.4 A Source Code Segment for Box B Nodes in Mode 5 .......... 112
  A.5 A Source Code Segment for Box E Switches ................... 115
  A.6 A Source Code Segment for Box E Nodes ....................... 118

BIBLIOGRAPHY .......................................................... 121
LIST OF TABLES

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Prefix Sums Example.</td>
<td>16</td>
</tr>
<tr>
<td>1.2</td>
<td>A Summary of Complexities.</td>
<td>27</td>
</tr>
<tr>
<td>2.1</td>
<td>Summary of Characteristic BSR Variable Contents for Published Algorithms</td>
<td>56</td>
</tr>
<tr>
<td>3.1</td>
<td>Sorted BSR Instructions for Published Algorithms.</td>
<td>97</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

1.1 The General Concept of BSR. ......................................................... 1
1.2 The PRAM. ............................................................................. 2
1.3 A Naive PRAM Implementation. ............................................... 4
1.4 Vishkin’s PRAM Top View. ....................................................... 5
1.5 Vishkin’s PRAM Boxes. ............................................................ 5
1.6 A Comparator Node. ............................................................... 6
1.7 Batcher’s Sort. ................................................................. 6
1.8 A Stage. .......................................................................... 7
1.9 A Complexity Constant. ..................................................... 8
1.10 A Merge Circuit. ............................................................. 10
1.11 Naive BSR. ................................................................. 12
1.12 Optimal BSR Topview........................................................ 14
1.13 Optimal BSR Boxes. ....................................................... 15
1.14 Prefix and Suffix Nodes. .................................................. 17
1.15 The Prefix Box. ............................................................ 18
1.16 Groups of Tags and Limits. ............................................... 19
1.17 Sample Distribute Leaders ............................................... 20
1.18 Using the Leaders. .......................................................... 21
1.19 The Distribute Box. ........................................................ 22
1.20 Overview of Processor Input. ............................................... 23
1.21 Overview of Memory Location Input. ................................... 24
1.22 Overview of Distribution. .................................................. 24
1.23 Overview of Outputs. ........................................................ 25

3.1 The First Transitional Top-View of Optimal ↔ Pipelined Optimal BSR. .......... 61
3.2 The Second Transitional Top-View of Optimal ↔ Pipelined Optimal BSR. .......... 64
3.3 Control in Box B is Bidirectional ........................................... 65
3.4 Control in Box B becomes Unidirectional ........................................ 66
3.5 The Final Transitional Top-view of Pipelined Optimal BSR .................. 66
3.6 Uneven Stages ................................................................. 67
3.7 Even Stages ................................................................. 67
3.8 Moving Box C ................................................................. 68
3.9 Pipelining ................................................................. 69
3.10 The Opening Screenshot of BSR in Java ........................................ 71
3.11 The Application Input Dialog Box ............................................. 72
3.12 The Darkened Switches Hold the New Input Records ......................... 74
3.13 A Popup Box Describes a Component’s Contents ............................ 75
3.14 The Class Dependencies in BSR in Java .................................... 76
3.15 The Source Code Segment for (i, t, r, d) ....................................... 77
3.16 The Source Code Segment for Pertinent Global Constants ................. 78
3.17 The Box Source Code Segment to Increment a Stage ....................... 78
3.18 Box A, Node 9 ..................................................................... 79
3.19 The Source Code Segment for Copying Records in Box A .................. 80
3.20 The Table ........................................................................ 82
3.21 Box B, Node 13 .................................................................. 83
3.22 The Source Code Segment for Box B Control Data ......................... 84
3.23 The Source Code Segment to Execute Box B Switches ....................... 84
3.24 The Source Code Segment for Box B Control Data in Nodes ............ 85
3.25 The Source Code Segment to Execute Box B Nodes ......................... 85
3.26 The Source Code Segment for Translating the Mode ....................... 86
3.27 The Source Code Segment for Box B Input Selection ...................... 86
3.28 Box B, Node 13, Mode 5 ...................................................... 87
3.29 Mode 5 ........................................................................ 88
3.30 The Source Code Segment for Box B Switches in Mode 5 ................. 89
3.31 Box E, Node 25 .................................................................................. 91
3.32 Box E, Switch 12 ................................................................................. 92
3.33 The Source Code Segment for Defining Data in Box E ....................... 93
3.34 Input for a Memory Read .................................................................... 95
3.35 An Optimal BSR Bucket Sort ............................................................... 99
3.36 Variable $n'$ ......................................................................................... 100
3.37 Dual Pipelined Optimal BSR’s .............................................................. 101
CHAPTER 1 – INTRODUCTION

Pipelined Optimal Broadcast with Selective Reduction (BSR) is the result of the most recent of a series of refinements on the theoretical Parallel Random Access Machine (PRAM). This Pipelined model not only extends the previous Optimal BSR implementation, but it also simplifies it in the process and takes BSR from the PRAM theoretical world into an actual computer. This Introduction starts with the general concept of what BSR means in Section 1.1. Then, it regresses to a review of the PRAM in Section 1.2 and shows the refinements accomplished to achieve Optimal BSR, which is introduced in Section 1.3.2. In this chapter, $n = Number$ of Processors and $m = Number$ of Memory Locations.

1.1 The General Concept of BSR

BSR is a concept of electronically processing information while it is in transit. Compare BSR with the delivering of mail on a train. Suppose that there is a mail car on this train on which a postal worker sorts the mail while it is being transported from one location to another. BSR can be compared to this postal worker as far as someone or something which processes information while it is in transit.

![Figure 1.1: The General Concept of BSR.](image-url)

Figure 1.1 illustrates the general concept of BSR for $n = 4$ and $m = 4$. $n$ and $m$ can vary and do not have to be equal. The data from all of the processors is broadcast towards all of the memory locations. The vertical line in the middle symbolically represents a barrier where only the selected data
passes through. A reduction operation, such as summation, is then appropriately applied to the data which has been selected. A final result then potentially proceeds to each target. Not all of the targets will necessarily get data. Figure 1.1 is just symbolic and does not indicate how BSR is implemented. Implementations will be explained beginning in Section 1.3.1, after some background material is given.

The flow of data in BSR is generally perceived as going from parallel processors to shared memory. However, it can also go from memory locations to processors, and it can potentially have other types of sources and targets, as explained in the Pipelined implementation in Section 3.2.

1.2 The PRAM

The Parallel Random Access Machine (PRAM), described in Akl[2], is the predecessor of BSR. The PRAM is an imaginary machine that is important for theoretical reasons. In the PRAM, any processor, by definition, can access any memory location in asymptotic $\Theta(1)$ (constant) time. The complexities of PRAM algorithms are used in the study of parallel algorithm theory.

![Figure 1.2: The PRAM.](image)

Figure 1.2 illustrates the PRAM. The black box represents that it is not known, nor does it matter, how the $n$ processors and $m$ memory locations are actually connected. The access time is defined as being $\Theta(1)$. The connecting box has been called both an Interconnection Unit (IU) in Lindon[13] and a Memory Access Unit (MAU) in Akl[3].
The PRAM has four types of memory access: Exclusive Read (ER), Exclusive Write (EW), Concurrent Read (CR), and Concurrent Write (CW). The abbreviations are often combined to designate the type of read and the type of write. For example, CRCW means Concurrent Read and Concurrent Write. ER and EW are routine reads and writes: One processor reads from or writes to one memory location. CR means that more than one processor can read from a single memory location at a time. CW has special implications beyond the other three types of memory accesses. It means that more than one processor can write to a single memory location at a time. The datum, if any, of the potential ones which is written to a memory location can be designated in many different ways. Here are some of the ways:

Priority: One of the processors has precedence, with the highest priority. This is often the one with the lowest index. However, other ways of determining priority are possible. The value of the processor with the highest priority is the value written to the specified memory location.

Common: All of the processors must be writing the same value.

Random: One of the values is picked at random to be written.

Arbitrary: The value of an arbitrary processor is written.

Combine: The values are combined in some way, such as addition, to provide a, possibly new, value to be written. Combine specifically applies to BSR.

1.2.1 Naïve PRAM

In a Naïve PRAM, an Address Decoder Tree (ADT) for each processor routes the datum, and a Concurrent Access Tree (CAT) for each memory location reduces the data sent to it. If appropriate, the CAT’s use a CW method, such as Priority or Combine, to reduce the values to one which can be written to memory. The size is $\Theta(nm)$ and the access time is $\Theta(log(n + m))$. This model does not indicate how the control information is sent to the CAT’s. This is not an issue at this time, but will become important in the Pipelined implementation in Section 3.2. Also, the processor data arrives already sorted. This will not always be the case with other models.
Figure 1.3 illustrates a naïve PRAM model with \( n = 4 \) and \( m = 4 \). A sample ADT is highlighted in a dashed box on the left and a sample CAT is highlighted in a dashed box on the right.

The PRAM has two stated time complexities depending upon the context: A defined time complexity of \( \Theta(1) \) for the study and comparison of algorithms, and, a \( O(\log(n + m)) \) time complexity in the naïve model. When AKS Sort is introduced in Section 1.2.2.1.2, a third stated complexity context will appear.

### 1.2.2 Vishkin’s PRAM

Vishkin [24] described an optimal PRAM using sorting and merging techniques.
A top-down view of Vishkin’s PRAM is in Figure 1.4 with $n = 4$ and $m = 4$. The processor and memory records enter from the left, exchange data, and exit to the left. The box is no longer black. How the model is constructed is important and is shown in the next figure.

The layout of Vishkin’s PRAM can be characterized as consisting of two boxes, as shown in Figure 1.5, with $n = 4$ and $m = 4$. These boxes represent combinational circuits, indicated by Box A and Box B, above. The processor records enter from the left and are sorted according to the memory locations that they are accessing. The memory records are already sorted. All of the records are then merged and paired up on the right. They exchange data and go back to their original locations. The circuits are
explained below. First, two sorting circuits, and then a merging circuit.

1.2.2.1 Sorting Circuits

Sorting circuits facilitate reducing the asymptotic sizes of PRAM and BSR, resulting in Optimal versions. Two sorting circuits are given because their different complexities are important in understanding the implementations and theory. Batcher’s odd-even sort is used as a practical circuit, but it has a higher asymptotic complexity. AKS Sort has lower asymptotic complexity, but is not practical to implement.

1.2.2.1.1 Batcher’s Odd-Even Sorting Circuit. Batcher’s Odd-Even Sorting Circuit[8] is a practical way of sorting with comparator nodes. Batcher created more than one sorting circuit. However, in this report, Batcher’s Sort refers to Batcher’s Odd-Even Sorting Circuit.

\[
\begin{array}{c}
    x \\
    y
\end{array}
\begin{array}{c}
    \min(x, y) \\
    \max(x, y)
\end{array}
\]

Figure 1.6: A Comparator Node.

Figure 1.6 shows how a comparator node sorts two inputs. Two inputs, designated by \( x \) and \( y \), are sorted. The minimum of \( x \) and \( y \) is output at the top, and the maximum is output at the bottom.

![Figure 1.7: Batcher’s Sort.](image)

Figure 1.7 shows how Batcher’s Sort places a group of comparator nodes to sort eight inputs. The number of inputs is designated as \( N \), which, depending upon the circumstances, might be equal to \( n \), to
m, to n + m, or to something else. Sample inputs are shown on the left. After going through a series of
odes, the sample is sorted and output on the right. Batcher’s Sort has a size complexity of O(Nlog²N)
and a time complexity of O(log²N). This will become useful later in considering the practical time
complexity of Optimal BSR in Section 1.3.2.4.

![Diagram of a stage](image)

Figure 1.8: A Stage.

A stage is a column of components in a combinational circuit, as shown in Figure 1.8. All
components in a stage are executed simultaneously. The number of stages indicates how long it takes a
combinational circuit to run. Typically, one stage is considered to take one time unit to run. The
Batcher’s Sort example in the figure has six stages, enumerated with arrows below the circuit. Thus, it
would take six time units to run. Stage 2, as an example, is enclosed in a dashed box. Batcher’s Sort
has (logN)(1 + logN)/2 stages. This matches the example in Figure 1.8 where N = 8 and there are six
stages. The number of asymptotic stages in Batcher’s Sort is O(log²N).

Typically, the size of a combinational circuit is given as the number of inputs times the number of
stages, Nlog²N, which for a Batcher’s Sort of size N = 8 is 72. However, the diagram of the instant
example shows 19 comparator nodes, which is a considerably smaller value. This is an important
consideration when comparing this sorting circuit with AKS Sort in the next section.

1.2.2.1.2 AKS Sort. AKS Sort is named after its originators, Ajtai, Komlós, and Szemerédi. See
Ajtai[1]. AKS Sort is a huge and intricate sorting circuit. The stage complexity, and thus time
complexity, of AKS Sort is $O(\log N)$ which is asymptotically smaller than the comparable Batcher’s Sort complexity of $O(\log^2 N)$.

![Graph](image)

Figure 1.9: A Complexity Constant.

However, as Figure 1.9 shows, the smallest asymptotic size is not always the smallest practical size. $f(x)$ increases at rate $x$ and has a complexity of $\Theta(x)$. $g(x)$ does not increase, at all, and has a lower complexity of $\Theta(1)$. However, $f(x)$ has a lower value than $g(x)$ when $x < n_0$, which is the intersection point. The relevance of this depends on $c$, which is a constant applied to $g(x)$. If $c$ is low, such as 1.5, then $g(x)$ can probably be assumed to be the faster function. However, if $c$ is high, such as 1.5 million, then $g(x)$ might be impractical, and $f(x)$ might be considered as the faster function.

Even though AKS Sort has the best asymptotic time, its constant is too high. For this reason, AKS Sort is not used in Pipelined Optimal BSR in Section 3.2. Batcher’s Sort is used, instead. AKS Sort can theoretically be used in Optimal BSR in Section 1.3.2, but it is not shown in the description of Optimal BSR in this report. Again, Batcher’s Sort is used.

Paterson[18] devised an AKS Sort example with a size constant of approximately 6,000. However, the components were in hundreds of layers such that the time/stage constant was approximately 6.15. To find $n_0$, set the time complexities of AKS Sort and Batcher’s Sort equal to each other and solve for $N$:

$$6.15 \log N \approx (\log N)(1 + \log N)/2 \quad (1.1)$$

$$11.3 \approx \log N \quad (1.2)$$

$$n_0 \approx N \approx 2,521 \quad (1.3)$$
Which means that more than approximately 2,521 processors need to be used before AKS Sort becomes faster than Batcher’s Sort. However, at 2,521 processors, the size of AKS Sort is $O(170, 918, 758)$ while the size of Batcher’s Sort is only $O(175, 190)$; a difference by a factor of approximately 975. As shown in the previous section, the actual sizes of the circuits are less than the upper size boundaries calculated here. However, this still shows that while AKS Sort is asymptotically faster than Batcher’s Sort, AKS Sort is impractical to use because of its high complexity constant.

AKS Sort brings a third context for time complexity into consideration. The three contexts are as follows:

$\Theta(1)$: PRAM is defined as having constant time. This context is used for the study and comparison of algorithms.

$O(\log N)$: AKS Sort has a time complexity of $O(\log N)$. PRAM can use it and have this same complexity. However, AKS Sort has a constant which is so high that AKS Sort is impractical.

$O(\log^2 N)$: Batcher’s Sort has a time complexity of $O(\log^2 N)$. PRAM can use it and this is PRAM’s practical complexity.

1.2.2.2 A Merge Circuit

A merge circuit can be used to pair up processor records with memory records in order to exchange data in the PRAM. This section demonstrates such a circuit as used by Vishkin’s PRAM.
Figure 1.10: A Merge Circuit.

In Figure 1.10, with \( n = 4 \) and \( m = 4 \), two sorted sets of data enter from the left. Each is enclosed in a box for emphasis. One set of data is from the processors and one set is from the memory locations. The processor data records are sorted based on the addresses of the memory locations that they are accessing. The circuit is created from comparator nodes, see Figure 1.6, and is actually just the right end of Batcher’s Sort, see Figure 1.7. The data is merged so that the processor and memory records are paired up as they exit on the right. In the PRAM, data is exchanged at this point. Then, the data goes backwards through the circuit to the original processor and memory locations, but with the new data which has just been exchanged.

Imagine the processor and memory records as being couriers which meet in the town square. The data to be read from or written to memory is a briefcase. After the merge, the appropriate couriers are standing next to each other where they can pass the briefcase. Then, they retrace their routes to their original locations.

1.2.2.3 Vishkin’s Complexity

The complexity of Vishkin’s PRAM depends upon which sorting circuit is in Box A, referring again to Figure 1.5. When basing PRAM complexities on AKS Sort or Batcher’s Sort, replace \( N \) with \( n + m \).

With impractical AKS Sort, Vishkin’s PRAM retains the time complexity of the Naïve PRAM in Section 1.2.1 of \( O(\log(n + m)) \), but reduces the size complexity from \( O(nm) \) to \( O((n + m)(\log(n + m))) \).

With Batcher’s Sort, the practical time complexity is \( O(\log^2(n + m)) \) and the size complexity is \( O((n + m)(\log^2(n + m))) \).
Vishkin suggested pipelining in this model and devised a way of doing so, even though data was going in both directions. Vishkin considered the inputs in each pulse to be virtual processors which he designated as $s$. The number of processors he designated as $p$. For sufficiently large $s$, the time complexity is $O(s/p)$.

1.3 BSR

BSR has roots in Naïve PRAM and Optimal BSR has roots in Vishkin’s PRAM. BSR and Optimal BSR are covered in the next two sections.

1.3.1 Naïve BSR

BSR was proposed by Akl[4] as an extension to the PRAM. Extension meaning that it extends the capabilities of the PRAM.
Figure 1.11: Naïve BSR.

Figure 1.11, with $n = 4$ and $m = 4$, shows BSR in a way that makes it easy to compare with the Naïve PRAM. The ADT’s are removed. All of the data is broadcast to all of the memory locations. In exchange for this, each datum carries a tag with it. The CAT’s compare these tags to limits to determine what data is selected. The selected data are then reduced. The vertical line symbolically represents a barrier at the CAT’s where the selection process occurs. The complexity is the same as for the Naïve PRAM, $\Theta(nm)$ for size and $\Theta(log(n + m))$ for time.

This BSR model is often called Naïve BSR because its complexity is the same as for the Naïve PRAM. This similar complexity between BSR and the PRAM has interesting implications in theory.
Since the PRAM is defined as having $\Theta(1)$ time complexity, and BSR has the same complexity as the PRAM, therefore BSR also has, at least for the study and comparison of algorithms, $\Theta(1)$ complexity.

BSR uses six variables to control the data that it processes. Two of these variables are related to the processors, and four of them are related to memory. The processor variables are the tag and the datum. When a write instruction is done, the tag and datum for each processor are broadcast to all of the memory locations. The tag can be the address to which the datum is to be written. However, tags can also be other things. A tag can be compared to a luggage tag at an airport designating where the luggage is to be sent, except that in BSR, the same luggage can be sent to more than one airport simultaneously.

The four memory variables are the limit, a comparator, a reduction operation, and a value. A comparator variable has nothing to do with comparator nodes discussed in Section 1.2.2.1.1. The four memory variables are explained further as follows:

**Limit:** The limit is a variable used by the CAT in the selection process. The limit can be the address of the memory location, or it can be something else. The CAT’s can have the same or different limits.

**Comparator:** The comparator is a relation used to compare the tags and the limits. The six possible comparators are $<,\leq,=,\geq,>,\neq$.

**Reduction Operation:** The Reduction Operation is an associative binary operation that is applied to the data which is selected. Typically, BSR is considered to have seven possible operations: summation, multiplication, maximum, minimum, logical AND, logical OR, and logical XOR.

**Value:** The value is the variable where a reduced result for a memory location is stored.

Control information designating the limits, comparator, and operation are sent to the CAT's prior to a broadcast of the tags and data. For example, a BSR instruction might consist of the following steps:

1. Send the appropriate limits to each CAT.

2. Send control information indicating the comparator to each CAT.
3. Send control information indicating the operation to each CAT.

4. Broadcast the tags and data to the CAT’s.

Examples of BSR problems are given in Section 2.2 with algorithms.

1.3.2 Optimal BSR

Lindon[13] took the BSR idea and applied it to the structure of Vishkin’s PRAM, filling in all of the working details to have, on paper, an implementation of BSR. Although Lindon’s model has six boxes, as opposed to Vishkin’s two boxes, it is still bounded by the sorting time, and has the same asymptotic time complexity as Vishkin’s, an impractical $O(log(n + m))$, or a practical $O(log^2(n + m))$, depending upon whether AKS Sort or Batcher’s Sort is used. Because Lindon’s BSR has the same complexity as Vishkin’s PRAM, which is optimal, Lindon’s BSR is referred to as Optimal BSR.

![Diagram](image)

Figure 1.12: Optimal BSR Topview.

In Figure 1.12, processor and memory records enter from the left, continue straight through as they are processed, and exit on the right, where the processor records are discarded. In Optimal BSR, it is assumed that all memory accesses are writes, and that reads can be accomplished with similar techniques. Processor records are in the form $(i, t, d)$, for index, tag, and datum. Memory records are in the form $(a, l, v)$, for address, limit, and value. A control line indicates the comparator and the
reduction operator. Each processor's tag is compared with each memory location's limit according to the control comparator. The ones that are selected are then reduced and stored in memory.

Figure 1.13: Optimal BSR Boxes.

Figure 1.13 shows the six boxes. Three are for sorting and one is for merging. Optimal BSR has two additional kinds of boxes called Prefix and Distribute. The Prefix Box performs the calculations based on the designated reduction operator. The Distribute Box distributes the data between the records. The Prefix and Distribute boxes are explained in more detail in the next two sections. Since the processor and memory records contain the unique processor index, i, and the unique memory location address, a, it is not necessary for other means to determine where they came from. For example, the record (0, t, d) came from processor 0. The record (m−1, t, v) came from, and is going to, memory location m−1. The t, d, l, and v variables are identified by index and address variables in the same record and do not need subscripts for identification. The input control record consists of (σ, Ρ), representing the comparator and the reduction operator, respectively.

Straightforward BSR would be to select the data, first, and then reduce it. But, Optimal BSR does this out of order. First the data is reduced, in the Prefix Box, then it is broadcast and selected in the Distribute Box. One more variable, r, for rank, facilitates this process. Once the processor records are sorted in Box A they are each given a unique rank, which does not necessarily have any correlation with their index. These ranks are added to the processor records so that the Distribute Box can use them in the selection process. The Prefix and Distribute Boxes are shown in the next two sections, then an
overview of all six boxes is given.

1.3.2.1 The Prefix Box

The Prefix Box is a combinational circuit based on prefix computations. A prefix computation reduces all of the data with a rank which is less than or equal to the instant rank. For example, prefix sums adds all of the data with a rank which is less than or equal to the instant rank.

Definition 1.1 Prefix Sums.

Given a finite set \( D = \{d_1, d_2, ..., d_n\} \) of numbers, the Prefix Sum, \( s_r \), for rank \( r \), is

\[
    s_r = \sum_{i=1}^{r} d_i \tag{1.4}
\]

A prefix sums example is shown here:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Datum</th>
<th>Prefix Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 1.1: A Prefix Sums Example.

The same definition applies to other Reduction Operations in addition to summation.

Definition 1.2 Prefix Computation.

Given a finite set \( D = \{d_1, d_2, ..., d_n\} \) of numbers, and a Reduction Operation \( \mathcal{R} \), the Prefix Computation, \( c_r \), for rank \( r \), is

\[
    c_r = \sum_{i=1}^{r} \mathcal{R} d_i \tag{1.5}
\]

The only difference between Equation 1.4 and Equation 1.5 is that the \( \mathcal{R} \) symbol has replaced the \( \sum \) symbol.
The Prefix Box, however, potentially goes in both directions. This other direction has been called reverse prefix computation or suffix computation. A suffix computation reduces all of the values with an rank which is greater than or equal to the instant rank.

**Definition 1.3 Suffix Computation.**

*Given a finite set \( D = \{d_1, d_2, \ldots, d_n\} \) of numbers, and a Reduction Operation \( \mathcal{R} \), the Suffix Computation, \( c_r \), for rank \( r \), is*

\[
c_r = \sum_{i=r}^{n} \mathcal{R}(d_i)
\]

(1.6)

Both directions together have been called a two-way prefix computation. This can also be called an *affix computation*, with affix = \{prefix, suffix\}.

**Definition 1.4 Two-Way Prefix Computation.**

*Given a finite set \( D = \{d_1, d_2, \ldots, d_n\} \) of numbers, and a Reduction Operation \( \mathcal{R} \), the Two-Way Prefix Computation, \( c_r \), for rank \( r \), is*

\[
c_r = \sum_{i=1}^{n} \mathcal{R}(d_i)
\]

(1.7)

Depending upon the comparator and the reduction operation, the Prefix Box performs either a prefix, suffix, or two-way computation.

![Figure 1.14: Prefix and Suffix Nodes.](image)

Figure 1.14 illustrates two possible nodes for a Prefix Box. Node (a) does a typical prefix computation, such as prefix sums. Inputs come from the left and from above left. They are summed and output on the right. Node (b) shows a suffix computation. Inputs come from the left and below left. Likewise, they are summed and output on the right. Depending upon the location in the Prefix Box, the nodes may have less than two inputs and two outputs.
Figure 1.15 shows two examples of the nodes assembled in a Prefix Box for $n = 8$. Box (a) only shows the configuration for prefix computation, and it has a prefix sums example. Each output value is the sum of its own input plus the input values above itself. Box (b) shows the actual Prefix Box. Computations can be prefix, suffix, or both, a control line indicates which. The last column of smaller nodes represents switches, which are used in two-way computations. Two-way computations are only used in Optimal BSR for finding the maximum or minimum values that are not equal to the limits. The regular nodes use the tags to find the maximum (or minimum) datum with a tag which is less than the instant tag and to find the maximum (or minimum) datum with a tag which is greater than the instant tag. The switch then finds the maximum (or minimum) datum from the two. A control line leads to a single node which leads to a table, the shaded box in the figure. Based on the control input of which comparator and which reduction operation to use, the table looks up whether to use prefix, suffix, or two-way prefix computation, and which formula, if any, the switches should use. It sends this information, via relay, to all of the nodes and switches.

The limits are not considered in the Prefix Box, and the tags are considered only for two-way computations. The selection process using the tags and limits is done in the Distribute Box, after the reducing computations are already finished.
1.3.2.2 The Distribute Box

The Distribute Box is a combinational circuit which broadcasts data and selects it for memory records. Unlike the Naive PRAM where processor and memory records are paired up by two’s, Optimal BSR can potentially have groups of processors and memory locations with the same tags and limits.

![Diagram of Box D with tags and limits]

Figure 1.16: Groups of Tags and Limits.

Figure 1.16, with $n = 8$ and $m = 8$, demonstrates that Vishkin’s PRAM method of pairing up processor records with memory records to exchange information after a merge does not work for Optimal BSR. Processor records, indicated with $t =$ for tag equals, enter from the upper left. Memory records, indicated by $l =$ for limit equals, enter from the lower left. The processor and memory records are merged and output on the right so that memory records line up underneath processor records who’s tags equal their limits. In this figure, for example, two processor records with $t = 2$ are passing data to four memory locations with $l = 2$. All other combinations are possible: A tag may not have a corresponding limit; a limit may not have a corresponding tag; or all of the tags and limits might be the same.

Since it is no longer possible for pairs of processor and memory records, which are adjacent to each other, to pass data, some other method has to be utilized. Leaders are designated for the groups and
these leaders are used with formulas to select the appropriate data, already reduced, for each limit.

This is done in the Distribute Box.

Each limit has five processor record leaders, designated in this report as First, Greatest, Preceding, Following, and Overall.

![Sample Distribute Leaders](image)

**Figure 1.17: Sample Distribute Leaders**

Figure 1.17 gives two example sets of processor record leaders. Example (a) is for \( l = 1 \) and Example (b) is for \( l = 2 \). The data is from the merge output of the previous figure. Horizontal dotted lines show the boundaries of the data at the beginning and ending of each example limit. Refer to this figure during the following definitions:

**First Leader:** The first processor record in which the tag equals the limit.

**Greatest Leader:** The greatest processor record in which the tag equals the limit.

**Preceding Leader:** The greatest processor record in which the tag precedes the limit.

**Following Leader:** The first processor record in which the tag follows the limit.

**Overall Leader:** The overall greatest processor record irregardless of the tags and limits.
Figure 1.18 demonstrates how leaders can be used to select reduced data. Prefix sums has been done so that each leader’s datum contains the accumulated data that is equal to and less than its tag. The area above the top horizontal line is considered to be reduced data for tags that are less than the limit. The area within the two horizontal lines is considered to be data for tags that are equal to the limit. The area after the bottom horizontal line is considered to be data for tags that are greater than the limit. Three leaders are pertinent in this case, and, it is easy to determine the reduced data, in this summation, for tags less, equal to, and greater than the limit:

- The summation of data less than the limit is equal to the *Preceding* datum, 10.
- The summation of data equal to the limit is equal to the *Greatest* datum minus the *Preceding* datum, $15 - 10 = 5$.
- The summation of data greater than the limit is equal to the *Overall* datum minus the *Greatest* datum, plus the *Preceding* datum, $30 - 15 + 10 = 25$.

Similar formulas apply to other situations. This is how Optimal BSR reduces the data first, and then selects it.
Figure 1.19: The Distribute Box.

Figure 1.19, with \( n = 8 \) and \( m = 8 \), shows Box E, the Distribute Box. It looks much like the Prefix Box. However, its function is different. It broadcasts every possible processor record among every possible memory record. While doing so, the five leaders are determined for each limit. In the last column, switches use formulas to determine the final selected results from the leaders. When the memory records leave the Distribute Box, they have the selected and reduced data in their \( v \) variables and they are ready to write it to memory. The last sort, in Box F, sends them to their correct destinations.

1.3.2.3 Optimal BSR Overview

This overview of Optimal BSR summarizes how the six boxes work together to broadcast, select, and reduce the data, though not in that order. The data is sorted and reduced and then merged with the memory records. Then, it is broadcast and selected. Finally, the memory records are sorted so that they can return to their original locations.
Figure 1.20 shows how the data is sorted and reduced. The data records enter in the form of 
(i, t, d), for index, tag, and datum. They are presumably presorted by index, but that does not have to 
be the case. Box A sorts them lexicographically by tag and datum. The tags are not necessarily in the 
same order as the indexes, so even if they arrive presorted by index, they still have to be sorted by tag 
and datum. A rank field is added and the records leave Box A in the form (i, t, r, d) for index, tag, rank, 
and datum. The rank is a unique number given to each record based on its line of output. The record 
exiting the top line is given Rank 1. The record exiting the bottom line is given Rank n.

The processor records, as also shown in Figure 1.20, enter Box B in the same form in which they 
exited Box A. A control record in the form (σ, R), for comparator and reduction operation, respectively, 
also enters Box B. Box B reduces all of the data in accordance with the control record. The processor 
records leave Box B in the same form of (i, t, r, d). However, the datum fields have potentially been 
changed to new reduced values. The rank fields are the same: The record on the top line still has Rank 
1 and the record on the bottom line still has Rank n.

The processor records are now ready to be merged with the memory records.
Figure 1.21: Overview of Memory Location Input.

Figure 1.21 shows how the memory records are sorted in preparation to be merged. They enter Box C in the form \( (a, l, v) \), for address, limit, and value. They are presumably presorted by address, but that does not have to be the case. Box C sorts them by limit. The limits are not necessarily in the same order as the addresses, so even if they arrive presorted by address, they still have to be sorted by limit. The records exit Box C in the same form, \( (a, l, v) \), ready to be merged with the processor records.

Figure 1.22: Overview of Distribution.

Figure 1.22 shows the merging and distribution. Processor records enter Box D from the upper left in the same form as they left Box B, \( (i, t, r, d) \). Memory records enter Box D from the lower left in the same form as they left Box C, \( (a, l, v) \). These records are stably merged based on their tags and limits. They leave Box D sorted on the tags and limits. When a tag equals a limit, the record with the tag comes before the record with the limit. When tags are equal, the record with the lowest rank comes
first. All of the records enter Box E in the same order and form as they left Box D. Box E broadcasts and selects the data for each limit, as described above in Section 1.3.2.2. The records leave Box E in the same order that they entered it, but the \( v \) fields in the memory records now contain the values to be written to memory. The rank fields are no longer needed in the processor records and are discarded, thus the processor records leaving Box E are back to the form \((i, t, d)\).

All that’s left to be done is to separate the memory records from the processor records so that they can be returned to their original locations.

![Diagram of Box F](image)

**Figure 1.23: Overview of Outputs.**

Figure 1.23 shows Box F, which separates the memory records from the processor records by sorting them. The inputs of Box F are the same outputs of Box E. Box F immediately increments the \( a \) variable of each memory record by \( n \). Then, Box F sorts all of the records on the \( i \) and \( a \) fields. This moves all of the processor records to the top and all of the memory records to the bottom. At the output of Box F, all of the processor records are discarded, as they are no longer needed. Immediately prior to output, the \( a \) variable of each memory record is decremented by \( n \) back to its original value. The memory records are then on the same lines as they were when they entered Optimal BSR in Box C. They have the values to be stored in their \( v \) fields, and they are ready to return to their memory locations.
1.3.2.4 Optimal BSR’s Complexity

The Prefix Box has $O(\log n)$ stages and the Distribute Box has $O(\log (n + m))$ stages. This means
that Optimal BSR is bounded by the sorting, the same as Vishkin’s PRAM in Section 1.2.2.3. Since
Optimal BSR is bounded by the sorting, and the complexity of the sorting depends upon the context,
this means that the complexity of Optimal BSR also depends upon the context. Immediately following
are the three contexts of complexity for Optimal BSR. They are very similar to the ones described in
Section 1.2.2.1.2 on AKS Sort.

$\Theta(1)$: PRAM is defined as having constant time. Optimal BSR has the same complexity as PRAM.
Therefore, Optimal BSR complexity is defined as being constant time. This context is useful in
the study of algorithms.

$O(\log n)$: Optimal BSR can use AKS Sort, which has a time complexity of $O(\log n)$. Therefore, the
time complexity of Optimal BSR is also $O(\log n)$. However, AKS Sort, because of its size, is
impractical to use.

$O(\log^2 n)$: Optimal BSR can use Batcher’s Sort, which has a time complexity of $O(\log^2 n)$. This is
Optimal BSR’s practical complexity.

1.3.3 Other BSR Models and Implementations

Other models and implementations of BSR exist, with other structures and as extensions of Naïve
BSR. For example, BSR$_k$, described by AkI[7], is referred to as multiple-criteria BSR, and it handles
problems involving multiple tags and limits per datum, which Optimal BSR cannot do. $k$ determines
how many criteria, how many pairs of tags and limits, are used to make a selection. These other models
and implementations of BSR are acknowledged, but not covered in this report.

General BSR in this report will mean a general model of BSR as described in Section 1.1 without
constraints such as the types of comparators and reduction operations allowed.
1.4 Complexity Issues

Complexity issues of BSR depend upon the context of the discussion. BSR has an imaginary time complexity of $O(1)$, by association with the defined complexity of the PRAM. BSR can be implemented with AKS sort for a time complexity of $O(\log(n + m))$, but this is impractical. BSR can be implemented with Batche's Sort for a time complexity of $O(\log^2(n + m))$ and this is practical.

<table>
<thead>
<tr>
<th>Model or Implementation</th>
<th>Time Complexity</th>
<th>Size Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRAM</td>
<td>$O(1)$ (By Definition)</td>
<td>(Undefined)</td>
</tr>
<tr>
<td>BSR</td>
<td>$O(1)$ (By Definition)</td>
<td>(Undefined)</td>
</tr>
<tr>
<td>AKS Sort</td>
<td>$O(\log N)$</td>
<td>$O(N\log N)$</td>
</tr>
<tr>
<td>Batcher's Sort</td>
<td>$O(\log^2 N)$</td>
<td>$O(N\log^2 N)$</td>
</tr>
<tr>
<td>AKS Sort, $N = n + m$</td>
<td>$O(\log(n + m))$</td>
<td>$O((n + m)\log(n + m))$</td>
</tr>
<tr>
<td>Batcher's Sort, $N = n + m$</td>
<td>$O(\log^2(n + m))$</td>
<td>$O((n + m)\log^2(n + m))$</td>
</tr>
<tr>
<td>Naive PRAM</td>
<td>$\Theta(\log(n + m))$</td>
<td>$\Theta(nm)$</td>
</tr>
<tr>
<td>Vishkin's PRAM with AKS Sort</td>
<td>$O(\log(n + m))$</td>
<td>$O((n + m)\log(n + m))$</td>
</tr>
<tr>
<td>Vishkin's PRAM with Batcher's Sort</td>
<td>$O(\log^2(n + m))$</td>
<td>$O((n + m)\log^2(n + m))$</td>
</tr>
<tr>
<td>Naive BSR</td>
<td>$\Theta(\log(n + m))$</td>
<td>$\Theta(nm)$</td>
</tr>
<tr>
<td>Optimal BSR with AKS Sort</td>
<td>$O(\log(n + m))$</td>
<td>$O((n + m)\log(n + m))$</td>
</tr>
<tr>
<td>Optimal BSR with Batcher's Sort</td>
<td>$O(\log^2(n + m))$</td>
<td>$O((n + m)\log^2(n + m))$</td>
</tr>
</tbody>
</table>

Table 1.2: A Summary of Complexities.

Table 1.2 summarizes the complexities of the various models and implementations, including the sorting circuits on which the complexities are based.

1.5 Statement of the Problem

To utilize a distributed software implementation to analyze the feasibility of creating a hardware implementation of Optimal BSR. Applications, performance, cost, and power will be considered. Published BSR algorithms will be analyzed and compared with their mathematical notations and combinational circuit implementations. The possibility of creating new algorithms will be considered. A PRAM simulation has been completed and this will be extended to include Optimal BSR. These hypotheses will be addressed:

**Hypothesis 1.1** Memory, itself, does not need to store $R$, $\sigma$, and $l_j$ and an actual memory record does not need to be input into BSR.
Hypothesis 1.2 The BROADCAST instruction needs to specify the memory address to be accessed.

Hypothesis 1.3 Setup information should not be distributed throughout the circuit separately from the other records.

Hypothesis 1.4 Pipelining is possible.

Hypothesis 1.5 With pipelining, R can be changed dynamically from within BSR.

Hypothesis 1.6 A BSR instruction set needs to be created.

Hypothesis 1.7 Not all published BSR algorithms fit Optimal BSR.

Hypothesis 1.8 Texture and luminosity graphics applications should be considered.

Hypothesis 1.9 Some reduction operations are never used.

Hypothesis 1.10 A system-wide logical clock should be considered.

Hypothesis 1.11 The object-oriented organization of the simulator should be explained.

Hypothesis 1.12 Running a larger case on a supercomputer should be considered.
CHAPTER 2 – REVIEW OF THE LITERATURE

Optimal BSR is introduced and described in detail in Lindon[13]. Other models of BSR are described in the literature, but the focus of this report is on the optimal model. Other literature describes BSR algorithms, and this chapter summarizes these algorithms as they relate to Optimal BSR. Various notations are useful in describing BSR algorithms. These notations will be explained in the next section and then published BSR algorithms will be reviewed in Section 2.2.

2.1 BSR Notations

Two types of pseudocode notation as well as two types of mathematical notation are useful in describing BSR algorithms. All four types are described in the following subsections.

Arrays and subscripts are both used, one for pseudocode and the other for mathematical notation. Given a processor record \((i,t,r,d)\), the fields can be referred to as \(i\), \(t[i]\), \(r[i]\), and \(d[i]\) or they can be referred to as \(i\), \(t_i\), \(r_i\), and \(d_i\). Likewise, give a memory record \((a,l,v)\), the fields can be referred to as \(a\), \(l[a]\), and \(v[a]\) or they can be referred to as \(a\), \(l_a\), and \(v_a\). Thus...
\[
\begin{align*}
t[i] & \equiv t_i; & r[i] & \equiv r_i; & d[i] & \equiv d_i; & l[a] & \equiv l_a; & \text{and} & v[a] & \equiv v_a.
\end{align*}
\]

2.1.1 Naïve BSR Notation

The notation used in Akl[4] is not consistent with the variable names used in Optimal BSR. However, it is useful as a pattern for a pseudocode applicable to BSR. The following memory location variable names are used:

- **data**: Corresponds to the Optimal BSR value variable, \(v\).
- **red**: Corresponds to the Optimal BSR Reduction Operation variable, \(R\).
- **sel**: Corresponds to the Optimal BSR comparator variable, \(\sigma\).
- **lim**: Corresponds to the Optimal BSR limit variable, \(l\).

This notation consists of a preparation phase and a broadcast phase. The preparation phase loads the memory location variables, like this:
c.red := sum
c.sel := <
c.lim := 10

The broadcast phase sends the tag and datum, like this:

```
BROADCAST tag, datum
```

This notation is acknowledged, but will not be used further in this report because of the inconsistency of the variable names. However, a pseudocode similar to this with consistent variable names is proposed in the next section.

### 2.1.2 BSR Pseudocode

This pseudocode is proposed as a way of describing BSR algorithms that is consistent across Naïve BSR and Optimal BSR. C programming language conventions are used because this will be consistent with C++ and Java programming code used later in this report.

Let constants defined below represent the possible comparator values:

```
LESS_THAN = 0;
LESS_THAN_OR_EQUAL_TO = 1;
EQUAL_TO = 2;
GREATER_THAN = 3;
GREATER_THAN_OR_EQUAL_TO = 4;
NOT_EQUAL_TO = 5;
```

Let constants defined below represent the possible Reduction Operation values:

```
SUMMATION = 0;
PRODUCT = 1;
MAXIMUM = 2;
MINIMUM = 3;
AND = 4;
OR = 5;
XOR = 6;
```

Let the comparator variable $\sigma$ be replaced by the variable name `comparator`. Similarly, let the Reduction Operation variable $\mathcal{R}$ be replaced by the variable name `operation`. `limit` will be used for $l$. Arrays often represent variables. For example, the following pseudocode prepares the variables for memory locations in array $v$:
operation = SUM;
comparator = LESS_THAN;
v[i].limit = 10;

All of the memory locations use the same operation and comparator, so it is not necessary to specify an array of operations or an array of comparators. \( i \) refers to the index of the processor executing the pseudocode. The code is executed in parallel, so all of the memory location variables are prepared. This notation assumes \( i = n \).

The broadcast instruction can now be called, like this:

\[
\text{BROADCAST}(t, d);
\]

It is uppercase by convention. \( t \) represents the tag, and \( d \) the datum. The code is executed in parallel, so all of the tags and data are broadcast.

This proposed pseudocode does not exactly match the records and flexibility of Optimal BSR, but it is sufficient to describe published algorithms, which will be done in Section 2.2.

### 2.1.3 Optimal BSR Notation

BSR has an elegant mathematical notation introduced by Lindon[13]. In this report, *BSR Notation* refers to the notation described in this section. Subscripts are necessary in this notation, as described in Section 2.

BSR is described mathematically as follows:

\[
v_a = \bigoplus_{1 \leq i \leq n} d_iv_a = \bigoplus_{1 \leq i \leq n} d_i
\]

Informally, the above formula says that for all \( v \) in \( a \), for all \( i \) in \( n \), if the tag for \( i \) passes the comparison with the limit for \( a \) then reduce the datum for \( i \), along with the other data which passes, and store it in \( v_a \).

When it is understood that the range of \( a \) is \( 1 \leq a \leq m \) and that the range of \( i \) is \( 1 \leq i \leq n \), then the BSR notation is simplified as:
\[ v_a = \mathcal{R}_{i,a} d_i \]  

(2.2)

Input is designated as \( x[i] \) or \( x_i \) in the examples, because input does not necessarily correspond to the data fields in processor records. \( x[i] \equiv x_i \).

### 2.1.4 One-Line BSR Notation

Gewali[11] first used a one-line mathematical notation for BSR which saves some space and will later make it easier to compare various broadcast statements. This notation is as follows:

\[ v_a = \mathcal{R}d_i \mid t \not= I_a \]  

(2.3)

### 2.2 Published BSR Algorithms

Numerous BSR algorithms have been published. This section briefly lists them in a manner which facilitates analyzing them as a group. The objective is not to see how cleverly BSR can be used, but rather to determine what kind of broadcast operations actually have applications. The algorithms are listed roughly in the chronological order in which they were published.

The variable names in published algorithms vary greatly, making it difficult to compare similar broadcast instructions. This chapter uses a standardized naming system. Processor indexes are always indicated by \( i \). Memory addresses are always indicated by \( a \). If there is one input array, it is \( x \). If there are two input arrays, the second one is \( y \). In special cases of numerous input arrays, they are designated by \( p, q, r, s, \) and \( t \) and are accompanied by a diagram. When a series of broadcast instructions are used in a single algorithm, interim arrays are stored beginning with \( a \) and proceeding alphabetically. A disadvantage of this system is that the array names usually have nothing to do with what they represent. An advantage of this system is that broadcast instructions from different algorithms can be compared directly.

As the algorithms get more complicated, only the unique broadcast instructions are listed.

A table summarizing the results follows the listing of algorithms.
Algorithm 2.2.1 Prefix Sums

Statement of the Problem: Prefix Sums is defined in Section 1.3.2.1.

Source: Akl[4]

Precondition: Input array $x[1..n]$. $n = m$.

Postcondition: A memory value array $v[1..m]$ containing the Prefix Sums.

BSR Instruction 1 of 1: Find the prefix sums.

Pseudocode:

\[
\begin{align*}
\text{operation} &= \text{SUM}; \\
\text{comparator} &= \text{LESS\_THAN\_OR\_EQUAL\_TO}; \\
v[i].\text{limit} &= i; \quad \text{// limit} = a \\
\text{BROADCAST}(i, x[i]); \quad \text{// tag} = i; \text{datum} = x[i]
\end{align*}
\]

Notation:

\[ v_a = \sum_{i \leq a} x_i \quad (2.4) \]

Comments: In this straightforward example, input array $x$ is the data which is broadcast.

Algorithm 2.2.2 Element Uniqueness

Statement of the Problem: Finds unique numbers.

Source: Akl[4]

Precondition: Input array $x[1..n]$. $n = m$.

Postcondition: A memory value array $v[1..m]$ with each location containing 1 if the corresponding datum is unique, or a number greater than 1 if it is not unique.

BSR Instruction 1 of 1: Find the unique numbers.

Pseudocode:

\[
\begin{align*}
\text{operation} &= \text{SUM}; \\
\text{comparator} &= \text{EQUAL\_TO}; \\
v[i].\text{limit} &= x[i]; \quad \text{// limit} = x[a] \\
\text{BROADCAST}(x[i], 1); \quad \text{// tag} = x[i]; \text{datum} = 1
\end{align*}
\]

Notation:

\[ v_a = \sum_{x[i] = x[a]} 1 \quad (2.5) \]

Comments: This algorithm counts the number of times each datum occurs. The input array $x$ becomes the tags and the limits. The broadcast data are 1.'s.
Algorithm 2.2.3  **Count Sort**

*Statement of the Problem:* Count sorts an input array of distinct numbers.

*Source:* Akhi

*Precondition:* Input array \( x[1..n] \), \( n = m \)

*Postcondition:* A memory value array \( v[1..m] \) of the counts of the lesser numbers, which can then be used as indexes to place the original numbers so that they are sorted.

**BSR Instruction 1 of 1:** Count the lesser numbers.

**Pseudocode:**

```plaintext
operation = SUM;
comparator = LESS_THAN;
v[i].limit = x[i];          // limit = x[a]
BROADCAST(x[i], 1);        // tag = x[i]; datum = 1
```

*Notation:*

\[
v_a = \sum_{x[0]<x[a]} 1
\]  \hspace{1cm} (2.6)

Algorithm 2.2.4  **Sieve of Eratosthenes**

*Statement of the Problem:* Finds prime numbers.

*Source:* Akhi

*Precondition:* Input array \( x[2..n] \), \( n = m \)

*Postcondition:* A memory value array \( v[2..m] \) with each location containing 1 if the corresponding datum is prime, or a value greater than 1 if it is not prime.

**BSR Instruction 1 of 1:** Find the prime numbers.

**Pseudocode:**

```plaintext
i += 2;
a += 2;
opration = SUM;
comparator = DIVIDES;        // not a listed comparator
v[i].limit = i;              // limit = a
BROADCAST(i, 1);             // tag = i; datum = 1
```

*Notation:*

\[
v_a = \sum_{a \% i = 0} 1
\]  \hspace{1cm} (2.7)

*Comments:* \( i \) and \( a \) are incremented so that the testing begins at 2 instead of at 0. This algorithm does not meet the stated criteria of Optimal BSR because modulo is not a defined comparator.
Algorithm 2.2.5 Maximal Sum Subsequence

Statement of the Problem: Finds the maximal sum subsequence.

Source: \( A[i,j] \)

Precondition: Input array \( x[1...n] \) containing positive and negative numbers. \( n = m \).

Postcondition: A memory value array \( v[1...m] \) containing the maximal sum subsequence.

**BSR Instruction 1 of 4: Find the prefix sums.**

Pseudocode:

\[
\begin{align*}
\text{a[i]} &= x[i]; & \text{// original data saved in array a} \\
\text{operation} &= \text{SUM}; \\
\text{comparator} &= \text{LESS\_THAN\_OR\_EQUAL\_TO}; \\
\text{v[i].limit} &= i; & \text{// limit = a} \\
\text{BROADCAST}(i, x[i]); & \text{// tag = i; datum = x[i]} \\
\text{b[i]} &= v[i]; & \text{// result saved in array b}
\end{align*}
\]

Notation:

\[
\begin{equation}
\sum_{i \leq a} 1
\end{equation}
\]  

**BSR Instruction 2 of 4: Find the largest sum on the right.**

Pseudocode:

\[
\begin{align*}
\text{x[i]} &= b[i]; & \text{// previous result becomes new input} \\
\text{operation} &= \text{MAXIMUM}; \\
\text{comparator} &= \text{GREATER\_THAN\_OR\_EQUAL\_TO}; \\
\text{v[i].limit} &= i; & \text{// limit = a} \\
\text{BROADCAST}(i, x[i]); & \text{// tag = i; datum = x[i]} \\
\text{c[i]} &= v[i]; & \text{// result saved in array c}
\end{align*}
\]

Notation:

\[
\begin{equation}
\bigcap_{i \geq a} x[i]
\end{equation}
\]  

**BSR Instruction 3 of 4: Find the index of the largest sum on the right.**

Pseudocode:

\[
\begin{align*}
\text{x[i]} &= v[i]; & \text{// previous result becomes new input} \\
\text{y[i]} &= \text{b[i]}; & \text{// array b also becomes input} \\
\text{operation} &= \text{MAXIMUM}; \\
\text{comparator} &= \text{EQUAL\_TO}; \\
\text{v[i].limit} &= x[i]; & \text{// limit = x[a]} \\
\text{BROADCAST}(y[i], i); & \text{// tag = y[i]; datum = i} \\
\text{d[i]} &= v[i]; & \text{// result saved in array d}
\end{align*}
\]
Notation: \[ v_a = \bigcap_{y[i] = x[a]} i \]  

(2.10)

BSR Instruction 4 of 4: Find the maximal sum subsequence.

Pseudocode:

// x[i] contains each maximal subsequence sum
x[i] = c[i] - b[i] + a[i];
operation = MAXIMUM;
comparator = GREATER_THAN;
v[i].limit = 0; // limit = 0
BROADCAST(i, x[i]); // tag = i; datum = x[i]
// v[a] now contains the overall maximal subsequence sum

Notation: \[ v_a = \bigcap_{i \geq 0} x[i] \]  

(2.11)

Comments: The endpoints can be determined form data in the arrays. In multiple instructions in this report, the intermittent arrays are selected in alphabetical order. These intermittent arrays could potentially be stored in shared memory or in local processor memory.

Algorithm 2.2.6 Maximal Vectors

Statement of the Problem: Finds vectors.

Source: Aki[i]

Precondition: Input arrays x[0...n-1] and y[0...n-1] representing the x and y values, respectively, of vectors. n = m.

Postcondition: A memory value array v[0...n-1] containing v[a] = y[a] if the corresponding vector is maximal, or v[a] > y[a] if it is not maximal.

BSR Instruction 1 of 1: Find the maximal vectors.

Pseudocode:

operation = MAXIMUM;
comparator = GREATER_THAN_OR_EQUAL_TO;
v[i].limit = x[i]; // limit = x[a]
BROADCAST(x[i], y[i]); // tag = x[i]; datum = y[i]

Notation: \[ v_a = \sum_{x_i \geq x_a} y_i \]  

(2.12)
Comments: If the resulting \( v[a] \) equals the input \( y[i] \) for \( a = i \), then the vector represented by \( x[i] \), \( y[i] \) is maximal. Otherwise, \( v[a] \) represents the greater \( y \) value of a vector which also has a greater \( x \) value and the instant vector is not maximal.

Algorithm 2.2.7 Convex Hull

Statement of the Problem: Finds the points of a convex hull.

Source: Akl[4]

Precondition: Parallel input arrays \( x[1...n] \) containing the sorted indexes of the points and \( y[1...n] \) containing the angles between each instant point and every other point. \( n = m^2 \).

Postcondition: Output array \( y[1...m] \) containing the largest angle between points for each point. If this angle is greater than \( \pi \), then the instant point is part of the convex hull.

BSR Instruction 1 of 1: Find the maximum angles between each point and every other point.

Pseudocode:

\[
\text{operation = MAXIMUM;}
\]
\[
\text{comparator = EQUAL_T0;}
\]
\[
\text{v[x[i]].limit = x[i];} \quad \text{// limit = a;}
\]
\[
\text{BROADCAST(x[i], y[i]);} \quad \text{// tag = x[i]; datum = y[i]}
\]

Notation:

\[
v_a = \bigcap_{x[i]=a} y[i] \quad (2.13)
\]

Comments: If each resulting angle in \( v[a] \) is greater than \( \pi \), then the corresponding point is in the convex hull.

Algorithm 2.2.8 Maximal Sum Subrectangle

Statement of the Problem: Finds the maximal sum subrectangle in a two-dimensional array.

Source: Akl[5]
**Precondition:** An array \( x[0...n-1] \) equivalent to array \( x[0...p-1,0...q-1] \), \( n = p \times q \), containing the input data. See notations in the figure. \( p[i] \) and \( p_i \) refer to the \( p \)-index of processor \( i \), and \( q[i] \) and \( q_i \) refer to the \( q \)-index of processor \( i \).

**Postcondition:** An array \( v[0...m-1] \) equivalent to array \( v[0...q-1,0...q-1,0...p] \) equivalent to array \( v[0...q^2(p+1)-1] \), \( m = q^2(p+1) \), containing maximal sum subsequences of bands of the 2D input data. These bands can be further processed by the one-dimensional maximal sum subsegment algorithm to obtain the final result. See the citation for further explanation.

**BSR Instruction 1 of 2:** Sum from \( x[r,t] \) to the end of the array.

**Pseudocode:**

```
op = SUMMATION;
comparator = GREATER_THAN_OR_EQUAL_TO;
v[r,s,t].limit = r + qt; // limit = r + qt
BROADCAST(p[i]*p+q[i], x[p[i], q[i]]); // tag = p[i]*p+q[i]
                      // datum = x[p[i], q[i]]
```

**Notation:**

\[
v_{r,s,t} = \sum_{p, p+q \geq r+qt} x[p_i, q_i] \quad (2.14)
\]

**BSR Instruction 2 of 2:** Sum from \( x[s+1,t] \) to the end of the array.

**Pseudocode:**

```
a = v; // array a stores the previous result
// input array x remains the same
op = SUMMATION;
comparator = GREATER_THAN;
v[r,s,t].limit = s + qt; // limit = s + qt
BROADCAST(p[i]*p+q[i], x[p[i], q[i]]); // tag = p[i]*p+q[i]
                      // datum = x[p[i], q[i]]
a[r,s,t] = a[r,s,t] - v[r,s,t];
```
Notation:

\[ v_{r,s,t} = \sum_{p_i, p_i + q_i > s + q_t} x[p_i, q_i] \]  
(2.15)

Comments: Array \( a \) now contains \( q^2 \) subsegments, each of which has the column sums of a band of \( x[p, q] \). See Akl[5] for the rest of the details on how this algorithm works.

Algorithm 2.2.9 External Watchman Routes

Statement of the Problem: Finds the shortest external route where all of the points on a convex polygon are visible.

Source: Gewali[11]

Preface: This algorithm introduces a concurrent read instruction, for reading among processors. The special index \( j \) is used to indicate a processor which is doing the reading.

BSR Instruction 1 of 3: Concurrent read between processors.

Pseudocode:

The existing pseudocode is not sufficient to indicate reads.

Notation:

\[ x_j = \sum_{i=j} x[i] \]  
(2.16)

BSR Instruction 2 of 3: With two input arrays, \( x \) and \( y \).

Pseudocode:

operation = MINIMUM;
comparator = GREATER_THAN_OR_EQUAL_TO;
v[i].limit = x[i]; \quad \text{// limit} = x[a];
BROADCAST(y[i], y[i]); \quad \text{// tag} = y[i]; \text{datum} = y[i]

Notation:

\[ v_a = \bigcup_{y[i] \geq x[a]} y[i] \]  
(2.17)

BSR Instruction 3a of 3: With two input arrays, \( x \) and \( y \).

Pseudocode:

operation = MAXIMUM;
comparator = EQUAL_TO;
v[i].limit = v[i]; \quad \text{// limit} = v[a];
BROADCAST(x[i], y[i]); \quad \text{// tag} = x[i]; \text{datum} = y[i]

Notation:

\[ v_a = \bigcap_{x[i] = v[a]} y[i] \]  
(2.18)
**Comments:** The above instruction could also be perceived as having three input arrays, \( x, y, \) and \( z \), like this:

**BSR Instruction 3b of 3:** With three input arrays, \( x, y, \) and \( z \).

**Pseudocode:**
\[
\begin{align*}
  z[i] &= v[i]; & \text{// save the previous result} \\
  \text{operation} &= \text{MAXIMUM}; \\
  \text{comparator} &= \text{EQUAL_TO}; \\
  v[i].\text{limit} &= z[i]; & \text{// limit = } z[a]; \\
  \text{BROADCAST}(x[i], y[i]); & \text{// tag = } x[i]; \text{ datum = } y[i]
\end{align*}
\]

**Notation:**
\[
v_o = \bigcap_{z[i]=z[o]} y[i]
\]  
\[\text{(2.19)}\]

**Algorithm 2.2.10 Overlapping Intervals**

**Statement of the Problem:** Given a set of intervals on a line, determine if any of them overlap.

**Source:** Akl[?]  

**Comments:** This problem is solved in a manner similar to the Maximal Vectors Problem in Algorithm 2.2.6. See the citation for more information.

**Algorithm 2.2.11 Union of Intervals**

**Statement of the Problem:** Given a set of intervals on a line, determine the size of the interval from their union.

**Source:** Akl[?]  

**Comments:** This problem is solved using sorting, prefix sum, and summation, examples of which are in algorithms 2.2.3, 2.2.1, and 2.2.8. See the citation for more information.

**Algorithm 2.2.12 \( \epsilon \)-Closeness**

**Statement of the Problem:** Given a set of numbers, determine if any two are at a distance less than \( \epsilon \) from each other.

**Source:** Akl[?]  

**Preface:** This problem is solved using sorting, which is described in Algorithm 2.2.3, and by finding the differences between neighboring numbers in the sorted list and the minimum of such differences. Explanations of these two additional types of broadcast follow.
**BSR Instruction 1 of 2:** Determine the differences between adjacent numbers. \( n = m \).

**Pseudocode:**

```
operation = SUMMATION;
comparator = EQUAL_TO;
v[i].limit = i + 1;       // limit = i + 1;
BROADCAST(i, x[i]);     // tag = i; datum = x[i]
difference = v[i] - x[i]
```

**Notation:**

\[
v_a = \sum_{i \geq a+1} x[i] \quad (2.20)
\]

**BSR Instruction 2 of 2:** Determine the minimum datum. \( n = m \).

**Pseudocode:**

```
operation = MINIMUM;
comparator = GREATER_THAN;
v[i].limit = 0;           // limit = 0;
BROADCAST(i, x[i]);      // tag = i; datum = x[i]
```

**Notation:**

\[
v_a = \bigcup_{i \geq 0} x[i] \quad (2.21)
\]

**Comments:** In Instruction 1, the operation does not matter because there will only be one datum matched per memory location. The value written to \( v[n] \) is an issue. Optimal BSR writes the identity for the operation, which does not work in this case. Other models of BSR up to this point do not define what happens. See the citation for more information on \( \epsilon \)-Closeness.

**Algorithm 2.2.13 Maximum Gap**

**Statement of the Problem:** Given a set of sorted numbers, determine the maximum gap between any two neighboring ones.

**Source:** Akl[7]

**Comments:** This problem is similar to Algorithm 2.2.12, \( \epsilon \)-Closeness, except the maximum is found instead of the minimum.

**Algorithm 2.2.14 Intersection of Two Convex Polygons**

**Source:** Akl[7]

**Comments:** The solution to this problem uses types of previously discussed broadcasts. See the citation for details.
Algorithm 2.2.15 Construct a Voronoi Diagram

Source: Akl[7]

Comments: The solution to this problem uses Algorithm 2.2.7, Convex Hull. See the citation for details.

Algorithm 2.2.16 ECDF Searching

Source: Akl[7]

Comments: This is a k = 2 problem and cannot be done on Optimal BSR.

Algorithm 2.2.17 2-Set Dominance

Source: Akl[7]

Comments: This is a k = 2 problem and cannot be done on Optimal BSR.

Algorithm 2.2.18 Intersection of Isothetic Line Segments

Source: Akl[7]

Comments: This is a k = 2 problem and cannot be done on Optimal BSR.

Algorithm 2.2.19 Vertical Segment Visibility

Source: Akl[7]

Comments: This is a k = 3 problem and cannot be done on Optimal BSR.

Algorithm 2.2.20 High Dimensional Maximal Elements

Source: Akl[7]

Comments: This is a k > 1 problem and cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.21 Rectangle Containment in d-Dimensional Space

Source: Akl[7]

Comments: This is a k > 1 problem and cannot be done on Optimal BSR. See the citation for more information.

---

1 See Section 1.3.3 on Page 26 for a description of problems involving k > 1.
Algorithm 2.2.22 Rectangle Enclosure Counting in $\mathbb{R}^d$

Source: Akl[7]

Comments: This is a $k > 1$ problem and cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.23 Rectangle Intersection Counting in $\mathbb{R}^d$

Source: Akl[7]

Comments: This is a $k > 1$ problem and cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.24 Histogram

Statement of the Problem: Creates a histogram of the grey values in an image.

Source: Melter[15]

Precondition: Input array $x[1...n]$ containing the grey values of the pixels in an image. $n = m$.

Postcondition: Output array $v[1...m]$ containing the count for the corresponding grey value.

Comments: The solution to this problem uses Algorithm 2.2.2, Element Uniqueness. See the citation for details.

Algorithm 2.2.25 All Nearest Neighbors and Furthest Pairs

Source: Melter[15]

Comments: This is a $k = 2$ problem and cannot be done on Optimal BSR.

Algorithm 2.2.26 Distance Transform

Statement of the Problem: Finds the shortest distance in a two-dimensional array between each 0 datum and a 1 datum.

Source: Melter[15]
Precondition: An array \( x[1...n] \) equivalent to array \( x[1..., q, 1...p] \), \( n = p \times q \), containing the input data of 1's and 0's, representing black pixels and white pixels, respectively. See notations in the figure. \( r[i] \) and \( r_i \) refer to the r-index of processor \( i \), and \( s[i] \) and \( s_i \) refer to the s-index of processor \( i \). \( n = m \).

Postcondition: An array \( v[1..m] \) equivalent to array \( v[1..., q, 1...p] \), containing \( r[a] \) and \( r_a \) refer to the r-index of memory address \( a \), and \( s[a] \) and \( s_a \) refer to the s-index of memory address \( a \).

**BSR Instruction 1 of 4:** Find the minimum datum in array \( b \) in the same column in the same or a greater row.

**Pseudocode:**

\[
\begin{align*}
a[i] &= x[i];  \\
&\quad \text{// store original input in array a} \\
&\quad \text{// array b contains the row for a black pixel or infinity for a white pixel} \\
&\text{if}(x[r,s] == 1) b[r,s] = r; \\
&\text{else} b[r,s] = n + 1; \\
x[i] &= b[i];  \\
&\quad \text{// array b becomes the input} \\
&\text{operation} = \text{MINIMUM}; \\
&\text{comparator} = \text{GREATER_THAN_OR_EQUAL_TO}; \\
v[r,s].\text{limit} = r;  \\
&\quad \text{// limit} = r; \\
&\text{BROADCAST}(r, x[r,s]);  \\
&\quad \text{// tag} = r; \text{ datum} = x[r,s] \\
c[i] &= v[i];  \\
&\quad \text{// save the result in array c}
\end{align*}
\]

**Notation:**

\[
\begin{equation}
v_{r_a,s_a} = \bigcup_{r \geq r_a} x[r_i,s_i] \tag{2.22}
\end{equation}
\]

**BSR Instruction 2 of 4:** Find the maximum datum in array \( b \) in the same column in the same or a lesser row.

**Pseudocode:**

\[
\begin{align*}
&\text{// put negative infinity for each white pixel in array b} \\
&\text{if}(a[r,s] == 0) b[r,s] = 0; \\
&\quad \text{// array b becomes the input for the next broadcast instruction} \\
x[i] &= b[i]; \\
&\quad \text{operation} = \text{MAXIMUM};
\end{align*}
\]
comparator = LESS_THAN_OR_EQUAL_TO;
v[r, s].limit = r; // limit = r;
BROADCAST(r, x[r, s]); // tag = r; datum = x[r, s]
d[i] = v[i]; // save the result in array d
// put the column distances between 0’s and the nearest 1 in array e
e[r, s] = 3n;
if (c[r, s] < n + 1) e[r, s] = c[r, s] - r;
if (d[r, s] > 0 && r - d[r, s] < e[r, s]) e[r, s] = r - d[r, s];

Notation:
\[ v_{r,a,s_a} = \bigcap_{r_i \leq r_a} x[r_i,s_i] \] (2.23)

BSR Instruction 3 of 4: Find the minimum datum in array \( f \) in the same row in the same or a greater column.

Pseudocode:

\[
\begin{align*}
f[r, s] &= e[r, s] + s; & \text{// prepare array } f \\
x[i] &= f[i]; & \text{// array } f \text{ becomes the input} \\
op = \text{MINIMUM}; \\
comparator &= \text{GREATER_THAN_OR_EQUAL_TO}; \\
v[r, s].\text{limit} &= s; & \text{// limit } = s; \\
\text{BROADCAST}(s, x[r, s]); & \text{// tag } = s; \text{ datum } = x[r, s] \\
g[i] &= v[i]; & \text{// save the result in array } g
\end{align*}
\]

Notation:
\[ v_{r,a,s_a} = \bigcup_{s_i \geq s_a} x[r_i,s_i] \] (2.24)

BSR Instruction 4 of 4: Find the maximum datum in array \( f \) in the same row in the same or a lesser column.

Pseudocode:

\[
\begin{align*}
&\text{// prepare array } h, \text{ which becomes the input for the next broadcast instruction} \\
h[r, s] &= e[r, s] - s; \\
x[i] &= h[i]; \\
op &= \text{MINIMUM}; \\
comparator &= \text{LESS_THAN_OR_EQUAL_TO}; \\
v[r, s].\text{limit} &= s; & \text{// limit } = s; \\
\text{BROADCAST}(s, x[r, s]); & \text{// tag } = s; \text{ datum } = x[r, s] \\
&\text{// combine the minimum column and row distances in array } v \\
v[r, s] &= \min(g[r,s] - s, s + v[r,s]);
\end{align*}
\]

Notation:
\[ v_{r,a,s_a} = \bigcup_{s_i \leq s_a} x[r_i,s_i] \] (2.25)
Algorithm 2.2.27 Medial Axis Transform

*Source:* Melter[15]

*Comments:* The citation reports this as a $k=3$ problem which cannot be done on Optimal BSR. However, Lee[12] showed later that this problem is equivalent to Algorithm 2.2.26, Distance Transform.

Algorithm 2.2.28 Area and Perimeter of Isooriented Rectangles

*Source:* Melter[15]

*Comments:* The solution to this problems uses types of previously discussed broadcasts. See the citation for details.

Algorithm 2.2.29 Discrete Voronoi Diagram for Labeled Images

*Source:* Melter[15]

*Comments:* This is a $k=2$ problem and cannot be done on Optimal BSR.

Algorithm 2.2.30 Parenthesis Matching

*Statement of the Problem:* Finds the pairs of matching parentheses in a given legal sequence.

*Source:* Stojmenovic[23]

*Precondition:* An array $x[1...n]$ of data containing 1 or -1 for '(' or ')', respectively. $n = m$.

*Postcondition:* An array $v[1...m]$ of data containing the index of the corresponding matching parenthesis.

**BSR Instruction 1 of 4:**

*Pseudocode:*

```plaintext
a[i] = x[i];  // save the input in array a
operation = SUMMATION;
comparator = LESS_THAN_OR_EQUAL_TO;
v[i].limit = i;  // limit = a;
BROADCAST(i, x[i]);  // tag = i; datum = x[i]
b[i] = v[i];  // save the result into array b
if(x[i] == -1) b[i] = b[i] + 1;
```

*Notation:*

$$v_a = \sum_{i\leq a} x[i]$$  \hspace{1cm} (2.26)

**BSR Instruction 2 of 4:**

*Pseudocode:*
v[i] = -1;
c[i] = a[i] - 1/i;
x[i] = c[i];
operation = MAXIMUM;
comparator = LESS_THAN;
v[i].limit = x[i]; // limit = x[a];
BROADCAST(x[i], x[i]); // tag = x[i]; datum = x[i]
d[i] = v[i]; // save the result into array d

Notation: \[ v_a = \bigcap_{x[i]<a} x[i] \] (2.27)

BSR Instruction 3 of 4:
Pseudocode:

v[i] = 0;
x[i] = c[i]; // array c becomes new input
y[i] = d[i]; // array d also becomes new input
operation = MAXIMUM;
comparator = EQUAL_TO;
v[i].limit = y[i]; // limit = y[a];
BROADCAST(x[i], i); // tag = x[i]; datum = i
e[i] = v[i]; // save the result into array e

Notation: \[ v_a = \bigcap_{x[i]<a} i \] (2.28)

BSR Instruction 4 of 4:
Pseudocode:

v[i] = 0;
x[i] = e[i];
operation = MINIMUM;
comparator = EQUAL_TO;
v[i].limit = i; // limit = a;
BROADCAST(x[i], i); // tag = x[i]; datum = i
f[i] = v[i]; // save the result in array f
if(a[i] == -1) v[i] = e[i];
else v[i] = f[i];

Notation: \[ v_a = \bigcup_{x[i]<a} i \] (2.29)

Comments: See the citation for the details on how the algorithm works.
Algorithm 2.2.31 Decoding Binary Trees

Source: Stojmenović[23]

Preface: This problem is solved using types of previously discussed broadcasts. See the citation for details. However, this problem also uses two slightly different broadcasts where the operation is minimum, examples of which follow.

**BSR Instruction 1 of 2:**

**Pseudocode:**

\[
\begin{align*}
\text{operation} & = \text{MINIMUM}; \\
\text{comparator} & = \text{EQUAL_TO}; \\
v[i].\text{limit} & = y[i]; \quad \text{// limit} = y[i]; \\
\text{broadcast}(x[i], i); & \quad \text{// tag} = x[i]; \text{ datum} = i
\end{align*}
\]

**Notation:**

\[ v_a = \bigcup_{x[i]=y[i]} i \] \hspace{1cm} (2.30)

**BSR Instruction 2 of 2:**

**Pseudocode:**

\[
\begin{align*}
\text{operation} & = \text{MINIMUM}; \\
\text{comparator} & = \text{EQUAL_TO}; \\
v[i].\text{limit} & = y[i] + 1; \quad \text{// limit} = y[i] + 1; \\
\text{broadcast}(i, x[i]); & \quad \text{// tag} = i; \text{ datum} = x[i]
\end{align*}
\]

**Notation:**

\[ v_a = \bigcup_{i=y[i]+1} x[i] \] \hspace{1cm} (2.31)

Algorithm 2.2.32 Generating Binary Trees

Source: Stojmenović[23]

Preface: This problem is solved using types of previously discussed broadcasts. See the citation for details. However, this problem also uses a no-criteria broadcast, and a variation where the operation is maximum, examples of each follow.

**BSR Instruction 1 of 2:** Determine the maximum of all values. \( n = m \) or \( n \neq m \).

**Pseudocode:**

\[
\begin{align*}
\text{operation} & = \text{MAXIMUM}; \\
\text{comparator} & = \text{GREATER_THAN_OR_EQUAL_TO}; \\
v[i].\text{limit} & = 0; \quad \text{// limit} = 0; \\
\text{broadcast}(i, x[i]); & \quad \text{// tag} = i; \text{ datum} = x[i]
\end{align*}
\]
**Notation:**

\[ v_a = \bigcap_{i \geq 0} x[i] \]  \hspace{1cm} (2.32)

**Comments:** The above broadcast puts the maximum value in each of the memory locations. The artificial criterion makes sure that every datum is selected for reduction.

**BSR Instruction 2 of 2:**

**Pseudocode:**

```plaintext
operation = MAXIMUM;
comparator = EQUAL_TO;
v[i].limit = y[i]; // limit = y[i];
BROADCAST(i, x[i]); // tag = i; datum = x[i]
```

**Notation:**

\[ v_a = \bigcap_{i=y[i]} x[i] \]  \hspace{1cm} (2.33)

---

**Algorithm 2.2.33 Reconstruction of a Binary Tree From Its Traversals**

**Source:** Stojmenović[23]

**Preface:** This problem is solved using types of previously discussed broadcasts. See the citation for details. However, this problem also uses one new type and four variations of previous types, examples of which follow.

**BSR Instruction 1 of 5:**

**Pseudocode:**

```plaintext
operation = MINIMUM;
comparator = EQUAL_TO;
v[i].limit = x[i] + 1; // limit = x[a] + 1;
BROADCAST(x[i], i); // tag = x[i]; datum = i
```

**Notation:**

\[ v_a = \bigcup_{i=x[a]+1} i \]  \hspace{1cm} (2.34)

**BSR Instruction 2 of 5:**

**Pseudocode:**

```plaintext
operation = OR;
comparator = EQUAL_TO;
v[i].limit = i; // limit = a;
BROADCAST(y[i], x[i]); // tag = y[i]; datum = x[i]
```
Notation: \[ v_a = \bigvee_{y[i]=a} x[i] \] \hfill (2.35)

Comments: Above, $\vee$ represent logical OR and the data type in $x[i]$ is boolean.

**BSR Instruction 3 of 5:**

Pseudocode:
- operation = MAXIMUM;
- comparator = LESS_THAN_OR_EQUAL_TO;
- $v[i].limit = i$; \hfill // limit = a;
- BROADCAST$(i, x[i])$; \hfill // tag = i; datum = $x[i]$

Notation: \[ v_a = \bigcap_{i\leq a} x[i] \] \hfill (2.36)

**BSR Instruction 4 of 5:**

Pseudocode:
- operation = MINIMUM;
- comparator = EQUAL_TO;
- $v[i].limit = y[i]$; \hfill // limit = $y[a]$;
- BROADCAST$(i, x[i])$; \hfill // tag = i; datum = $x[i]$

Notation: \[ v_a = \bigcup_{i=y[a]} x[i] \] \hfill (2.37)

**BSR Instruction 5 of 5:**

Pseudocode:
- operation = MINIMUM;
- comparator = EQUAL_TO;
- $v[i].limit = i$; \hfill // limit = a;
- BROADCAST$(y[i], x[i])$; \hfill // tag = $y[i]$; datum = $x[i]$

Notation: \[ v_a = \bigcup_{y[i]=a} x[i] \] \hfill (2.38)

Algorithm 2.2.34 *All Nearest Smaller Values*

Source: Xiang[25]

Preface: A special case of the problem is solved. The following new broadcast instructions are used in this algorithm. See the citation for details.

**BSR Instruction 1 of 3:** Convert array data to a bitmap. $n = m$.

Pseudocode:
operation = OR;
comparator = LESS_THAN;
v[i].limit = x[i];   // limit = x[a];
BROADCAST(x[i], 2^i);   // tag = x[i]; datum = 2^i

Notation: \[ v_a = \bigvee_{x[i] < x[a]} 2^i \] (2.39)

**BSR Instruction 2 of 3:** Find nearest lower set bit in a bitmap.

**Pseudocode:**

operation = MAXIMUM;
comparator = LESS_THAN_OR_EQUAL_TO;
v[i].limit = x[i];   // limit = x[a];
BROADCAST(2^i, i);   // tag = 2^i; datum = i

Notation: \[ v_a = \bigcap_{2^i \leq x[a]} i \] (2.40)

**BSR Instruction 3 of 3:** Find nearest higher set bit in a bitmap.

**Pseudocode:**

operation = MINIMUM;
comparator = LESS_THAN_OR_EQUAL_TO;
v[i].limit = x[i];   // limit = x[a];
BROADCAST(2^i, i);   // tag = 2^i; datum = i

Notation: \[ v_a = \bigcup_{2^i \leq x[a]} i \] (2.41)

**Comments:** In the citation, solutions are also given for other models of BSR.

Algorithm 2.2.35 **Maximal Sum Subsegment Problem**

**Source:** Bergogine[9]

**Comments:** This algorithm reviews previous broadcast instructions, defines a new operation, \( \widehat{\sum} \), which is not present in Optimal BSR. See the citation for more information.

Algorithm 2.2.36 **2D Maximal Sum Subsegment Problem**

**Source:** Bergogine[9]

**Comments:** This is a k > 1 problem and cannot be done on Optimal BSR. See the citation for more information.
Algorithm 2.2.37 Longest Increasing Sequence Problem

Source: Bergogne[9]

Comments: This is a $k > 1$ problem and cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.38 Sequence Alignment Problem

Source: Semé[21]

Comments: This is a $k > 1$ problem and cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.39 Longest Increasing Subsequence

Source: Myoupo[16]

Comments: This is a $k > 1$ problem and cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.40 Longest Common Subsequence

Source: Myoupo[16]

Comments: This is a $k > 1$ problem and cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.41 Sorting Vertices of a Multidimensional Space Lexicographically

Source: Xiang[28]

Comments: This problem uses operations which cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.42 Three Counting Problems for a Relational Database

Source: Xiang[28]

Comments: This problem uses operations which cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.43 Decoding a Binary Tree from its $i - p$ Sequence
Source: Xiang[26]

Comments: This problem is solved using types of previously discussed broadcasts. See the citation for details.

Algorithm 2.2.44 Drawing a Binary Tree from its $i-p$ Sequence

Source: Xiang[26]

Comments: This problem is solved using types of previously discussed broadcasts. See the citation for details.

Algorithm 2.2.45 Rearranging Scattered Information

Source: Xiang[27]

Comments: This is a $k = 2$ problem. See the citation for details.

Algorithm 2.2.46 $k$-compaction

Source: Xiang[27]

Comments: This algorithm uses prefix sums, which has already been discussed. See the citation for details.

Algorithm 2.2.47 Extremal Search of Convex Polygons

Source: Myoupo[17]

Comments: This algorithm uses BSR instructions which have already been discussed. See the citation for details.

Algorithm 2.2.48 Vertex-Vertex Pairs of Convex Polygons

Source: Myoupo[17]

Preface: This problem is solved using types of previously discussed broadcasts. See the citation for details. However, this problem also uses a slightly different broadcast where the operation is maximum.

BSR Instruction 1 of 1

Pseudocode:

operation = MAXIMUM;
comparator = EQUAL_TO;

v[i].limit = y[i]; // limit = y[i];

BROADCAST(x[i], z[i] + 1); // tag = x[i]; datum = z[i] + 1
Notation: \[ v_a = \bigcap_{i=1}^{n} z[i] + 1 \] (2.42)

Algorithm 2.2.49 Vector Sum of Two Convex Polygons

Source: Myoupo[17]

Comments: This algorithm uses BSR instructions which have already been discussed. See the citation for details.

Algorithm 2.2.50 Critical Support Lines of Two Convex Polygons

Source: Myoupo[17]

Comments: This algorithm uses BSR instructions which have already been discussed. See the citation for details.

Algorithm 2.2.51 Maximal Distance Between Two Convex Polygons

Source: Myoupo[17]

Comments: This algorithm uses BSR instructions which have already been discussed. See the citation for details.

Algorithm 2.2.52 Diameter of a Polygon

Source: Myoupo[17]

Comments: This algorithm uses BSR instructions which have already been discussed. See the citation for details.

Algorithm 2.2.53 Width of a Polygon

Source: Myoupo[17]

Comments: This algorithm uses BSR instructions which have already been discussed. See the citation for details.

Algorithm 2.2.54 Detection of Repetitions

Source: Delacourt[10]

Comments: This algorithm uses BSR instructions which have already been discussed. See the citation for details.
Algorithm 2.2.55 $k$-LCS Problem

*Source*: Semé[20]

*Comments*: This is a $k > 1$ problem and cannot be done on Optimal BSR. See the citation for more information.

Algorithm 2.2.56 Geometrical Problems

*Source*: Semé[22]

*Comments*: This is a $k > 1$ problem and cannot be done on Optimal BSR. See the citation for more information.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( \Sigma )</th>
<th>Datum</th>
<th>Tag</th>
<th>( \sigma )</th>
<th>Limit</th>
<th>( n ) vs. ( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix Sums 3.2.1</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( \leq )</td>
<td>( a )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Element Uniqueness 3.2.2</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( = )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Count Sort 3.2.3</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( &lt; )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Maximal Sum Subsequence 3.2.5-1</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( \leq )</td>
<td>( a )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Maximal Sum Subsequence 3.2.5-2</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( \geq )</td>
<td>( a )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Maximal Sum Subsequence 3.2.5-3</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( = )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Maximal Sum Subsequence 3.2.5-4</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( &gt; )</td>
<td>( 0 )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Maximal Vectors 3.2.6</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( \geq )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Convex Hull 3.2.7</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( = )</td>
<td>( a )</td>
<td>( n = m^2 )</td>
<td></td>
</tr>
<tr>
<td>Maximal Sum Subrectangle 3.2.8-1</td>
<td>( x[p[i], q[i]] )</td>
<td>( p[i] )</td>
<td>( p + q[i] )</td>
<td>( \geq r + qt )</td>
<td>( n = pq )</td>
<td>( m = (p+1)q^2 )</td>
</tr>
<tr>
<td>Maximal Sum Subrectangle 3.2.8-2</td>
<td>( x[p[i], q[i]] )</td>
<td>( p[i] )</td>
<td>( p + q[i] )</td>
<td>( &gt; s + qt )</td>
<td>( n = pq )</td>
<td>( m = (p+1)q^2 )</td>
</tr>
<tr>
<td>External Watchman Routes 3.2.9-1</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( = )</td>
<td>( j )</td>
<td>( \text{no } m )</td>
<td></td>
</tr>
<tr>
<td>External Watchman Routes 3.2.9-2</td>
<td>( y[i] )</td>
<td>( y[i] )</td>
<td>( \geq x[a] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>External Watchman Routes 3.2.9-3a</td>
<td>( x[i] )</td>
<td>( x[i] )</td>
<td>( = y[a] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>External Watchman Routes 3.2.9-3b</td>
<td>( x[i] )</td>
<td>( x[i] )</td>
<td>( = y[a] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \varepsilon )-Closeness 3.2.12-1</td>
<td>( i )</td>
<td>( i )</td>
<td>( = i + 1 )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \varepsilon )-Closeness 3.2.12-2</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( &gt; 0 )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Histogram 3.2.24</td>
<td>( x[i] )</td>
<td>( = x[a] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance Transform 3.2.26-1</td>
<td>( x[r, s] )</td>
<td>( r_i )</td>
<td>( \leq r_s )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance Transform 3.2.26-2</td>
<td>( x[r, s] )</td>
<td>( r_i )</td>
<td>( \geq r_s )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance Transform 3.2.26-3</td>
<td>( x[r, s] )</td>
<td>( s_i )</td>
<td>( \leq s_s )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance Transform 3.2.26-4</td>
<td>( x[r, s] )</td>
<td>( s_i )</td>
<td>( \geq s_s )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parenthesis Matching 3.2.30-1</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( = a )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parenthesis Matching 3.2.30-2</td>
<td>( x[i] )</td>
<td>( x[i] )</td>
<td>( &lt; x[a] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parenthesis Matching 3.2.30-3</td>
<td>( i )</td>
<td>( x[i] )</td>
<td>( = y[a] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parenthesis Matching 3.2.30-4</td>
<td>( i )</td>
<td>( x[i] )</td>
<td>( = a )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decoding Binary Trees 3.2.31-1</td>
<td>( i )</td>
<td>( i )</td>
<td>( = y[i] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decoding Binary Trees 3.2.31-2</td>
<td>( i )</td>
<td>( i )</td>
<td>( = y[i] + 1 )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generating Binary Trees 3.2.32-1</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( &gt; 0 )</td>
<td>( n \leq m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generating Binary Trees 3.2.32-2</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( = y[i] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reconstruction of a Binary</td>
<td>( i )</td>
<td>( x[i] )</td>
<td>( = x[a] + 1 )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree From Its Traversals 3.2.33-1</td>
<td>( x[i] )</td>
<td>( y[i] )</td>
<td>( = a )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree From Its Traversals 3.2.33-2</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( &lt; a )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree From Its Traversals 3.2.33-3</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( = y[a] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reconstruction of a Binary</td>
<td>( i )</td>
<td>( x[i] )</td>
<td>( = y[i] )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree From Its Traversals 3.2.33-4</td>
<td>( x[i] )</td>
<td>( y[i] )</td>
<td>( = a )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree From Its Traversals 3.2.33-5</td>
<td>( x[i] )</td>
<td>( y[i] )</td>
<td>( = a )</td>
<td>( n = m )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Summary of Characteristic BSR Variable Contents for Published Algorithms.
CHAPTER 3 – METHODOLOGY

The research for this thesis was prepared and planned by the High Performance Computing (HPC) Group in the Department of Computer Science at Southern Illinois University at Carbondale, led by Dr. Chih-Fang Wang. When the author joined the group, a program written by Rakavipat[19] existed which was a serial PRAM simulator written in the C++ Programming Language for the Solaris Operating System. The group extended this PRAM Simulator to a Solaris distributed programming environment.

A five-unit Linux computing cluster named Cluster B was acquired for the HPC Group, and the distributed PRAM simulator was converted from the Solaris Operating System to the Linux Operating System so that it could be used on Cluster B. The HPC Group met approximately every two weeks for over a year while the distributed PRAM simulator was further developed on Cluster B. See Luo[14].

The author independently began to extend the distributed PRAM simulator to include Optimal BSR. Other programming languages were investigated to develop a graphical user interface to monitor the calculations occurring in the Optimal BSR model. A display model, only, was developed in OpenGL, but this 3D programming environment was not ideal for the project, which was being displayed in 2D. The OpenGL code was converted to Java, which worked well for the research being accomplished. Java not only produced an effective user interface, but also was effective in implementing a working real time model of Optimal BSR.

During the graphical modeling of Optimal BSR in OpenGL, it became apparent that Optimal BSR could be pipelined. The Java implementation was accomplished from the beginning with the intent of creating the Pipelined Optimal BSR application. Once a working application of Pipelined Optimal BSR existed in Java, published algorithms were run on it to demonstrate that the models were consistent. Then, fundamental PRAM algorithms were run on it to see if they were compatible. Finally, other algorithms in the literature were examined to see if they were suitable for the model. The Cluster B application of Pipelined Optimal BSR is the only working application known to exist anywhere of any model of BSR.

The following sections in this chapter give a limited description of the PRAM simulator, as a precursor to the pipelined model, and then give a detailed account of Pipelined Optimal BSR. A later
section in this chapter analyzes BSR algorithms as a group.

3.1 The Distributed PRAM Simulator

An intricate knowledge of the distributed PRAM simulator is not necessary in order to understand Pipelined Optimal BSR. A general outline of the distributed PRAM simulator is given in this section because it was the precursor to Pipelined Optimal BSR and also because future research might benefit by combining the two systems into one.

The distributed PRAM simulator consists of three general components: (1) A Server which contains the PRAM; (2) a Booster which starts the clients; and, (3) the clients, which simulate processors that are accessing the PRAM. All of the components are written in C++. The Booster also uses the Expect Scripting Language. The PRAM simulator typically runs on a five-unit cluster, Cluster B, but it can run on one or more units, alone, in a cluster, or over the Internet. Each of the components runs one or more processes which can be on the same or different hosts.

A PRAM simulation runs in these steps:

1. The user starts the Server, which reads a file containing the code to be run. The Server waits to be contacted by the Booster.

2. The user starts the Booster, on the same or a different host.

(a) The Booster contacts the Server to determine how many clients are needed as virtual processors to run the instant code.

(b) The Booster prompts the user for a username and password for the hosts which will be used to run the clients.

(c) The Booster starts an Expect script and passes this information over to it.

(d) The Booster program exits.

(e) The Booster script opens a file to see what hosts are available to run the clients.

(f) The Booster script accesses each host, in order, and starts a client on that host. If the end of the host list is reached, the script begins, again, at the top of the list of hosts.
(g) The Booster script exits.

3. Each client contacts the Server and obtains the code which is to be executed.

4. The Server accounts for all of the clients contacting it and receiving the code.

5. In a lockstepped loop, each client sends a memory request, if any, to the Server. Appropriate log entries are made by each client.

   (a) The Server accounts for each client making contact.

   (b) The Server processes the memory requests, if any, in a C++ PRAM object. If error-checking is turned on, the PRAM object sorts the requests and determines if there are any ER or EW errors.

   (c) Appropriate log entries are made by the Server.

   (d) The Server notifies the clients of the results of the memory access requests, if any.

6. The above loop repeats, with all of the clients in lockstep, until the code is complete.

7. The clients exit.

8. The Server exits.

The Cluster B web site contains the source code and extensive information about the detailed operation of the simulator. The Cluster B web site is at http://c-b0.cs.sin.edu. See, also, Luo[14].

### 3.2 Pipelined Optimal BSR Issues

Pipelined Optimal BSR came about from research into making a software implementation of Optimal BSR in order to test algorithms and determine the feasibility of making a hardware implementation. This section gives a detailed top-down account of the issues involved in converting Optimal BSR into Pipelined Optimal BSR.

#### Issue 3.2.1 Storage of Operations, Comparators, and Limits


Memory Location in the BSR literature appears to refer to actual memory addresses. Akl[4] states that each memory location stores the data, the operation, the comparator, and the limit. To apply this, the usable memory would essentially be cut to 1/A of its original size, in order to store this additional information, or else the system would require a specially designed memory unit. However, none of the published algorithms require the storage of the operations, comparators, and limits except for the ones in instant use. Also, Algorithm 2.2.9 has a BSR instruction which does not even access memory.

As implied in the implementation of Optimal BSR, see Figure 1.12, the limit is contained in a memory record, and the operation and comparator are sent via control lines to BSR. None of these have to actually reach a memory location. Thus, full memory can be used, and no special memory units are necessary.

Pipelined Optimal BSR only uses memory, if applicable, to store data. Pipelined Optimal BSR also puts the input control information, the operation and the comparator, into a record.

**Issue 3.2.2 Disconnecting BSR from the PRAM**

The PRAM is not necessary in describing BSR, and, as an imaginary machine, the PRAM is not an appropriate component of a BSR application.

Pipelined Optimal BSR can stand alone and does not have to be a part of a PRAM. As Figure 1.1 shows, BSR is what is inside the box. The inputs and outputs are generally perceived to be from processors and to memory locations, but they do not have to be. For example, Algorithm 2.2.9 has a BSR instruction which is a concurrent read from processors to processors. This leads to four possibilities: (1) From processors to memory locations; (2) from memory locations to processors; (3) from processors to processors; and, (4) from memory locations to memory locations.

Furthermore, everything starts from the same place. In the examples of the published algorithms, as shown in Section 2.2, the pseudocode shows that this starting place is the processors. The instant processor in the pseudocode sets the corresponding limit, tag, data, operation, and comparator. As Algorithm 2.2.9 shows, memory records are not necessarily even required. Lindon’s[13] figures of Optimal BSR leave out the presumed association with a PRAM, presumably because a PRAM is not necessary in illustrating Optimal BSR.
Therefore, as a generalized case, there are source input records and target input records. Each one may represent processors or memory locations. They could also represent other things, such as data arriving from or going to other sources.

![Diagram](image)

**Figure 3.1:** The First Transitional Top-View of Optimal → Pipelined Optimal BSR.

Figure 3.1 shows a step in the transition from Optimal to Pipelined Optimal BSR. The input and output records are no longer referred to as processor or memory records. What were processor records are now source input records, and what were memory records are now target input records. What used to be memory output records are now target output records. This more accurately reflects that the source input records can come from processors, memory, or someplace else. Likewise, the target input records can come from processors, memory, or someplace else. The target output records can go to processors, memory, or someplace else.

**Issue 3.2.3 Inconsistent Record Lengths**

*In Optimal BSR, processor records carry an extra field, rank, from Box A to Box E. Thus, Box D sorts records of the type \((i, t, r, d)\) with records of the type \((a, l, v)\). Likewise, Box E has to interchangeably handle both types of records.*

Pipelined Optimal BSR adds a rank field to Optimal BSR’s memory records, now called target input records. This gives them the form \((a, l, r, v)\). This additional rank field will help solve additional issues. Also, the changes to the records are not complete, yet.
Issue 3.2.4 Processor and Memory Records are Really the Same Records

Lindon[13] added new processor and memory records and adjusted Optimal BSR’s write implementation in order to do reads. This is not necessary. Once processor and memory records become the same, reads can be done with no additional adjustments to BSR.

With the rank field added to the memory record, except for the labels, it becomes identical to the processor records. Let the $i$ and $a$ fields be integers which indicate the indexes of the processors and memory locations, respectively. Let the tags and limits be numerical float types. Let the ranks be integers. Let the $d$ and $v$ fields also be numerical float types. Now, both $(i, t, r, d)$ and $(a, l, r, v)$ have the same record type (integer, float, integer, float).

Since the source and target records can represent processors, memory locations, or other things, they are not accurately represented by two different records, $(i, t, r, d)$ and $(a, l, r, v)$, which are, in fact, interchangeable.

Pipelined Optimal BSR uses one record, $(i, t, r, d)$ to represent both sources and targets, both processor records and memory records. The field names $i$, $t$, $r$, and $d$ are arbitrary, and could just as well have been named $a$, $l$, $r$, and $v$ or other names. The references of the names to these fields in pseudocode and mathematical notation does not change.

Issue 3.2.5 Differentiating Source and Target Records

Optimal and Pipelined Optimal BSR need a way to differentiate between processor and memory records, and source and target records, respectively.

In Optimal BSR, boxes D and E inherently know what kind a record is, even though it is not explicitly indicated. This issue is recognized in Optimal BSR in Box F, where each of the $a$ fields is temporarily incremented by $n$ in order to sort the memory records from the processor records.

In Pipelined Optimal BSR, source records are indicated by positive values in the rank field, while target records are indicated by negative values in the rank field. In Optimal BSR, this would be comparable to the processor records having a positive value in this field while memory records have a negative value.

The rank field in target records will also have other uses in other issues.
Issue 3.2.6 Determining Where to Send the Output

Optimal BSR sends the read output to the processors via the upper right part of Box F, heretofore indicated, for write instructions, as being null output. However, in doing so, Optimal BSR uses different records than for write instructions, and some of the boxes perform slightly different functions than they do for write instructions. Even so, Optimal BSR reads are conventional reads that do not perform any reduction operations. In short, when Optimal BSR does reads, it increases the complexity and has less power than it does with writes.

Pipelined Optimal BSR uses the same records for reads as it does for writes. The boxes perform the same functions for reads as they do for writes. Furthermore, Pipelined Optimal BSR potentially performs reduction operations on the data while it is being read. In short, Pipelined Optimal BSR gets the same power with reads as it does for writes without any increase in complexity. The output for Pipelined Optimal BSR reads goes to the lower right of Box F just the same as for writes. A way needs to be devised as to whether to send this output to the processors or to the memory.

Pipelined Optimal BSR already has a field available to it, the rank field, to indicate where the output data goes. A negative value in the rank field indicates that instant record is target output. By making this value $-1$, it indicates that the output is for a processor. By making this value $-2$, it indicates that this output is for memory. If other output devices are needed, additional negative values can be used for them, as well.

Furthermore, the rank field will have a use in another issue.

Issue 3.2.7 The Identity does Not Work

Optimal BSR assumes that every operator has an identity. However, some operators do not have identities. For example, random, priority, and arbitrary.

This issue is resolved in the discussion of the next issue.

Issue 3.2.8 Indicating Valid Output

Optimal BSR writes to every memory record, even if the data is just the identity of the reduction operation, whether data was intended for the corresponding memory location or not. This can result in
unintended or inconvenient writes. Furthermore, it can result in ambiguous results: If the result was coincidentally the identity, there is nothing to indicate this.

For example, if the maximum datum from all of the processors is being written to memory location 0, all of the other instant memory locations could have negative infinity written to them. This could be an undesirable result.

Pipelined Optimal BSR subtracts 2 from the ranks of target output records that do not contain valid data. Target output records now have the following four possibilities:

1. \( r = -1 \): The output is for a processor and is valid output.
2. \( r = -2 \): The output is for a memory location and is valid output.
3. \( r = -3 \): The output was potentially for a processor, but is not valid output.
4. \( r = -4 \): The output was potentially for a memory location, but is not valid output.

If more output devices are needed, the rank values could be adjusted to indicate this. In Box F, where the rank field is not needed, the same thing could be accomplished by replacing the rank field with two bits.

![Diagram](image)

*Figure 3.2: The Second Transitional Top-View of Optimal \(\sim\) Pipelined Optimal BSR.*

Figure 3.2 shows another step in the top-view transition from Optimal BSR to Pipelined Optimal BSR. All of the records are now of the form \((i, t, r, d)\), representing processor records, memory records,
and potentially representing records from other input and output devices, with the rank field being used to indicate ranks of source inputs, as well as the types of target inputs and whether or not output data is valid. The top-view transition from Optimal BSR to Pipelined Optimal BSR is not yet complete.

**Issue 3.2.9 Control in Box B is Bidirectional**

In Optimal BSR, control information flows to a node, to a table, back to the same node, and then bidirectionally to the other nodes in the box. This bidirectional flow of control information is more complex than necessary and is not conducive to pipelining.

![Control in Box B is Bidirectional](image)

Figure 3.3: Control in Box B is Bidirectional

Figure 3.3 illustrates the bidirectional flow of control information in Box B in 10 steps. Part (a) shows (1) the information goes to the lower right node, (2) to the table, (3) back to the same node, and (4-6) spreads out to all of the rows on the left. Part (b) shows, starting with (6), how the information then proceeds to all of the nodes and the switches on the right.

Figure 3.4 shows that Pipelined Optimal BSR takes the table out of Box B and directs control information to the first column of nodes, where it is distributed in one direction only to the other nodes and switches on the right. The table and control lines will have other uses in other issues.

**Issue 3.2.10 Table 3’s Location is Vague**

Both Box B and Box E access control information from Table 3, but Optimal BSR does not specify where this table is at or specifically how the control information is distributed.
Pipelined Optimal BSR combines Table 3 with the table removed from Box B and extends the control line to Box E, as represented by the dotted line in Figure 3.4.

Figure 3.5 shows the final outcome of the top-view transition from Optimal BSR to Pipelined Optimal BSR. All of the records are of the same type, and all control information enters from the upper left where it will be unidirectionally distributed to both boxes which need it.

**Issue 3.2.11 Uneven Stages**
Records cannot flow uniformly through Optimal BSR because some of the the stages are uneven. Figure 1.8 outlines, in a dashed line, an even stage, Stage 2, of Batcher’s Sort. It is even because it has the same number of nodes, four, as the first stage in the box. The next stage, Stage 3, in the same figure, is uneven because it has a fewer number of nodes, two, than the first stage in the box. Records will not all arrive at Stage $j$ at the same expected time. On a paper implementation, such as Optimal BSR, this situation is alright because human readers assume that the data automatically arrives at each stage when it is supposed to arrive. However, in an application, such as Pipelined Optimal BSR, the uneven stages have to be accounted for so that the records arrive when they are supposed to arrive.

Figure 3.6: Uneven Stages

Figure 3.6 shows how a literal interpretation of Box A splits up the records on the third step between two different stages. Each record goes to the next node during a step, whether the next node is in the next stage or in a stage further down the box.

Figure 3.7: Even Stages
Pipelined Optimal BSR adds registers as placeholders in uneven stages as Figure 3.7 shows. A register does not change a record, it just holds the record temporarily until the record is supposed to proceed to the next stage. Figure 3.7 points out a single register in the lower right corner of the box. This figure also outlines in a dotted line the third stage, which now includes two nodes and four registers.

**Issue 3.2.12 Placement of Box C**

*Box C conceptually belongs underneath Box A rather than underneath Box B because boxes a and c are both sorting input records, whereas Box B is doing calculations.*

![Figure 3.8: Moving Box C](image)

Figure 3.8 shows Box C moved underneath Box A for cosmetic reasons. It was also shown this way in Akl[6].

**Issue 3.2.13 Additional Registers**

*Records cannot flow evenly underneath Box B and control records do not flow evenly from the beginning to Box E. If \( n \neq m \), the number of stages in boxes A and C will not be the same and the records will not exit these boxes at the same time.*

Pipelined Optimal BSR adds additional registers as needed.
Figure 3.9: Pipelining

Figure 3.9 shows that with registers added to the control line, a single stage can contain all of the source and target records as well as the corresponding control information. This can be done throughout all of the stages in Pipelined Optimal BSR so that each stage can contain a completely separate BSR instruction. Pipelining has been achieved.
3.3 Pipelined Optimal BSR

The Pipelined Optimal BSR application has taken the form of a Java applet named *BSR in Java* located at http://eb0/es/siu.edu/clusterb/java/bsrjava.html.

The source code for *BSR in Java* precisely defines the internal operation of Pipelined Optimal BSR. This section describes *BSR in Java*, which simultaneously gives a detailed blueprint for Pipelined Optimal BSR.

First, a general overview is given of *BSR in Java*. Then, a detailed account is given, component by component, from start to finish.

3.3.1 An Overview of *BSR in Java*

The following page shows the opening screenshot for *BSR in Java*. 
Figure 3.10: The Opening Screenshot of BSR in Java

Figure 3.10 shows the opening screenshot of BSR in Java. The primary focus of the applet is a
detailed layout of Pipelined Optimal BSR with \( n = m = 8 \). This home view is shown at the start of the applet. The overhead buttons can zoom in and zoom out, and the scroll bars can move the layout about. The Home button returns the layout to its original size and centers the display.

The three other buttons in Figure 3.10 operate the BSR. The Load Inputs button brings up a dialog box to load the inputs for a BSR instruction. The two increment buttons execute stages either one stage at a time or five stages at a time. Once the inputs are loaded, the BSR does nothing unless an increment button is selected.

![Problem: Lindon Table I, Step 1](image)

**Figure 3.11: The Application Input Dialog Box**

Figure 3.11 shows the input dialog box. Java Choice menus facilitate entering the information for a BSR instruction. The Problem Choice Menu allows the selection of an instruction from a list of given problems. In the figure, a prefix sums BSR instruction from a problem in Lindon[13] has been chosen. The user also has the option of entering a new BSR instruction, or of making changes to the selected BSR instruction.

The Operation Choice Menu allows the user to select from Sum, Product, Maximum, Minimum,
and others, however, only the four named operations have been tested. The comparator Choice Menu allows the selection of $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$. These two choices make up the control record.

The unnamed Choice menu with the selection *Not Flipped* represents whether or not the source and target inputs are flipped. That is, whether this instruction is a write or read. A typical BSR write instruction is not flipped. A BSR read instruction would be flipped, because it conceptually flips the source and target inputs. In reality, the only thing that is changed in the input dialog box is that all of the -1 Ranks are changed to -2. This will indicate to the Box F switches that the output is intended for processors rather than for memory. As indicated earlier, this can be adapted to other input and output devices besides processors and memory.

On the left of Figure 3.11 are eight $(i, t, r, d)$ records representing the source inputs. The Index, Tag, and Data fields in this area of the input dialog box can individually be changed by the user. The Rank field has been preset to $n + 1$ to represent that this is a source record which has not yet received its rank.

On the right of Figure 3.11 are eight $(i, t, r, d)$ records representing the target inputs. The Index, Limit, and Data fields here can individually be changed by the user. The Rank fields are set to either -1 or -2 depending upon whether the input is Flipped or Not Flipped, as explained two paragraphs above.

If the source and target fields are conceptually grouped in columns, instead of rows, they can be considered to be arrays or vectors, six of which the user can set.

The bottom two buttons either save the inputs and close the dialog box or else cancel the inputs.

It becomes apparent after using this input dialog box that what is essentially happening is not just an interaction between processors and memory. What is essentially happening is that six arrays are set up to interact with each other in specified ways to produce certain results.
Figure 3.12: The Darkened Switches Hold the New Input Records

The seventeen darkened switches in zoomed-in Figure 3.12 hold the input records which were just provided by the Input Dialog Box. The top one holds the control record which is read to go into the Table Node. The others are source and target records of the form \((i, t, r, d)\) ready to enter boxes A and C.
Figure 3.13 shows a popup box describing the contents of Box A, Switch 0. Each component has an identifying number which is displayed when zoomed-in sufficiently. The user can click on any of the component’s to get a popup box describing the contents of that component. The title of the popup box reminds the user that the record is in the \((i, t, r, d)\) form of index, tag/limit, rank, data. The first line of the popup box identifies the component which is being described. In this case, Switch 0 of Box A. The next line describes what this component is doing. Holding data for input. The next two lines show the data in the input line and the output line. In this case, it is the same data, Index 1, Tag 1.0, Rank 9, and Data 1.0, or \((1, 1.0, 9, 1.0)\). This is the same data from the upper left record in the Input Dialog Box in Figure 3.11. A component does not necessarily have to keep the data from the input lines. However, BSR in Java does this in order to provide all of the information about a component in one place. The convenience of this is more obvious in the nodes with complicated behaviors. The information that a popup box provides depends upon the function of the component being examined.

Clicking on the darkened control switch would provide a popup box showing that this component is holding the control record, which contains the operation and comparator for this BSR instruction.

After a subsection on an overview of the source code, each component of each box will be described
from the start to the finish of Pipelined Optimal BSR.

### 3.3.1.1 An Overview of the Source Code

The .java source code files of *BSR in Java* total over 9,000 lines. The code is principally in 10 primary classes in 10 .java files.

![Class Dependencies Diagram](image)

**Figure 3.14: The Class Dependencies in BSR in Java**

Figure 3.14 shows the class dependencies in *BSR in Java*. *Swich* is intentionally misspelled to differentiate it from the *switch* statement in Java. Here is a summary of the classes:

- **BSR.java**: Contains main(). Instantiates and initializes components and variables. Paints the screen. Executes a stage. Passes records between boxes, as appropriate.

- **Table.java**: Manages control records. Advances stages in the control lines.

- **Box.java**: Instantiates and initializes subcomponents and variables. Executes a stage within a box.

- **Gbox.java**: Determines the graphical layout of the components of a box.
• **Node.java**: Instantiates and initializes subcomponents and variables. Stores where a node's inputs come from. Executes a node. Resets a node.

• **Switch.java**: Instantiates and initializes subcomponents and variables. Stores where a switch's inputs come from. Executes a switch. Resets a switch.

• **BSRdata.java**: Contains an \((i, t, r, d)\) record.

• **BSRdata.java**: Contains information concerning the leaders in Box E.

• **BSRconstants.java**: Contains global constants.

• **BSRcontrols.java**: Contains information controlling the graphical display.

Many segments of *BSR in java* source code will be shown as figures to describe how Pipelined Optimal BSR works. Each segment of source code will be explained after the box in which it appears.

```java
public class BSRdata{
    public int i;   // the index
    public float t; // the tag or the limit
    public int r;   // the rank
    public float d; // the data
} // BSRdata
```

Figure 3.15: The Source Code Segment for \((i, t, r, d)\)

Figure 3.15 shows how the fundamental record \((i, t, r, d)\) is declared in BSRdata.java. The variable names \(i, t, r,\) and \(d\) are exact matches to the variable names in the record.
public final class BSRconstants {
    public static final int BOX_A = 0;
    public static final int BOX_B = 1;
    public static final int BOX_C = 2;
    public static final int BOX_D = 3;
    public static final int BOX_E = 4;
    public static final int BOX_F = 5;
    public static final int TABLE = 6;
    public static final int RANK_INFINITY = 9;
} // BSRconstants

Figure 3.16: The Source Code Segment for Pertinent Global Constants

Figure 3.16 shows how some of the global constants are declared in BSRconstants.java. These constants merely make the code more readable.

```java
for(int lcv = BOX_F; lcv >= BOX_A; lcv--){
    box[lcv].increment_stage();
} // for
table.increment_stage();
```

Figure 3.17: The Box Source Code Segment to Increment a Stage

Figure 3.17 shows how BSR.java increments a stage by calling on each box, in reverse order. Then, it calls on the table to increment a stage. BSR.java does some other housekeeping, not shown here, which assists in getting records from one box to the next one.

### 3.3.2 Box A

Box A is based on Batcher’s Sort from Section 1.2.2.1.1, and it lexicographically sorts on the tag and data fields.

In Figure 3.18 a popup box shows an example of what can happen in Box A, Node 9. Two records have been input, (3, 3.0, 9, 6.0) and (2, 2.0, 9, 2.0). They have been sorted on their tag and data fields and output in sorted order. The other nodes in Box A likewise sort on the tag and data fields. Nodes can typically have up to three inputs and three outputs, all of which are shown in the popup box whether used or not. The unused ones have data fields set to −1.
3.3.2.1 Box A Source Code

The Java source code for the nodes in Box A precisely defines what is happening in Box A. The code is SIMD: Single Instruction Multiple Data. Each node runs the same code on whatever data is passing through it. The nodes are executed in reverse order, by index, so that old records are moved out of a node before new ones move in. The end of the pipeline is emptied before the beginning of the pipeline is filled.
for(int lcv = number_of_nodes - 1; lcv >= 0; lcv--)
{
    node[lcv].active = false;
    ...
    // copy the node data for each of the three possible inputs
    for(int lcv2 = 2; lcv2 >= 0; lcv2--)
    {
        input_source1 = node[lcv].input_source1[1cv2];
        if(input_source1 != -1)
        {
            if(node[lcv].input_from_node)
            {
                if(node[input_source1].active)
                {
                    node[lcv].active = true;
                    // output from the previous node is copied to the input of this node
                    ...
                }
            }
        }
    }
    node[lcv].execute("", ");
    ...
} // for

Figure 3.19: The Source Code Segment for Copying Records in Box A

Figure 3.19 shows part of the code from Box.java which executes a stage inside of Box A. The code loops through each node in reverse order. It sets the instant node to active = false. The code loops through each possible input line of the instant node. It finds the source of that input. It determines if the source is valid (-1 indicates an invalid source). It determines if the source of the input is a node, a switch, or a previous box. It determines if the source contains active data. Then, if appropriate, if copies the records from the output line of the source to the input line of the instant node. Once this is done, the node is executed. The ellipses in the code indicate where code has been omitted. This could be because the code is not related to Box A, or else because the code is lengthy and performs a straightforward concept, such as copying records. For example, the code for copying from a switch has been omitted for brevity.

The source code segment for Box A nodes is in Appendix A.1. It shows how each node in Box A does the same code, which sorts on the tag and data fields, when execute() is called on that node. The nodes in the last stage of Box A also assign a rank to each record as it exits Box A.
3.3.3 The Table

The table translates a control record into information which is more useful to the boxes. Specifically, it designates a mode for Box B, an operator corresponding to the operation (addition corresponding to summation, for example), and a formula for the switches in Box E.

Lindon [13] designated five possible modes for Box B:

1. A prefix computation for all operations except minimize.

2. A suffix computation for all operations except maximize.

3. A prefix computation for minimize.

4. A suffix computation for maximize.

5. A two-way computation for maximize or minimize when the tags should not equal the limits.

The mode is determined by the Table and sent to Box B, where it is acted on appropriately. The formulas will appear in the code for Box E. Lindon [13] also had the Table determine the identity for the operator and sent along as well. However, BSR in Java just determines the identity directly from the operator, when appropriate. BSR in Java does not use identities to the same extent as Optimal BSR. See Issue 3.2.8.
In Figure 3.20 a popup box shows an example of what can happen in the Table while the records are in the first stage of Box A. In this figure, the operation is Sum and the comparator is \( \leq \). These have been converted to Mode 1, addition, and a formula of \textit{greatest equivalent or preceding}, which refers to leaders in Box E. This figure also shows that, when zoomed in sufficiently, the control registers, over the popup box, are indexed so that they can be easily referenced. \textit{gr\_equiv} refers to the \textit{Greatest} leader in Box E. Also, \textit{1st\_equiv}, when it appears, refers to the \textit{First} leader.

### 3.3.3.1 The Table Source Code

The source code segment for executing the table is in Appendix A.2. Untested logical operations have been left out. Other experimental operations, such as arbitrary and priority, have also been omitted. The code first determines if the input switch for the control record has an active record. If so, then it obtains the operation and comparator from the switch. The code determines the mode, the operator, and the formula for output.
3.3.4 Box B

Box B is based on The Prefix Box from Section 1.3.2.1, and it performs calculations based on the control records.

In Figure 3.21 a popup box shows an example of what can happen in Box B, Node 13. Three records have been input, (5, 5.0, 5, -2.0), (6, 6.0, 6, 3.0), and (7, 7.0, 7, -5.0). The popup box indicates that Mode 1 is being used with the + operator, which is a prefix sums computation. This calculation is performed and output on all three lines. Lindon [13] proposed a $T$ variable which associates a tag value with each row of nodes in Box B. This tag value is the tag of the first record entering a row of a box. In the example above, $T = 6$, indicating that the Tag for this row is 6. $T$ also happens to equal the tag for the node record entering on Input Line 1, which can be seen in the example. This $T$ value is useful in keeping track of which data falls less than, equal to, or greater than the tag of the instant record.

Although the value of $T$ remains the same through a row for given instruction, this value moves in a pipelined fashion through the box with the corresponding records, a stage at a time. The actual values
of $T$ present in a row may vary depending upon the possible preceding and following records for other instructions which may be pipelining through the box at the same time.

The popup box for Mode 5 is different and is explained below in a separate section which specifically covers Mode 5.

### 3.3.4.1 Box B Source Code

Box B, like all of the boxes, moves records through the nodes, starting with the last node. Duplicate code, from Box A, above, will not be shown here.

```java
// copy control data to Box B switch
if (whoami == BOX_B) {
    input_source1 = switch[0].input_source1[1];
    if (node[input_source1].active) {
        switch_mode = node_mode[input_source1].substring(0);
        switch_operator = node_operator[input_source1].substring(0);
    } // if(swich input is active)
} // if(BOX_B)

Figure 3.22: The Source Code Segment for Box B Control Data

Figure 3.22 shows specialized code for Box B which moves control data through the box. Some additional code for Mode 5 also goes here. However, Mode 5 will be covered separately below.

```switch``[1].execute(swich_mode, switch_operator, "");
```

Figure 3.23: The Source Code Segment to Execute Box B Switches

Figure 3.23 shows how Box B executes the switches, which are looping in reverse order. All that the switches do in modes 1 through 4 is to pass the records through without changing them. The nodes are now free to obtain their own new control data, if appropriate.
// copy control data to Box B nodes
if(lcv > 7 && whoami == BSRconstants.BOX_B) {
    input_source1 = node[lcv].input_source1[1];
    if(node[input_source1].active) {
        node_mode[lcv] = node_mode[input_source1].substring(0);
        node_operator[lcv] = node_operator[input_source1].substring(0);
    } // if(active)
} // if(lcv > 7 and BOX_B)
else if(lcv < 8 && whoami == BSRconstants.BOX_B) {
    input_source1 = node[lcv].input_source1[1];
    if(input_node[input_source1].active) {
        node_mode[lcv] = mode_holder.substring(0);
        node_operator[lcv] = operator_holder.substring(0);
    } // if(active)
} // if(lcv < 8 and BOX_B)

node[lcv].execute(node_mode[lcv], node_operator[lcv]);

Figure 3.24: The Source Code Segment for Box B Control Data in Nodes

Figure 3.24 shows how the higher nodes get control information from the lower nodes, and how the
lowest nodes get control information from outside the box. The BSR.java code has already placed, if
appropriate, the control data from the control line into holder variables which are accessible to the first
Box B nodes.

Figure 3.25: The Source Code Segment to Execute Box B Nodes

Figure 3.25 shows that the command to execute Box B nodes has additional control information as
parameters.

The switch source code for modes 1 through 4 merely passes the records through and is not interesting.
The code for Mode 5 will be explained in Section 3.3.4.1.1.
boolean prefix = false;
boolean suffix = false;
if(mode_input.compareTo("1") == 0)
    prefix = true;
else if(mode_input.compareTo("2") == 0)
    suffix = true;
else if(mode_input.compareTo("3") == 0)
    prefix = true;
else if(mode_input.compareTo("4") == 0)
    suffix = true;
else { // if(mode_input.compareTo("5") == 0) {
    prefix = true;
    suffix = true;
} // if(mode 5)

Figure 3.26: The Source Code Segment for Translating the Mode

The source code segment in Figure 3.26 translates from the mode to the type of computation, prefix or suffix.

int start = 1;
int finish = 1;
if(prefix)
    start = 0;
if(suffix)
    finish = 2;

Figure 3.27: The Source Code Segment for Box B Input Selection

The source code in Figure 3.27 determines the valid input lines for the Box B nodes.

The source code segment for executing Box B nodes is in Appendix A.3. The code loops through the valid inputs and performs the operations as appropriate. The calculations are performed on a temp_data record which is then copied to the output lines.

3.3.4.1.1 Mode 5 Mode 5 is more complex than all of the other modes in Box B put together and is not used in any of the published algorithms. Mode 5 is used for maximum or minimum operations in which the tag is not equal to the limit. Table 2.1 shows that these conditions have not been used in any of the published algorithms. Therefore, it is questionable if a hardware application of BSR would include Mode 5. For this reason, it is treated separately in this paper.
In Figure 3.28 a popup box shows an example of what can happen in Box B, Node 13, while in Mode 5. Three times as many records have been input, \((5, -1.0, 5, -\infty), (5, 5.0, 5, -2.0), (5, -1.0, 5, -\infty), (6, -1.0, 6, -\infty), (6, 6.0, 6, 3.0), (6, -1.0, 6, -\infty), (7, -1.0, 7, -\infty), (7, 7.0, 7, -5.0), \) and \((7, -1.0, 7, -\infty)\). The popup box indicates that Mode 5 is being used with the max operator. \(-\infty\) appears because it is the identity for max. Calculations are performed with different results for the three output lines. This time \(T\), see Figure 3.21, comes into real play. Output 0 is for records with tags less than \(T\); Output 1 is for records with tags equal to \(T\); and, Output 2 is for records with tags greater than \(T\).

With nine inputs instead of three, Mode 5 does not actually fit the diagrams given in figures 1.14 and 1.15.
Figure 3.29 shows how a Box B node can have nine inputs in Mode 5. Each Box B node in Mode 5 has three separate outputs: (1) For computations of records with tags less than $T$; (2) for computations of records with tags equal to $T$; and, (3) for computations of records with tags greater than $T$. In other Box B modes, three separate inputs are possible. However, in Mode 5, each of these three possibilities must include all three types of output. Therefore, a Mode 5 node can have up to nine inputs.

### Mode 5 Source Code
The source code for Mode 5 is greater in length and complexity because of the extra inputs and outputs. Three times as many records need to be copied to and from the switches and nodes. Part of the source code segment for Box B nodes in Mode 5 is in Appendix A.4. The code finds the maximum or minimum values that are less than, equal to, and greater than the limits. This will be used in the switches to determine the maximum and minimum values when the tags do not equal the limits. This code is twice the length of the comparable code for all of modes 1 through 4, yet Mode 5 is not used in any of the published algorithms. The code that just copies records is not shown.
if(box == BSRconstants.BOX_B & & mode.compareTo("5") == 0) {
    if(operator.compareTo("Max") == 0)
        if(data_input.a > data_inputc.d)
            data_output[1].d = data_input.a;
    else
        data_output[1].d = data_inputc.d;
    else if(operator.compareTo("Min") == 0)
        if(data_input.a < data_inputc.d)
            data_output[1].d = data_input.a;
    else
        data_output[1].d = data_inputc.d;
} // if(Box B & & Mode 5)

Figure 3.30: The Source Code Segment for Box B Switches in Mode 5

Figure 3.30 shows the source code in the switches in Mode 5 to select the greatest (or least) values for records in which tags are not equal to limits.

3.3.5 Box C

Box C, like Box A, is based on Batcher’s Sort from Section 1.2.2.1.1. However, it only sorts on the limits, saved in the t variables in the records.

3.3.5.1 Box C Source Code

The Box C source code is similar to the code for Box A, except that Box C only sorts on the limits. The Box C source code is omitted from this report.

3.3.6 Box D

Box D is based on the Merge Circuit from Section 1.2.2.2. The nodes sort on their inputs similar to the nodes in boxes A and C. However, the nodes in Box D sort on the tags, the sources, and the indexes. Sorting on the sources means that if the rank variable is positive, then this is a source record and it is output on a higher line. If the rank variable is negative, then this is a target record and it is output on a lower line.
3.3.6.1 Box D Source Code

The Box D source code is similar to the Box A source code which does the merging part of the sort. This same implementation is used for simplicity, not efficiency. The Box D source code is omitted from this report.

3.3.7 Box E

Box E is based on The Distribute Box from Section 1.3.2.2, and it determines the record leaders which are used to distribute data.
Figure 3.31: Box E, Node 25

The popup box of Box E Node 25 in Figure 3.31 shows that the Box E nodes are the busiest of all the nodes in Optimal BSR. Each of the three input lines not only carries a record, but also the ranks and data of all five of the leaders as determined, so far, by one of the previous nodes. The instant node considers each of the three sets of leaders and combines them into one new set, taking into account that
each of the the tags of the incoming data might be the same, higher, or lower than \( T \). The instant node sends the new set of leaders onward to all of it’s output lines. By the end of Box E, the data has been thoroughly distributed and the final leaders chosen. Two numbers are associated with each leader: (1) the first number, which is the rank of that leader, and (2) the second number, which is the datum associated with that rank.

Figure 3.32: Box E, Switch 12

Figure 3.32 shows the popup box for Box E, Switch 12. Each Box E switch has one input which contains a record and the set of leaders for the \( T \) of that row. The switches are given a formula, via the control line, which is used as described in Section 1.3.2.2 to determine the reduced value. In the above figure, the formula is \( gr\text{-equiv} \mid preceding \), which refers to the Greatest and Preceding leaders. The value for the Greatest leader, if valid, is placed in the \( d \) field of the output record. Otherwise the value for the Preceding leader is placed there. This is not the same as Lindon’s[13] formula, which did not consider the possibility that none of the tags would equal a limit.

The calculations are done and the next box will separate the source and target records.
3.3.7.1 Box E Source Code

Box E uses a class called BSRedata, where each instance of which contains one set of leader information. Each set keeps track of the five leaders as described in Section 1.3.2.2. The data for each leader consists of its rank, and the value associated with that rank.

```java
// the structure of data in Box E
public class BSRedata{
    public float first;
    public float greatest;
    public float preceding;
    public float following;
    public float overall;
    public int firstr;
    public int greatestr;
    public int precedingr;
    public int followingr;
    public int overallr;
    public BSRedata() {
        first = -1;
        greatest = -1;
        preceding = -1;
        following = -1;
        overall = -1;
        firstr = 9;
        greatestr = 0;
        precedingr = 0;
        followingr = 9;
        overallr = 0;
    } // BSRedata()
} // BSRedata
```

Figure 3.33: The Source Code Segment for Defining Data in Box E

BSRedata, declared as in Figure 3.33, is referred to informally as edata, for Box E data. Each node and switch in Box E, when active, has its own edata associated with it. In additional to passing records from node to node, and from node to switch, Box E also has to pass edata from node to node, and from node to switch. The code for doing so, as well as the code for passing along control information, is omitted from this report.

The switches are done first in Box E to make room for the nodes to advance. Part of the source code segment for Box E Switches is in Appendix A.5. This code is for after the nodes have already
determined the leaders. The ellipses represent code which has been omitted because it does not apply
to Box E. The code is divided according to the control formula used and it tests for invalid data. The
formulas are not necessarily the same as in Lindon[13], which assumed in some cases that data would be
valid.

A source code segment for Box E nodes is in Appendix A.6. This is the code which determines the
leaders. It does not include code which moves data around or which initializes the edata. The code is
the same for each node and is executed in reverse order by node index. \textit{lev}, for \textit{loop control variable},
contains the current node index. At the end, the resulting leader information is in edata[lev].

\textbf{3.3.8 Box F}

Box F, like Box A, is based on Batcher’s Sort from Section 1.2.2.1.1. However, it sorts
lexicographically on the source and, if a target record, on the index. The result is that all of the source
records go to the top of the Box E exit where they go to a null device. All of the target records go to
the bottom of the Box E exit where they are sorted on their indexes. The Box E switches just hold the
output for future use.

\textbf{3.3.8.1 Box F Source Code}

The Box F source code is like the code for Box A, except that Box F sorts on the source and, if a
target record, on the index.

\textbf{3.3.9 A Read}

Most references to BSR concern memory writes. However, memory reads are equally possible. As
stated in Issue 3.2.4, Pipelined Optimal BSR can do reads without any changes to its internal structure.
The Lindon[13] implementation sends memory records for reading to the lower set of inputs. They can
just as easily be directed to the upper set of inputs.
Figure 3.34: Input for a Memory Read

Figure 3.34 shows a possible input box for a memory read. *Flipped* has been selected which set all of the target ranks to $-2$. The memory data to be read is in the records on the left, from 14.0 down to 47.0. The processor records are on the right. For example, processors 2, 4, and 6 are all reading from memory location 4.0. The operation has been set to Sum and the Comparator set to $\neq$. However, these do not matter in this case because only one value is going to each destination. Multiple values could go to same destinations by settings tags equal to each other, in which case the operation and comparator would have more effect.

### 3.4 Analysis of the Algorithms

Some 16 published algorithms were run through successfully on the BSR in Java application of Pipelined Optimal BSR with the results matching existent published results. Basic PRAM algorithms, not in the published literature, were attempted on the application. The published Optimal BSR
algorithms were examined in conjunction with their possible use with *BSR in Java*.

### 3.4.1 Basic PRAM algorithms

Exclusive write, one-to-all broadcasting, concentrate, distribute, and combine algorithms all work fine in Optimal BSR. Generalize works in two steps. Concurrent reads with sum, product, maximum, and minimum work fine. However, the following algorithms have some issues.

#### 3.4.1.1 Random

Optimal BSR contains no mechanism to randomly select values.

#### 3.4.1.2 Priority

The priority algorithm based on processor indexes fails in Box E, because the indexes are not currently distributed with the data. This could be accomplished if desired. Priority algorithms based on other criteria are $k = 2$ problems.

#### 3.4.1.3 Arbitrary

Optimal BSR contains no mechanism to arbitrarily select values other than to use other existing types of selection. For example, if the maximum operation qualifies as one's definition or *arbitrary*, then Optimal BSR can do arbitrary algorithms, but it is actually consistently producing something else, such as maximum. If one's definition of *arbitrary* is a pseudo-random selection of values, potentially different each time, then Optimal BSR is not capable of doing arbitrary selections.

### 3.4.2 The Published Algorithms

The published algorithms were examined to determine what kinds of BSR instructions of the potential ones were actually being used. Table 2.1 was sorted as follows with duplicate types of instructions omitted:
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( R )</th>
<th>Datum</th>
<th>Tag</th>
<th>( \sigma )</th>
<th>Limit</th>
<th>( n ) vs. ( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parenthesis Matching 3.2.30-3</td>
<td>( \cap )</td>
<td>( x[i] )</td>
<td>( x[i] )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Maximal Sum Subsequence 3.2.5-3</td>
<td>( \cap )</td>
<td>( y[i] )</td>
<td>( y[i] )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>All Nearest Smaller Values 3.2.34-2</td>
<td>( \cap )</td>
<td>( r[i] )</td>
<td>( r[i] )</td>
<td>( r_a )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Generating Binary Trees 3.2.32-2</td>
<td>( \cap )</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Maximal Sum Subsequence 3.2.5-4</td>
<td>( \cap )</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( 0 )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Maximal Sum Subsequence 3.2.5-5</td>
<td>( \cap )</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( a )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Generating Binary Trees 3.2.32-1</td>
<td>( \cap )</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( \gamma )</td>
<td>( n \leq m )</td>
<td></td>
</tr>
<tr>
<td>Reconstruction of a Binary</td>
<td>( \cup )</td>
<td>( x[i] )</td>
<td>( x[i] )</td>
<td>( x[a] + 1 )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Tree From Its Traversals 3.2.33-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decoding Binary Trees 3.2.31-1</td>
<td>( \cup )</td>
<td>( i )</td>
<td>( x[i] )</td>
<td>( y[i] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>All Smallest Values 3.2.34-3</td>
<td>( \cup )</td>
<td>( i )</td>
<td>( 2^i )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Reconstruction of a Binary</td>
<td>( \cup )</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( y[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Tree From Its Traversals 3.2.33-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decoding Binary Trees 3.2.31-2</td>
<td>( \cup )</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( y[i] + 1 )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>( \epsilon )-Closeness 3.2.12-2</td>
<td>( \cup )</td>
<td>( x[i] )</td>
<td>( y[i] )</td>
<td>( a )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Tree From Its Traversals 3.2.33-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance Transform 3.2.26-1</td>
<td>( \cup )</td>
<td>( r[s] )</td>
<td>( r_s )</td>
<td></td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Distance Transform 3.2.26-3</td>
<td>( \cup )</td>
<td>( r[s] )</td>
<td>( s_s )</td>
<td></td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>External Watchman Routes 3.2.9-2</td>
<td>( \cup )</td>
<td>( y[i] )</td>
<td>( y[i] )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Element Uniqueness 3.2.2</td>
<td>( \Sigma )</td>
<td>1</td>
<td>( x[i] )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>Count Sort 3.2.3</td>
<td>( \Sigma )</td>
<td>1</td>
<td>( x[i] )</td>
<td>( x[a] )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>( \epsilon )-Closeness 3.2.12-2</td>
<td>( \Sigma )</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( i + 1 )</td>
<td>( n = m )</td>
<td></td>
</tr>
<tr>
<td>External Watchman Routes 3.2.9-1</td>
<td>( \Sigma )</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( j )</td>
<td>no ( m )</td>
<td></td>
</tr>
<tr>
<td>Prefix Sums 3.2.1</td>
<td>( \Sigma )</td>
<td>( x[i] )</td>
<td>( i )</td>
<td>( a )</td>
<td>( n = m )</td>
<td></td>
</tr>
</tbody>
</table>
| Maximal Sum Subrectangle 3.2.8-2             | \( \Sigma \) | \( x[p[i], q[i]] \) | \( p[i]p + q[i] \) | \( s + qt \) | \( n = pq \)
| Maximal Sum Subrectangle 3.2.8-1             | \( \Sigma \) | \( x[p[i], q[i]] \) | \( p[i]p + q[i] \) | \( r + qt \) | \( n = pq \)
| Reconstruction of a Binary                  | \( \forall \) | \( x[i] \)  | \( y[i] \)  | \( a \)      | \( n = m \)  |
| Tree From Its Traversals 3.2.33-2             |          |            |            |             |       |
| All Nearest Smaller Values 3.2.34-1          | \( \forall \) | \( 2^i \)   | \( x[i] \)  | \( x[a] \)  | \( n = m \)  |

Table 3.1: Sorted BSR Instructions for Published Algorithms.

What is most notable from Table 3.1 is what is missing: There are no problems using the product, logical AND, or logical OR operations. Also, not equals is not used, even though Mode 5, which handles
not equals, is more complex than modes 1 through 4 all together.¹

Also notable from Table 3.1 is the wide variety of possible types of values for the datums, tags, and limits, implying flexibility, but also suggesting that it takes a certain amount of cleverness to determine what belongs where.

3.5 Pipelined Algorithms

All of the Optimal BSR algorithms potentially benefit from pipelining. After one instruction begins, another one, if it is ready, can start immediately without waiting for the first one to finish. However, the published algorithms were examined to determine if any of them particularly benefited from pipelining. Two of them stood out, Histogram and External Watchman Routes, and they are discussed further in the following sections.

3.5.1 Histogram

Histogram from Algorithm 2.2.24, see Melter[15], is similar to Bucket Sort: each datum is examined and conceptually placed in a bucket which has been designated to collect data of that corresponding value. In the end, the buckets are lined up by their values and their corresponding contents are thus sorted. By counting the number of datums in each bucket, one obtains a histogram.

¹Serné[20] uses not equals, but it is in conjunction with a $k > 1$ algorithm, which Optimal BSR cannot do.
Figure 3.35: An Optimal BSR Bucket Sort

Figure 3.35 illustrates the Bucket Sort on Pipelined Optimal BSR. Figures of buckets have been placed at the output to signify that the sorted outputs are going into the appropriate buckets. See Algorithm 2.2.24. Bucket Sort is suitable for pipelining because not all of the data needs to be in the same stage at the same time. Therefore, the input can be larger than n. The data is broken up into n-sized chunks and sent through Pipelined Optimal BSR in consecutive stages. Imagine an army marching into Pipelined Optimal BSR in rank and file, sorting themselves by hat size. Suppose a bucket exists for each hat size. Each soldier drops his hat in the appropriate bucket as he exits. Pipelined Optimal BSR counts the hats as they are dropped, and the result is a histogram of hat sizes. In the above figure, the vertical rectangles represent columns of data lined up to enter Pipelined Optimal BSR. Each column will become a stage as it enters, and the columns can enter immediately, one after another, without having to wait for the previous one to exit, first. A pipelined algorithm has been achieved.

3.5.1.1 Histogram Complexity

A new variable n' is needed to describe the complexity of a pipelined algorithm in Pipelined Optimal BSR.
Figure 3.36 illustrates variable $n'$, which is the total number of datums which need to enter Pipelined Optimal BSR for a problem.

The number of stages which will be needed to solve the problem is $\lceil n'/n \rceil$. The first stage will take $O(\log^2(n + m))$ time to complete (see Section 1.4). The remaining stages will take $O(n'/n)$ time to complete. Therefore, the total time complexity is $O(\log^2(n + m) + n'/n)$ This is an improvement over the time complexity of a similar problem in Optimal BSR, which is $O((n'/n)\log^2(n + m))$, yet the cost remains the same at $O((n + m)(\log^2(n + m)))$.

Further increasing the speedup by $n$ has an additional cost of $O(n\log^2 n)$.

In an existing circuit, $n$ and $m$ will be constants, so the actual time becomes $O(n')$, which is not interesting asymptotically. However, Pipelined Optimal BSR might provide an answer to an
algorithm in 1 day instead of 30 days, which could be significant to the person waiting for the answer.

3.5.2 External Watchman Routes

External Watchman Routes from Algorithm 2.2.9, see Gewali[11], also pipelines well, even though it contains more than one instruction. A partial dedicated solution is to place two Pipelined Optimal BSR’s side by side, with the output of one being part of the input for the next. This algorithm can be used to compute collision-free paths with applications in robotics and graphics[11].

For clarity, the variable names are taken directly from this pseudocode in Gewali[11]:

\[ w_j = \bigcup s_i' | s_i' \geq s_j \]
\[ w_j = \bigcap q_i | s_i' = w_j \]

![Diagram](image_url)

Figure 3.37: Dual Pipelined Optimal BSR’s

Figure 3.37 illustrates pipelined External Watchman Routes, adapted from Gewali[11]. The inputs for the first instruction are \( s_i' \) and \( s_j \). While they are being processed, the future input \( q_i \) is kept in the same stages as its corresponding data by using registers, which are not shown. The results of the first
instruction, $W_j$, are sent directly into the second Pipelined Optimal BSR, along with the other appropriate inputs, for further processing. All of the stages can be used simultaneously in both copies of Pipelined Optimal BSR. Pipelining has been achieved with two instructions.

With External Watchman Routes, unlike Histogram, each stage is a different problem. The pipelining in this case is for numerous similar problems in immediate succession, such as for graphics.

The complexity of pipelined External Watchman Routes is asymptotically the same as for Histogram, except that, additionally, $n'$ could represent real-time incoming data.
CHAPTER 4 – RESULTS

Optimal BSR was converted to Pipelined Optimal BSR and a real-time interactive application of it was written in Java. Two algorithms, Histogram and External Watchman Routes, were found which take advantage of the pipelining. It was shown that multiple copies of Pipelined Optimal BSR can be used to process multiple instructions of the same algorithm simultaneously. Pipelined Optimal BSR improves the time complexity for Histogram and External Watchman Routes problems with large input sizes from $O((n'/n)\log^2(n + m))$ to $O(\log^2(n + m) + n'/n)$ yet the cost remains the same as Optimal BSR at $O((n + m)\log^2(n + m))$.

The following hypotheses were also considered:

Hypothesis 1.1 Memory, itself, does not need to store $R$, $\sigma$, and $I_j$ and an actual memory record does not need to be input into BSR.

Confirmed. See Issue 3.2.1.

Hypothesis 1.2 The BROADCAST instruction needs to specify the memory address to be accessed.

Confirmed. Algorithms with multiple instructions typically access more than one array in memory. The existing pseudocode has no way to indicate which array in memory is being accessed.

Hypothesis 1.3 Setup information should not be distributed throughout the circuit separately from the other records.

Confirmed. See Issue 3.2.9.

Hypothesis 1.4 Pipelining is possible.

Confirmed. See Figure 3.9.

Hypothesis 1.5 With pipelining, $R$ can be changed dynamically from within BSR.

Not confirmed. However, the Java code can obviously be changed to accomplish this. Further research should be done in this area.

Hypothesis 1.6 A BSR instruction set needs to be created.

Confirmed. The existing pseudocode cannot handle reads and memory locations of multiple arrays.

Hypothesis 1.7 Not all published BSR algorithms fit Optimal BSR.

Confirmed. See Algorithm 2.2.4, which uses an operator which is not available. Also, the $k > 1$
problems cannot be done.

Hypothesis 1.8 *Texture and luminosity graphics applications should be considered.*

Confirmed. Other graphics applications for BSR exist and these should also be considered.

Hypothesis 1.9 *Some reduction operations are never used.*

Confirmed. See Table 3.1.

Hypothesis 1.10 *A system-wide logical clock should be considered.*

Pipelined Optimal BSR was accomplished in Java without a system-wide logical clock. However, other systems may need one.

Hypothesis 1.11 *The object-oriented organization of the simulator should be explained.*

Confirmed. See Section 3.3.

Hypothesis 1.12 *Running a larger case on a supercomputer should be considered.*

Confirmed. Histogram and External Watchman Routes should be considered for supercomputer use. Further research to find other appropriate algorithms should also be done in this area.
CHAPTER 5 – CONCLUSIONS

5.1 General Conclusions

The Java applet of Pipelined Optimal BSR took another step towards developing an actual combinational circuit. The application demonstrated that Optimal BSR could be actualized, and that it could even be done with pipelining. Numerous algorithms have been published which could be run on such a circuit. Some sorting, graphics, and robotics problems, at least, could benefit from such a pipelined circuit, now with improved asymptotic times over Optimal BSR.

5.2 Summary of Contribution

Optimal BSR was Pipelined and programmed to a real-time interactive Java applet. Existing published algorithms were analyzed and summarized. Two existing algorithms, Histogram and External Watchman Routes, were found which are particularly adaptable to pipelining. One of these, External Watchman Routes, was used to demonstrate that multiple Pipelined Optimal BSR units could be used to execute multiple instructions simultaneously. The time complexity for some problems was improved from $O((n'/n)\log^2(n + m))$ to $O(\log^2(n + m) + n'/n)$ while retaining the same cost of $O((n + m)\log^2(n + m))$.

5.3 Future Research

BSR is a rich area for further research. The next obvious step is for an Electrical Engineer to begin a detailed design of the combinational circuit. The circuit might be on a separate chip, but, for more speed, would more likely reside on a CPU as a coprocessor. This could have dedicated applications in sorting, robotics, and graphics.

An instruction set needs to be created. Further algorithms might be found, particularly ones which consider changing the operator dynamically, and those that could benefit texture and luminosity graphics applications.

The existing Java applet I/O can only be done with dialog boxes. This should be changed so that it
gets input from files or the network, and so that it sends its results to files or the network.

Pipelined Optimal BSR should be considered on a macro level. For example, can the indexes represent internet addresses? Each node, for example, could be an agent.

Applications should be written for AKS Sorting and for $k > 1$ BSR problems.
APPENDIX
APPENDIX A – LONGER SOURCE CODE SEGMENTS

A.1 A Source Code Segment for Box A Nodes

    // BOX A -- sorts on tag and data
    if(box == BSRconstants.BOX_A) {
        data_output[1].i = data_input[1].i;
        data_output[1].t = data_input[1].t;
        data_output[1].r = data_input[1].r;
        data_output[1].d = data_input[1].d;
        if(data_input[0].t <= data_input[2].t) {
            if(data_input[0].t == data_input[2].t) {
                if(data_input[0].d <= data_input[2].d) {
                    data_output[0].i = data_input[0].i;
                    data_output[0].t = data_input[0].t;
                    data_output[0].r = data_input[0].r;
                    data_output[0].d = data_input[0].d;
                    data_output[2].i = data_input[2].i;
                    data_output[2].t = data_input[2].t;
                    data_output[2].r = data_input[2].r;
                    data_output[2].d = data_input[2].d;
                } else {
                    data_output[0].i = data_input[2].i;
                    data_output[0].t = data_input[2].t;
                    data_output[0].r = data_input[2].r;
                    data_output[0].d = data_input[2].d;
                    data_output[2].i = data_input[0].i;
                    data_output[2].t = data_input[0].t;
                    data_output[2].r = data_input[0].r;
                    data_output[2].d = data_input[0].d;
                } // if/else
            } else {
                data_output[0].i = data_input[0].i;
                data_output[0].t = data_input[0].t;
                data_output[0].r = data_input[0].r;
                data_output[0].d = data_input[0].d;
                data_output[2].i = data_input[2].i;
                data_output[2].t = data_input[2].t;
                data_output[2].r = data_input[2].r;
                data_output[2].d = data_input[2].d;
            } // if/else
        } else {
            data_output[0].i = data_input[2].i;
            data_output[0].t = data_input[2].t;
            data_output[0].r = data_input[2].r;
            data_output[0].d = data_input[2].d;
            data_output[2].i = data_input[0].i;
            data_output[2].t = data_input[0].t;
            data_output[2].r = data_input[0].r;
            data_output[2].d = data_input[0].d;
        } // if/else
    } // if
A.2 A Source Code Segment for Executing the Table

```c
if(switch_active) {
    active = true;
    table_operation = switch_operation.substring(0);
    table_comparator = switch_comparator.substring(0);
    if(table_operation.compareTo("$Sum") == 0)
        table_operator = "+";
    else if(table_operation.compareTo("Product") == 0)
        table_operator = "*";
    else if(table_operation.compareTo("Maximum") == 0)
        table_operator = "Max";
    else if(table_operation.compareTo("Minimum") == 0)
        table_operator = "Min";
    if(table_comparator.compareTo("<") == 0) {
        table_mode = "\";
        table_formula = "preceding";
    } // if(<)
    else if(table_comparator.compareTo(">") == 0) {
        table_mode = "\";
        table_formula = "gr\_equiv\_preceding";
    } // if(>)
```
table_formula = "1st_equiv\|following";
} // if(>=)
else if(table_comparator.compareTo("=") == 0) {
    if(table_operation.compareTo("Sum") == 0) {
        table_mode = "1";
        table_formula = "gr\_equiv-preceding";
    } // if(Sum)
    else if(table_operation.compareTo("Product") == 0) {
        table_mode = "1";
        table_formula = "gr\_equiv/preceding";
    } // if(Product)
    else if(table_operation.compareTo("Maximum") == 0) {
        table_mode = "4";
        table_formula = "gr\_equiv";
    } // if(Maximum)
    else if(table_operation.compareTo("Minimum") == 0) {
        table_mode = "3";
        table_formula = "gr\_equiv";
    } // if(Minimum)
} // if(!=)
else if(table_comparator.compareTo("!=") == 0) {
    if(table_operation.compareTo("Sum") == 0) {
        table_mode = "1";
        table_formula = "overall-gr\_equiv-preceding";
    } // if(Sum)
    else if(table_operation.compareTo("Product") == 0) {
        table_mode = "1";
        table_formula = "overall/gr\_equiv/preceding";
    } // if(Product)
    else if(table_operation.compareTo("Maximum") == 0) {
        table_mode = "5";
        table_formula = "gr\_equiv\|other";
    } // if(Maximum)
    else if(table_operation.compareTo("Minimum") == 0) {
        table_mode = "5";
        table_formula = "gr\_equiv\|other";
    } // if(Minimum)
} // if(<)
} // if

A.3 A Source Code Segment for Box B Nodes

if(operator_input.compareTo("=") == 0) {
    temp_data.i = data_input[1].i;
    temp_data.t = data_input[1].t;
    temp_data.r = data_input[1].r;
    temp_data.d = data_input[1].d;
    float sum_d = 0;
    for(int lcv = start; lcv <= finish; lcv++) {
if(data_input[lcv].i != -1)
    sum_d += data_input[lcv].d;
} // for
temp_data.d = sum_d;
} // if(+)
else if(operator_input.compareTo("=") == 0) {
    temp_data.i = data_input[i].i;
    temp_data.t = data_input[i].t;
    temp_data.r = data_input[i].r;
    temp_data.d = data_input[i].d;
    float prod_d = 1;
    for(int lcv = start; lcv <= finish; lcv++) {
        if(data_input[lcv].i != -1)
            prod_d *= data_input[lcv].d;
    } // for
    temp_data.d = prod_d;
} // if(*)
else if(operator_input.compareTo("Max") == 0) {
    temp_data.i = data_input[i].i;
    temp_data.t = data_input[i].t;
    temp_data.r = data_input[i].r;
    temp_data.d = data_input[i].d;
    float max_d = Float.NEGATIVE_INFINITY;
    for(int lcv = start; lcv <= finish; lcv++)
        if(data_input[lcv].i != -1)
            if(data_input[lcv].d > max_d)
                if(mode_input.compareTo("4") == 0)
                    if(temp_data.t == data_input[lcv].t)
                        max_d = data_input[lcv].d;
                else // not mode 4
                    max_d = data_input[lcv].d;
    temp_data.d = max_d;
} // if(Max)
else if(operator_input.compareTo("Min") == 0) {
    temp_data.i = data_input[i].i;
    temp_data.t = data_input[i].t;
    temp_data.r = data_input[i].r;
    temp_data.d = data_input[i].d;
    float min_d = Float.POSITIVE_INFINITY;
    for(int lcv = start; lcv <= finish; lcv++)
        if(data_input[lcv].i != -1)
            if(data_input[lcv].d < min_d)
                if(mode_input.compareTo("3") == 0)
                    if(temp_data.t == data_input[lcv].t)
                        min_d = data_input[lcv].d;
                else // not mode 3
                    min_d = data_input[lcv].d;
    temp_data.d = min_d;
} // if(Min)
data_output[0].i = temp_data.i;
data_output[0].t = temp_data.t;
data_output[0].r = temp_data.r;
data_output[0].d = temp_data.d;
data_output[1].i = temp_data.i;
data_output[1].t = temp_data.t;
data_output[1].r = temp_data.r;
data_output[1].d = temp_data.d;
data_output[2].i = temp_data.i;
data_output[2].t = temp_data.t;
data_output[2].r = temp_data.r;
data_output[2].d = temp_data.d;

A.4 A Source Code Segment for Box B Nodes in Mode 5

if(mode_input.compareTo("5") == 0) {
temp_data.i = data_input[1].i;
temp_data.t = data_input[1].t;
temp_data.r = data_input[1].r;
temp_data.d = data_input[1].d;
float lower_tag = -1; // tag lower than current Tag
float upper_tag = -1; // tag higher than current Tag
if(operator_input.compareTo("Max") == 0) {
    float maxa = Float.NEGATIVE_INFINITY;
    float maxb = Float.NEGATIVE_INFINITY;
    float maxc = Float.NEGATIVE_INFINITY;
    for(int lcv = start; lcv <= finish; lcv++) {
        if(data_input[lcv].i != -1) {
            if(data_input[lcv].t < temp_data.t)
                if(data_input[lcv].d > maxa) {
                    maxa = data_input[lcv].d;
                    lower_tag = data_input[lcv].t;
                } // if
            else if(data_input[lcv].t == temp_data.t)
                if(data_input[lcv].d > maxb)
                    maxb = data_input[lcv].d;
            else // if(data_input[lcv].t > temp_data.t)
                if(data_input[lcv].d > maxc) {
                    maxc = data_input[lcv].d;
                    upper_tag = data_input[lcv].t;
                } // if
            if(data_input[lcv].t < temp_data.t)
                if(data_input[lcv].d > maxa) {
                    maxa = data_input[lcv].d;
                    lower_tag = data_input[lcv].t;
                } // if
            else if(data_input[lcv].t == temp_data.t)
                if(data_input[lcv].d > maxb)
                    maxb = data_input[lcv].d;
            else // if(data_input[lcv].t > temp_data.t)
                if(data_input[lcv].d > maxc) {

}
maxc = data_input[lcv].d;
upper_tag = data_input[lcv].t;
} // if
if(data_inputc[lcv].t < temp_data.t)
if(data_inputc[lcv].d > maxa) {
maxa = data_inputc[lcv].d;
lower_tag = data_inputc[lcv].t;
} // if
else if(data_inputc[lcv].t == temp_data.t)
if(data_inputc[lcv].d > maxb)
maxb = data_inputc[lcv].d;
else // if(data_inputc[lcv].t > temp_data.t)
if(data_inputc[lcv].d > maxc) {
maxc = data_inputc[lcv].d;
upper_tag = data_inputc[lcv].t;
} // if
} // fi(valid input)
} // for(each input line)
temp_data.d = maxb;
data_output[1].i = temp_data.i;
data_output[1].t = temp_data.t;
data_output[1].r = temp_data.r;
data_output[1].d = temp_data.d;
temp_data.t = lower_tag;
data_output[0].i = temp_data.i;
data_output[0].t = temp_data.t;
data_output[0].r = temp_data.r;
data_output[0].d = temp_data.d;
temp_data.d = maxc;
temp_data.t = upper_tag;
data_output[2].i = temp_data.i;
data_output[2].t = temp_data.t;
data_output[2].r = temp_data.r;
data_output[2].d = temp_data.d;
} // fi(Max)
else if(operator_input.compareTo("Min") == 0) {
float mina = Float.POSITIVE_INFINITY;
float minb = Float.POSITIVE_INFINITY;
float minc = Float.POSITIVE_INFINITY;
for(int lcv = start; lcv <= finish; lcv++) {
if(data_input[lcv].i != -1) {
if(data_inputa[lcv].t < temp_data.t)
if(data_inputa[lcv].d < mina) {
mina = data_inputa[lcv].d;
lower_tag = data_inputa[lcv].t;
} // if
else if(data_inputa[lcv].t == temp_data.t)
if(data_inputa[lcv].d < minb)
minb = data_inputa[lcv].d;
else // if(data_inputa[lcv].t > temp_data.t)
if(data_input[lcv].d < minc) {
    minc = data_input[lcv].d;
    upper_tag = data_input[lcv].t;
} // if
if(data_input[lcv].t < temp_data.t)
if(data_input[lcv].d < mina) {
    mina = data_input[lcv].d;
    lower_tag = data_input[lcv].t;
} // if
else if(data_input[lcv].t == temp_data.t)
if(data_input[lcv].d < minb) {
    minb = data_input[lcv].d;
else // if(data_input[lcv].t > temp_data.t)
if(data_input[lcv].d < minc) {
    minc = data_input[lcv].d;
    upper_tag = data_input[lcv].t;
} // if
if(data_inputc[lcv].t < temp_data.t)
if(data_inputc[lcv].d < mina) {
    mina = data_inputc[lcv].d;
    lower_tag = data_inputc[lcv].t;
} // if
else if(data_inputc[lcv].t == temp_data.t)
if(data_inputc[lcv].d < minb) {
    minb = data_inputc[lcv].d;
else // if(data_inputc[lcv].t > temp_data.t)
if(data_inputc[lcv].d < minc) {
    minc = data_inputc[lcv].d;
    upper_tag = data_inputc[lcv].t;
} // if
} // for(each input line)
temp_data.d = minb;
data_output[1].i = temp_data.i;
data_output[1].t = temp_data.t;
data_output[1].r = temp_data.r;
data_output[1].d = temp_data.d;
temp_data.d = mina;
data_output[0].i = temp_data.i;
data_output[0].t = temp_data.t;
data_output[0].r = temp_data.r;
data_output[0].d = temp_data.d;
temp_data.d = minc;
data_output[2].i = temp_data.i;
data_output[2].t = temp_data.t;
data_output[2].r = temp_data.r;
data_output[2].d = temp_data.d;
} // fi(Min)
} else {
    // not Mode 5 ...
}
A.5 A Source Code Segment for Box E Switches

```java
public void execute(String mode, String operator, String formula) {
    if(active) {
        data_output[1].i = data_input[1].i;
        data_output[1].t = data_input[1].t;
        data_output[1].r = data_input[1].r;
        data_output[1].d = data_input[1].d;
        ...
        if(box == BSlotConstants.BOX_E && data_input[1].r < 0) {
            if(formula.compareTo("1st_equiv|following") == 0) {
                if(edata.firstr == 9) {
                    if(edata.followingr == 9) {
                        // there is no data
                        if(operator.compareTo("=") == 0) {
                            data_output[1].d = 0;
                        } else {
                            // invalid data
                            data_output[1].r = 2;
                        } // fi/else(Sum)
                    } else {
                        // first is not valid, but following is valid
                        data_output[1].d = edata.following;
                    } // fi(following is valid or not)
                } // fi/else(firstr == 9)
                } // fi(first)
            else if(formula.compareTo("gr\_equiv") == 0) {
                if(edata.greatestr == 0) {
                    // invalid data
                    data_output[1].r = 2;
                } else {
                    data_output[1].d = edata.greatest;
                } // fi/else(greatestr == 0)
            } // fi(greatest)
            else if(formula.compareTo("gr\_equiv|preceding") == 0) {
                if(edata.greatestr == 0) {
                    if(edata.precedingr == 0) {
                        // there is no data
                        if(operator.compareTo("=") == 0) {
                            data_output[1].d = 0;
                        } else {
                            // invalid data
                            data_output[1].r = 2;
                        } // fi/else(Sum)
                    } else {
                        // greatest is invalid, but preceding is valid
                        data_output[1].d = edata.preceding;
                    } // fi(preceding is valid or not)
                } else {
        ```
data_output[1].d = edata.greatest;
} // fi/else(greatestd == 0)
} // fi(greatest|preceding)
else if(formula.compareTo("gr\_equiv\|other") == 0) {
    if(edata.greatestd == 0 ||
        edata.greatest == Float.POSITIVE_INFINITY ||
        edata.greatest == Float.NEGATIVE_INFINITY ) {
        // greatest is not valid
        // find the appropriate value from preceding or following
        if(edata.preceding == 0 ||
            edata.preceding == Float.POSITIVE_INFINITY ||
            edata.preceding == Float.NEGATIVE_INFINITY ) {
            // preceding is not valid
            if(edata.followingd == 9 ||
                edata.following == Float.POSITIVE_INFINITY ||
                edata.following == Float.NEGATIVE_INFINITY ) {
                // both are not valid
                // invalid data
                data_output[1].d = 2;
            } else {
                // only following is valid
                data_output[1].d = edata.following;
            } // fi(following is not valid or valid)
        } else {
            // preceding is valid
            if(edata.followingd == 9 ||
                edata.following == Float.POSITIVE_INFINITY ||
                edata.following == Float.NEGATIVE_INFINITY ) {
                // only preceding is valid
                data_output[1].d = edata.preceding;
            } else {
                // both are valid
                // apply the correct operator
                if(operator.compareTo("Max") == 0) {
                    if(edata.preceding > edata.following) {
                        data_output[1].d = edata.preceding;
                    } else {
                        data_output[1].d = edata.following;
                    } // fi (preceding is greater)
                } // fi(Max)
                else if(operator.compareTo("Min") == 0) {
                    if(edata.preceding < edata.following) {
                        data_output[1].d = edata.preceding;
                    } else {
                        data_output[1].d = edata.following;
                    } // fi (preceding is greater)
                } // fi(Min)
            } // fi(following is not valid or valid)
        } // fi(preceding is not valid or valid)
    } else {
        // greatest is valid
data_output[1].d = edata.greatest;
} //fi/else(greatestr == 0)
} // fi(greatest|other)
else if(formula.compareTo("gr\_equiv-precending") == 0) {
    if(edata.greatestr == 0) {
        // greatest is invalid
        data_output[1].r = 2;
    } else if(edata.precedingr == 0) {
        // greatest is OK, but preceding is invalid
        data_output[1].d = edata.greatest;
    } else {
        // both are OK
        data_output[1].d = edata.greatest - edata.preceding;
    } // fi/else(greatestr == 0)
} // fi(greatest - preceding)
else if(formula.compareTo("gr\_equiv/precending") == 0) {
    if(edata.greatestr == 0) {
        // greatest is invalid
        data_output[1].r = 2;
    } else if(edata.precedingr == 0) {
        // greatest is OK, but preceding is invalid
        data_output[1].d = edata.greatest;
    } else {
        // both are OK
        data_output[1].d = edata.greatest / edata.preceding;
    } // fi(greatestr == 0)
} // fi(greatest / preceding)
else if(formula.compareTo("preceding") == 0) {
    if(edata.precedingr == 0) {
        // there is no data in preceding
        if(operator.compareTo("+/") == 0) {
            data_output[1].d = 0;
        } else {
            // invalid data
            data_output[1].r = 2;
        } // fi/else(Sum)
    } else {
        data_output[1].d = edata.preceding;
    } // fi/else(precedingr == 0)
} // fi(preceding)
else if(formula.compareTo("following") == 0) {
    if(edata.followingr == 9) {
        // there is no data in following
        if(operator.compareTo("+/") == 0) {
            data_output[1].d = 0;
        } else {
            // invalid data
            data_output[1].r = 2;
        } // fi/else(Sum)
    } else {
        data_output[1].d = edata.following;
    } // fi/else(followingr == 0)
} // fi(following)
A.6 A Source Code Segment for Box E Nodes

// Compare limits for Box E
if (whoami == BSRconstants.BOX_E && lcv > 15 && node[lcv].active) {
  edata0[lcv].first = edata[line0_source].first;
  edata0[lcv].firstr = edata[line0_source].firstr;
  edata0[lcv].greatest = edata[line0_source].greatest;
  edata0[lcv].greatestr = edata[line0_source].greatestr;
  edata0[lcv].preceding = edata[line0_source].preceding;
  edata0[lcv].precedingr = edata[line0_source].precedingr;
  edata0[lcv].following = edata[line0_source].following;
  edata0[lcv].followingr = edata[line0_source].followingr;
  edata0[lcv].overall = edata[line0_source].overall;
  edata0[lcv].overallr = edata[line0_source].overallr;
} // fi
edata[lcv].first = edata[line1_source].first;
edata[lcv].firststr = edata[line1_source].firststr;
edata[lcv].greatest = edata[line1_source].greatest;
edata[lcv].greatestr = edata[line1_source].greatestr;
edata[lcv].preceeding = edata[line1_source].preceeding;
edata[lcv].preceedingr = edata[line1_source].preceedingr;
edata[lcv].following = edata[line1_source].following;
edata[lcv].followingr = edata[line1_source].followingr;
edata[lcv].overall = edata[line1_source].overall;
edata2[lcv].first = edata[line2_source].first;
edata2[lcv].firststr = edata[line2_source].firststr;
edata2[lcv].greatest = edata[line2_source].greatest;
edata2[lcv].greatestr = edata[line2_source].greatestr;
edata2[lcv].preceeding = edata[line2_source].preceeding;
edata2[lcv].preceedingr = edata[line2_source].preceedingr;
edata2[lcv].following = edata[line2_source].following;
edata2[lcv].followingr = edata[line2_source].followingr;
edata2[lcv].overall = edata[line2_source].overall;
edata2[lcv].overallr = edata[line2_source].overallr;
if(line0_open) {
    if(line0_data.t < line1_data.t) {
        if(edata[line0_source].greatestr > edata[lcv].preceedingr) {
            edata[lcv].preceeding = edata[line0_source].greatest;
edata[lcv].preceedingr = edata[line0_source].greatestr;
        } // fi(line0 greatestr > current greatestr)
        if(edata[line0_source].preceedingr > edata[lcv].preceedingr) {
            edata[lcv].preceeding = edata[line0_source].preceeding;
edata[lcv].preceedingr = edata[line0_source].preceedingr;
        } // fi(line0 preceedingr > current preceedingr)
    } // fi(line0 tag < line1 tag)
    if(line0_data.t == line1_data.t) {
        if(edata[line0_source].firststr != -1) {
            if(edata[line0_source].firststr < edata[lcv].firststr) {
                edata[lcv].first = edata[line0_source].first;
edata[lcv].firststr = edata[line0_source].firststr;
            } // fi(new first data)
        } // fi(line0 first source is valid)
        if(edata[line0_source].greatestr > edata[lcv].greatestr) {
            edata[lcv].greatest = edata[line0_source].greatest;
edata[lcv].greatestr = edata[line0_source].greatestr;
        } // fi(line0 greatestr > current greatestr)
        if(edata[line0_source].preceedingr > edata[lcv].preceedingr) {
            edata[lcv].preceeding = edata[line0_source].preceeding;
edata[lcv].preceedingr = edata[line0_source].preceedingr;
        } // fi(line0 preceedingr > current preceedingr)
        if(edata[line0_source].followingr != -1) {
            if(edata[line0_source].followingr < edata[lcv].followingr) {
                edata[lcv].following = edata[line0_source].following;
edata[lcv].followingr = edata[line0_source].followingr;
            } // fi(new following data)
if(line0_tag == line1_tag)
if(edata[line0_source].overallr > edata[lcv].overallr) {
  edata[lcv].overall = edata[line0_source].overall;
  edata[lcv].overallr = edata[line0_source].overallr;
} // fi(overall>)
} // if(line0_open)
if(line2_open) {
  if(line2_data.t > line1_data.t) {
    if(edata[line2_source].firstr != -1) {
      if(edata[line2_source].firstr < edata[lcv].followingr) {
        edata[lcv].following = edata[line2_source].first;
        edata[lcv].followingr = edata[line2_source].firstr;
      } // fi(line2 firstr < current followingr)
      } // fi(line2 is valid)
      if(edata[line2_source].followingr != -1) {
        if(edata[line2_source].followingr < edata[lcv].followingr) {
          edata[lcv].following = edata[line2_source].following;
          edata[lcv].followingr = edata[line2_source].followingr;
        } // fi(new following data)
      } // fi(line2 following source is valid)
    } // fi(line2 tag > line 1 tag)
  } // fi(line2_data.t == line1_data.t)
  if(edata[line2_source].firstr != -1) {
    if(edata[line2_source].firstr < edata[lcv].firstr) {
      edata[lcv].first = edata[line2_source].first;
      edata[lcv].firstr = edata[line2_source].firstr;
    } // fi(line2 firstr < current firstr)
  } // fi(line2 is valid)
  if(edata[line2_source].greatestr > edata[lcv].greatestr) {
    edata[lcv].greatest = edata[line2_source].greatest;
    edata[lcv].greatestr = edata[line2_source].greatestr;
  } // fi(line2 greatestr > current greatestr)
  if(edata[line2_source].precedingr > edata[lcv].precedingr) {
    edata[lcv].preceding = edata[line2_source].preceding;
    edata[lcv].precedingr = edata[line2_source].precedingr;
  } // fi(line2 precedingr > current precedingr)
  if(edata[line2_source].followingr != -1) {
    if(edata[line2_source].followingr < edata[lcv].followingr) {
      edata[lcv].following = edata[line2_source].following;
      edata[lcv].followingr = edata[line2_source].followingr;
    } // fi(new following data)
  } // fi(line2 following source is valid)
} // fi(line2 tag == line 1 tag)
if(edata[line2_source].overallr > edata[lcv].overallr) {
  edata[lcv].overall = edata[line2_source].overall;
  edata[lcv].overallr = edata[line2_source].overallr;
} // fi(overall>)
} // if(line2_open)
} // if
BIBLIOGRAPHY


VITA

Graduate School
Southern Illinois University

Chester L. Langin
P.O. Box 1262, Carbondale, IL 62903

Southern Illinois University, Carbondale, IL
Bachelor of Science, Journalism, June 1974

Thesis Title:
  Pipelined Optimal Broadcast with Selective Reduction (BSR)

Major Professor: Dr. Chih-Fang Wang

Publication: